Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 317 (2015) 117–131

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Certifying Square Root and Division Elimination

Pierre Neron [1]

T.U. Delft
Delft, Netherlands

Abstract

This paper presents the implementation of a program transformation that removes square roots and divisions from functional programs without recursion, producing code that can be exactly computed. This transformation accepts different subsets of languages as input and it provides a certifying mechanism when the targeted language is Pvs. In this case, we provide a relation between every function definition in the output code and its corresponding one in the input code, that specifies the behavior of the produced function with respect to the input one. This transformation has been implemented in OCaml and has been tested on different algorithms from the NASA ACCoRD project.

Keywords: Program transformation; Real number computation; Certifying transformation; Semantics preservation

1 Introduction

Critical embedded systems, for example in aeronautics, require a very high level of safety. One approach to produce code that may satisfy this required level of safety is to verify its correctness in a proof assistant such as Pvs. The embedded systems do not run the Pvs code but from the proved Pvs specification we can extract a corresponding program in a real language that corresponds to this specification, (see [6] for an example of extraction).

However, these embedded systems may also be cyber-physical systems and therefore have an extended use of mathematical operations over real numbers that can not be computed exactly. In particular, this is a problem if we aim at satisfying the usual requirements of embedded systems, *i.e.,* bounded memory and bounded loops. Indeed, some methods have been developed to compute exactly with real numbers (see [2, 17, 18]) or sufficient precision using lazy evaluation (see [13]) but these techniques usually involve unbounded behaviors. Therefore, for embedded

[1] Email: p.j.m.neron@tudelft.nl

http://dx.doi.org/10.1016/j.entcs.2015.10.012

systems, one usually rely on a finite representation and an analysis of rounding errors via abstract interpretation or interval arithmetic [3,5]. Alternatively, one might want to directly prove the correctness properties not on real numbers but on the effective implementation the system uses, *e.g.,* floating point numbers [1], but in this case the proofs become very difficult since any mathematical intuition is lost.

Aeronautics embedded systems, for example, use square root and divisions in conflict detection and resolution algorithms. These operations can not be exactly computed in a finite memory since they may produce infinite sequence of digits. This is not the case for addition and multiplication. These operations only produce finite sequences of digits and therefore they can be exactly computed using some fixed point representation. Determining the required size for this fixed point representation is relatively easy in embedded systems with only bounded loops. This paper presents a program transformation tool that eliminates these square roots and divisions, this transformation allows the extracted code to be exactly computed. The transformation also provides a certifying mechanism [16] to prove the semantics preservation.

This paper focuses on the system description and the implementation aspects. The theoretical aspects of this work are presented in [9–11]. The paper is structured as follows. Section 2 focuses on the main implementation of the transformation in OCaml. Section 3 describes the embedding of a subset of Pvs to provide the certifying process. Section 4 introduces some of the technical details and features of the transformation. Section 5 presents an application of this transformation to a conflict detection algorithm from the ACCoRD [2] framework.

2 The OCaml Transformation

2.1 Language

The program transformation is defined in OCaml and operates on a language denoted MiniPvs that is a typed functional language containing numerical ($\mathbb{R}$) and Boolean constants, tests (if then else), pairs, the usual arithmetic operators $+$, $-$, $\times$, $/$, $\sqrt{}$, the comparisons $=$, $\neq$, $>$, $\geq$, $<$, $\leq$, Boolean operators ($\wedge$, $\vee$, $\neg$), variable and function definition and application. Figure 1 presents the OCaml definition of the language as an abstract datatype where uvar is the set of variable identifiers.

The semantics of such a language is quite straightforward. The expression Letin x body scope is interpreted as *let* x = *body in scope* and Letfun f (v,tv) t body scope is the definition of the function taking v as argument of type tv and returning an element of type t, *i.e., let* f $(v : tv) : t$ = *body in scope*; their semantics use call by value. The detailed semantics of this language can be found in [11], Chapter 3 and 4. We denote $[\![\mathsf{p}]\!]_{Env}$ the semantics of a the program p in the environment Env. Function and variable definitions allow for multi-variable definitions (*e.g.,*let f (x,y) = x + y) but partial application is not allowed. This language can represent a subset of many programming language and programs written in such a subset can

[2] http://shemesh.larc.nasa.gov/people/cam/ACCoRD/

```
type prog =
  | Value of uvar
  | Const of constants
  | UOp of unop * prog
  | BOp of binop * prog * prog
  | Pair of prog * prog
  | Fst of prog
  | Snd of prog
  | If of prog * prog * prog
  | App of uvar * prog
  | Letin of var * prog * prog
  | Letfun of uvar * (var * types) * types * prog * prog
type unop =
  | Sqrt | Umin | Neg
type binop =
  | Plus | Times | Div | And | Or
  | Neq | Eq | Gt | Geq | Lt | Leq
type var =
  | Uvar of uvar | Pairvar of var * var
```

Fig. 1. MiniPvs Abstract Syntax

be translated using our transformation by building an input and output interfaces (*i.e.,* a parser and a pretty printer).

2.2 Transformation Specification

The goal of this transformation is to remove square roots and divisions from the programs defined in this language. The transformation of a program relies on 2 distinct steps.

The first one is the transformation of all the variable definitions and functions, to ensure that the values of variables or function calls that may appear in an expression do not depend on square roots or divisions. This transformation relies on an anti-unification algorithm to partially inline the variable and function definitions as in the following example:

$$\begin{array}{l}\mathsf{let\ x} = \\ \mathsf{if\ F\ then}\ a + \sqrt{b \cdot c + d} \\ \quad\ \mathsf{else}\ e \cdot f/(g+h) \\ \mathsf{in\ SC}\end{array} \longrightarrow \begin{array}{l}\mathsf{let}\ (\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3) = \\ \quad \mathsf{if\ F\ then}\ (a, b \cdot c + d, 1) \\ \quad\quad \mathsf{else}\ (e \cdot f, 0, g + h) \\ \mathsf{in\ SC}[\mathsf{x} := (\mathsf{x}_1 + \sqrt{\mathsf{x}_2})/\mathsf{x}_3]\end{array}$$

The anti-unification problem introduced independently by Reynolds [15] and Plotkin [14] in 1970 is the dual of the unification problem. Given two terms t_1 and t_2, an anti-unification algorithm computes a term t and two substitutions σ_1 and σ_2 such that $t\sigma_1 = t_1$ and $t\sigma_2 = t_2$. In the previous example, the term $(\mathsf{x}_1 + \sqrt{\mathsf{x}_2})/\mathsf{x}_3$ (denoted t) is a template of $a + \sqrt{b \cdot c + d}$ and $e \cdot f/(g + h)$ since we have the

following equalities:

$$t[\mathsf{x}_1 \mapsto a; \mathsf{x}_2 \mapsto b \cdot c + d; \mathsf{x}_3 \mapsto 1] = a + \sqrt{b \cdot c + d}$$
$$t[\mathsf{x}_1 \mapsto e \cdot f; \mathsf{x}_2 \mapsto 0; \mathsf{x}_3 \mapsto (g + f)] = e \cdot f/(g + f)$$

The second step of the transformation is the elimination of square roots and divisions in Boolean expressions. It relies on the succesive application of simple transformation rules for comparisons between arithmetic expressions that eliminate all the square roots and divisions. These rules relies on usual equivalence of such comparisons, e.g.

$$p \cdot \sqrt{q} = r \iff p \cdot r \geq 0 \wedge p^2 \cdot q = r^2$$
$$a/b > c \iff b > 0 \wedge a > b \cdot c \vee b < 0 \wedge a < b \cdot c$$

Of course we can not remove square roots and divisions from all programs, (*e.g.,* the program sqrt(2) will still have to return a rounded value of $\sqrt{2}$). However, when square roots and divisions are eliminated from all the Boolean expressions and all the definitions, the Boolean values of such a transformed program can be exactly computed. Thus the control flow of the program is protected from any rounding.

Avoiding rounding errors in the control flow of a program prevents divergence in its behavior. An error in the condition of an *if then else* statement can provoke the execution of a completely different syntactic part of the program and therefore provoke huge errors between the effective and the expected result. For example, if executed with a standard implementation of floating point numbers, the following program would return 1000:

$$\mathit{if}\ \sqrt{2} * \sqrt{2} > 2\ \mathit{then}\ 1000\ \mathit{else}\ 0$$

This kind of divergence is no longer possible if the Boolean expressions deciding the control flow of the program are computed exactly.

The transformation of Boolean expressions is described in [10] and the transformation of definitions using anti-unification and partial inlining is described in [9]. This transformation preserves the semantics of the program in every environment where the semantics does not fail due to division by zero or square root or negative numbers.

Definition 2.1 Semantics preservation Let tr be a program transformation transformation and let $[\![\mathsf{p}]\!]_{Env}$ be the semantics of p in the environment Env (the environment maps identifier to values); tr preserves the semantics if and only if for all programs p and $\mathsf{p'}$ be such that $\mathsf{p'} = tr(\mathsf{p})$:

$$\forall\ Env,\ [\![\mathsf{p}]\!]_{Env} \neq Fail \Longrightarrow [\![\mathsf{p}]\!]_{Env} = [\![\mathsf{p'}]\!]_{Env}$$

We do not consider the cases when the input program fails since we do not want to throw explicit errors due to division by zero in the input program that do not make sense in the output. Moreover the programs our transformation target have already been proved free of such errors (in PVS it is not allowed to write `sqrt(x)` without proving that this `x` is positive).

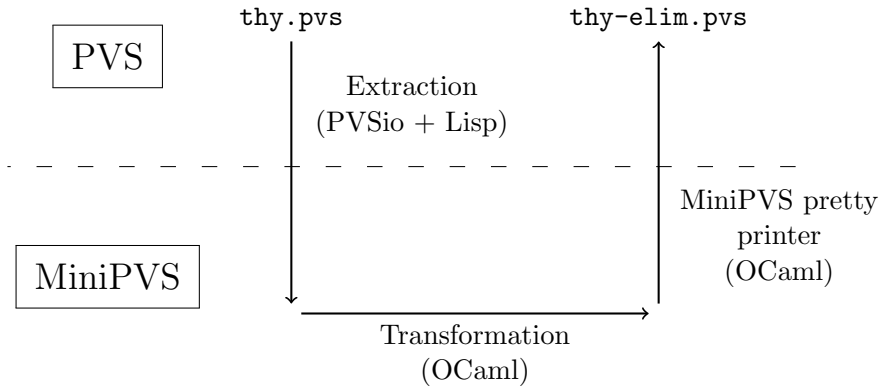

Fig. 2. PVS theory transformation

The semantics preservation has been proven in Pvs for the Boolean expression transformation but the anti-unification is not certified and thus neither are the variable and function definitions transformations. However, while an anti-unification algorithm is not easy to be proven correct, its result is quite easy to check so we can use this property to define a certifying transformation. Instead of formally proving that the transformation algorithm is correct we provide with each transformed program, a proof that this program is equivalent to the input program.

3 Transformation of Pvs specifications

In this section, we present how we have been able to use the transformation defined in OCaml on the language described in Figure 1, *i.e.,* MiniPvs, to transform Pvs specifications. The language of Pvs is much more complex that MiniPvs, however this language embeds enough constructions to represent the subset of the Pvs language that includes non-recursive function definitions and real valued expressions. Thus the first step of the transformation is to translate Pvs specification to the prog type of OCaml. In particular, we have to erase most of the logical part of the Pvs specifications to make it fit the prog type. We are not interested in all the sub-typing predicates that are encoded in the type of the Pvs functions. This process is similar to the usual extraction of code in proof assistants (*e.g.,* [7]). This extraction produces a program in the concrete syntax corresponding to MiniPVS that the OCaml program is able to process as an input.

The first step of the transformation is achieved by using the PVSio utility of Pvs [8]. PVSio is a package that extends the ground evaluator with a predefined library of imperative programming language features such as side effects, unbounded loops, input/output operations, floating point arithmetic, exception handling, pretty printing, and parsing. This package allows us to access the underlying Lisp structure representing the Pvs code, and therefore to print the corresponding program in MiniPVS that can be parsed by the OCaml implementation. The transformation scheme of a Pvs theory is presented in Figure 2

However, there is one subtle difference between PVS and MiniPVS. On one side, in MiniPVS, every definition has a scope and thus every program is complete. This allows us to specify that the transformation is correct by stating that the semantics of such a program is preserved, as stated in Definition 2.1. On the other side, a PVS file is a sequence of definitions of variables, functions and lemmas without any scope that would allow us to state the semantics preservation. In [9], the mechanism introduced to transform definitions in this program transformation, provides a correctness relation between every input definition and its corresponding definition in the output program. Thus, we extend the MiniPVS language by adding a constructor that adds a sub-typing predicate to every transformed definition. :

```
| Letfst of uvar * (var * types) *
                   (var * types * prog) * prog * prog
```

This constructor has to be interpreted in the following way, Letfst *f (u,tu) (u,tu,pu) body scope* represents the function definition:

$$let\ f\ (\ v : tv\ ) : \{\ u : tu \mid P(v,u)\ \} = body\ in\ scope$$

that is, a function taking argument v of type tv and returning an element of type tu such that $P(v, f(v))$. $\{x : T \mid P(x)\}$ denotes the subtype of T of elements satisfying P. This notation is similar to the PVS syntax for subtypes. Thus the subtyping predicate will be used to encode the relation between the definition of the output program with respects to the corresponding definition of the input one.

Example 3.1 The following PVS function and its call:

```
f(x : real ) : real = (x - 1) / (x + 1)

... f(c + sqrt(d))
```

are transformed into the following code:

```
f_e(x1,x2 : real) : {z1, z2,z3 : real |
         (z1 + sqrt(z2)) / (z3 + sqrt(z2)) = f(x1 + sqrt(x2)) } =
    (x1-1, x2, x1+1)

... f_e(c,d)
```

In a more general way, the transformation of a PVS function definition has the following form:

```
f(x : T1 | P(x)) : { y : T2 | Q(y) } = ...
```

is transformed into a corresponding definition:

```
f_e(x' : T1') : {y' : T2' | g_o(y') = f(g_i(x')) } = ...
```

The predicate `g_o(y') = f_in(g_i(x'))` is the relation specifying the behavior of the output function (`f_out`) relatively to the one of the input (`f_in`). `g_i(x')` and `g_o(y')` are terms computed by the transformation algorithm that depends on the underlying anti-unification process (see [9] for details). The function `f_in` appearing in the predicate is the one from the input program. The input program is imported into the transformed one to verify that the transformation is correct. The variables

defined in the input program are only used in the subtyping predicates not in the executable part of the transformed program. One can notice that in general the input and output types of the input and output functions do not match. However the transformation of Boolean functions does not change the output type, in such a case, the transformation will have the following shape:

Example 3.2 Given PVS function returning a Boolean expression:

```
f(x : T1 | P(x)) : { y : bool | Q(y) } = ...
```

Its corresponding transformed definition has the following form :

```
f_e(x' : T1') : {y' : bool | y' = f(g_i(x')) } = ...
```

The input type can still be transformed since the input can be numerical values and thus this function can be applied to square roots or divisions which would enforce the change of the input type.

Embedding a correctness lemma for each transformed definition allows us to transform usual PVS theories that consist of sequences of definitions and to prove the equivalence without requiring a returned expression (a scope for this definitions), as it is required for the MiniPVS language (in this language every definition has a scope). The problem of the absence of scope in a real PVS specification is overcome by using a simple fresh variable as the scope of the corresponding MiniPVS program. The definitions being transformed and the correctness being established independently from the scope transformation, the scope transformation is simply the identity (a free variable is transformed into itself) and thus we can erase this artificial scope when we print the output PVS program.

The only step left to formally verify that the transformed program is correct is to prove the lemmas corresponding to the subtyping predicates that are generated in the output program. PVS already decomposes the different cases that might appear in the function bodies (corresponding to *if then else* expressions), thus the only lemmas that we need to prove are equalities of arithmetic expressions containing square roots and divisions *e.g.,*

Example 3.3 The proof of the subtyping predicates from example 3.1 relies on the following equality:

```
((x1 - 1) + sqrt(x2))/((x1 + 1) + sqrt(x2)) = f(x1 + sqrt(x2))
```

In most cases, a simple PVS strategy for arithmetic such as `(grind-reals)` is enough. The more complicated cases might require first an elimination of square roots and divisions appearing in these predicates. This can be done using the PVS strategy `(elim-sqrt)` that eliminates square roots and divisions from inequalities in the proof context of PVS. This strategy is derived from the certified elimination of square roots and divisions in Boolean expressions as described in [12]. This tools lift this transformation to the program level by allowing the direct treatment of variable and function definitions and certifying it.

The general principle of the transformation code being outlined, let us now focus on the different features of the OCaml transformation.

4 The transformation of MiniPvs

The OCaml implementation of the transformation embeds a few options that aim at reducing the size of the produced code in some particular cases.

4.1 Rules for Boolean expressions

As mentioned previously the program transformation relies on a square root and division elimination in Boolean expressions. This elimination is described in [10] but in some places different rules can be preferred. For example the elimination of divisions in a comparison can be done using case distinction or by using the square of the denominator, *e.g.,*

Example 4.1 Assuming $B \neq 0$, the following equivalences hold:

$$A/B > C \iff (B > 0 \wedge A > B \times C) \vee (B < 0 \wedge A < B \times C) \quad (1)$$

$$A/B > C \iff A \times B > C \times B \times B \quad (2)$$

Elimination (1) produces smaller arithmetic terms, thus the size of the fixed-point representation for exact computation with $+, \times, -$ is smaller whereas rule (2) produces less comparisons and smaller formulas.

Depending on its objectives, the user might prefer one elimination scheme to another for division elimination, thus the choice of the rule is an option of the transformation.

In a similar way the elimination of square root can use variable definition inside Boolean expressions to reduce the size of the output term, since its complexity is exponential in the number of square roots in the expression. However the user might want to use the transformation only for expressions built with Boolean and arithmetic operators and restrict the transformation to such an expression language. Thus the transformation provides two schemes for square root elimination.

4.2 Transformation of comparison operators

The elimination of square roots and divisions in Boolean expressions having an exponential complexity, the transformed expressions greatly decrease the readability of the output code, these eliminations producing large Boolean expressions. In order to resolve this issue, we propose to handle this complexity in a different file by defining template specific comparison expressions. The transformation of functions using anti-unification is efficient regarding the size of the produced code and the equivalence lemmas are relatively easy to prove using the `(elim-sqrt)` strategy. Therefore we decided to use this function transformation in order to factorize the large Boolean expressions produced by the elimination of square roots in Boolean. This is done by first replacing the comparisons operators in the input program by functions that have the same semantics before applying the transformation, *e.g.,* the program from Figure 3 is transformed into the one in Figure 4.

```
f(x1,y1 : posreal) : bool = x1 + sqrt(y1) * y1 > 0

g(x,y, z : posreal) : real =
  IF z + sqrt(y + x) > 1 OR f(y,z)
  THEN y
  ELSE sqrt(x) + y
  ENDIF
```

Fig. 3. Program using $>$ operator

```
gt1(gt1l,gt1r : real) : bool = gt1l > gt1r

gt2(gt2l,gt2r : real) : bool = gt2l > gt2r

f(x1,y1 : posreal) : bool = gt1(x1 + sqrt(y1) * y1,0)

g(x,y, z : posreal) : real =
  IF gt2(z + sqrt(y + x),1) OR f(y,z)
  THEN y
  ELSE sqrt(x) + y
  ENDIF
```

Fig. 4. Program with $>$ declared as function

This pre-processing produces one comparison function per comparison operator used in the program. This process is required to produce comparisons function whose specification exactly match the required use whereas creating only one comparison function would produce a transformed function whose specification has to match all the use cases. The transformation of Boolean function being the cause of the code size blowup we want to avoid such a generic function that would have a huge definition body.

However since we create a new function for each comparison operator in the input program we introduce some redundancy. In program in Figure 3 both of the comparisons are applied to expressions that use only one square root. Thus both of the comparisons have the following form: $t + u.\sqrt{v} > 0$ with t, u and v being square root and division free expressions. Therefore the transformed functions corresponding to `gt1` and `gt2` will have the same specification and definition (modulo α-equivalence), thus we can only use one of them and the program is re-factorized in order to only use one of the equivalent functions. Our transformation produces the program in Figure 5 that only contains one comparison function, namely `gt0_e` taking four arguments. This function that has the following specification encoded in its type:

$$\texttt{gt0\_e}(x, y, z, sq) \Leftrightarrow x + y\sqrt{sq} > z$$

The definition body of this function is much larger than the one of the input function (*i.e.*, $x > y$), however the result is now computed without any square root or division. Therefore this transformation of Boolean expressions with functions requires only an automatic pre and post process of the transformation:

- Before applying the main transformation, replace every occurrence of the comparison operators by a function whose definition is this operator.
- Apply the main transformation.
- For all the transformed functions corresponding to the comparison operator, factorize the ones that have the same specification.

```
gt0_e(gt0_1_1, gt0_1_2, gt0_2, sq_2 : real) :
 {res : bool |
     res = gt0_1_1 + gt0_1_2 * sqrt(sq_2) > gt0_2} =
 LET (at_p, at_r, at_rel, at_neq) =
  (gt0_1_2 > 0, gt0_1_1 - gt0_2 > 0,
   gt0_1_2*gt0_1_2*sq_2 -
            (gt0_1_1 - gt0_2)*(gt0_1_1 - gt0_2) > 0,
   gt0_1_2*gt0_1_2*sq_2 -
            (gt0_1_1 - gt0_2)*(gt0_1_1 - gt0_2) /= 0)
 IN
     at_p AND at_r OR
     at_p AND at_rel OR
     at_r AND NOT at_rel AND at_neq

f_e(x1, y1 : real) : {res : bool | res = f((x1, y1))} =
 gt0_e((x1, y1, 0, y1))

g_e(x, y, z : real) : {g_1, g_2, sq_0 : real |
  g_1 + g_2 * sqrt(sq_0) = g((x, y, z))} =
 IF gt0_e((z, 1, 1, y + x)) OR f_e((y, z))
 THEN (y, 0, 0)
 ELSE (y, 1, x)
 ENDIF
```

Fig. 5. Transformed program with comparison function

This transformation greatly reduces the size of the output file since the large expressions corresponding to the transformation of Boolean expression are now factorized in functions. Moreover these new comparisons functions can also be generated in a separate file and then imported in the transformed program and shared between different transformed program. In this way the transformed programs have exactly the same structure as the input one.

In order to illustrate our transformation on a real example, we present in Section 5 the transformation of a real PVS program for conflict detection in two dimensions.

5 Application

A complete PVS specifications can be processed by the OCaml implementation of the transformation linked with PVSio. In this section we present the transformation of a conflict detection algorithm, namely cd2d, that has been developed by NASA in the ACCoRD framework. This algorithm aims at detecting loss of separation between two aircrafts in a two-dimensional space. A formal verification of this algorithm assuming floating point arithmetic has been presented in [4], the algorithm is described in that paper but we recall its main characteristics.

Coordinates of the aircraft are represented relatively thus, given $s_1 = (x_1, y_1)$ and $s_2 = (x_2, y_2)$ the positions in two dimension of the aircrafts, $s = (x_1 - x_2, y_1 - y_2)$ represents the relative distance between these aircrafts. The aircrafts are supposed to have a constant speed during at least a *lookahead time* and their velocities are also represented relatively in a two-dimensional space $v = (vx_1 - vx_2, vy_1 - vy_2)$.

Given a distance D, a loss of separation occurs when the aircraft are too close, this means that their distance is less than D:

$$loss?(s) \iff \sqrt{s_x^2 + s_y^2} < D$$

And a conflict occurs when a loss of separation is going to occur before the end of

```
cd2d  : THEORY
BEGIN

  IMPORTING reals@sqrt, Elim

  zero_vect2?(zerov : [real,real]) : bool =
     zerov`1 = 0 AND zerov`2 = 0

  det(sdet,vdet : [real,real]) : real =
     sdet`1 * vdet`2 - sdet`2 * vdet`1

  horizontal_los?(horizv : [real,real], horizD : real) : bool =
     horizv`1 * horizv`1 + horizv`2 * horizv`2 < horizD * horizD

  minmax(maxv1,maxv2, minv : real) : real =
     LET maxi = IF maxv1 > maxv2 THEN maxv1 ELSE maxv2 ENDIF IN
        IF maxi < minv THEN maxi ELSE minv ENDIF

  maxmin(minv1,minv2,maxv : real) : real =
     LET mini = IF minv1 < minv2 THEN minv1 ELSE minv2 ENDIF IN
        IF mini > maxv THEN mini ELSE maxv ENDIF

  Delta(sDelt,vDelt : [real,real], DDelt : real) : real =
     (DDelt * DDelt) * (vDelt`1 * vDelt`1 + vDelt`2 * vDelt`2) -
      det(sDelt,vDelt)*det(sDelt,vDelt)

  Theta_D(sThe,nzvThe : [real,real], eps, Dthe : real):real =
     LET a = (nzvThe`1 * nzvThe`1 + nzvThe`2 * nzvThe`2),
         b = sThe`1 * nzvThe`1 + sThe`2 * nzvThe`2,
         c = (sThe`1 * sThe`1 + sThe`2 * sThe`2) - Dthe * Dthe
     IN
        (-b + eps*sqrt((b*b) - a*c))/a ;

  dtct_2D(s,v : [real,real],B,T,D,Entry,Exit : real) : [real,real] =
     IF zero_vect2?(v) AND horizontal_los?(s,D)
     THEN
      (B,T)
     ELSIF Delta(s,v,D) > 0 THEN
      LET tin  = Theta_D(s,v,Entry,D),
          tout = Theta_D(s,v,Exit,D)
      IN
        (minmax(tin,B,T),maxmin(tout,T,B))
     ELSE
      (B,B)
     ENDIF

  detect?(st, vt : [real,real], Bt, Tt, Dt, Entryt, Exitt : real) : bool =
    LET (tint,toutt) = dtct_2D(st,vt,Bt,Tt,Dt,Entryt,Exitt) IN
      tint < toutt

END cd2d
```

Fig. 6. `cd2d` Conflict Detection Program

a lookahead time T:

$$conflict?(s,v) \Longleftrightarrow \exists\ t \leq T,\ loss?(s + t \cdot v)$$

Where $+$ and $\cdot$ are the usual addition and constant multiplication in $\mathbb{R}^2$. A function named `dtct_2D` is defined in `cd2d`, it computes the interval of time where the loss of separation occurs as described by the predicates in Figure 7 that have been proven in Pvs.

In order to be able to transform the original program we had to clean it from all the lemmas and theorems to only keep the computation part. The Pvs specification of the conflict detection algorithm as defined in the ACCoRD system is presented in Figure 6.

The only square roots and divisions of this program are in the `Theta_D` function, however in the body of the `dtct_2D` function, the result of `Theta_D` is then used by

```
dtct_2D_correct : THEOREM
  LET (tin,tout) = dtct_2D(s,v) IN
    tin < t AND t < tout IMPLIES horizontal_los?(s+t*v)

dtct_2D_complete : THEOREM FORALL (s,v)
  LET (tin,tout) = dtct_2D(s,v) IN
    horizontal_los?(s+t*v) IMPLIES
      tin <= t AND t <= tout AND tin < tout

conflict_dtct_2D : THEOREM FORALL (s,v)
  LET (tin,tout) = dtct_2D(s,v) IN
    conflict_2D?(s,v) IFF tin < tout
```

Fig. 7. cd2d correctness lemmas

```
cd2d_elim : THEORY
BEGIN
   IMPORTING cd2d, cd2d_operators, reals@sqrt, Elim

   zero_vect2?_e(zerov : [real,real]) :
      {res : bool | res = zero_vect2?(zerov)} =
        zerov`1 = 0 AND zerov`2 = 0

   det_e(sdet, vdet : [real,real]) :
      {det : real | det = det((sdet, vdet))} =
        sdet`1 * vdet`2 - sdet`2 * vdet`1

   horizontal_los?_e(horizv : [real,real], horizD : real) :
     {res : bool | res = horizontal_los?((horizv, horizD))} =
        horizv`1 * horizv`1 + horizv`2 * horizv`2 - horizD * horizD < 0

   minmax_e(maxv1_n_1, maxv1_n_2, maxv1_d, maxv2, minv, sq_4 : real) :
     {minmax_n_1, minmax_n_2, minmax_d, sq_6 : real |
       (minmax_n_1 + minmax_n_2 * sqrt(sq_6)) / minmax_d =
         minmax(((maxv1_n_1 + maxv1_n_2 * sqrt(sq_4)) / maxv1_d,
                 maxv2,
                 minv))} =
       LET (maxi_n_1, maxi_n_2, maxi_d, sq_5) =
            IF gt0_e((maxv1_n_1, maxv1_n_2, maxv1_d, maxv2, sq_4))
            THEN (maxv1_n_1, maxv1_n_2, maxv1_d, sq_4)
            ELSE (maxv2, 0, 1, 0)
            ENDIF
       IN
       IF lt1_e((maxi_n_1, maxi_n_2, maxi_d, minv, sq_5))
       THEN (maxi_n_1, maxi_n_2, maxi_d, sq_5)
       ELSE (minv, 0, 1, 0)
       ENDIF

   maxmin_e(minv1_n_1, minv1_n_2, minv1_d, minv2, maxv, sq_1 : real) :
     {maxmin_n_1, maxmin_n_2, maxmin_d, sq_3 : real |
        (maxmin_n_1 + maxmin_n_2 * sqrt(sq_3)) / maxmin_d =
           maxmin(((minv1_n_1 + minv1_n_2 * sqrt(sq_1)) / minv1_d,
                   minv2,
                   maxv))} =
       LET (mini_n_1, mini_n_2, mini_d, sq_2) =
            IF lt1_e((minv1_n_1, minv1_n_2, minv1_d, minv2, sq_1))
            THEN (minv1_n_1, minv1_n_2, minv1_d, sq_1)
            ELSE (minv2, 0, 1, 0)
            ENDIF
       IN
       IF gt0_e((mini_n_1, mini_n_2, mini_d, maxv, sq_2))
       THEN (mini_n_1, mini_n_2, mini_d, sq_2)
       ELSE (maxv, 0, 1, 0)
       ENDIF
```

Fig. 8. Transformed cd2d Pt. 1

the `minmax` and `maxmin` functions. Therefore, the results of square root and division operations would propagate to these other functions during the execution.

The OCaml implementation of the transformation of programs with function definitions with the subtype predicate generation outlined in Section 3 transforms the Pvs program from Figure 6 into the program in Figures 8 and 9. The compar-

```
    Delta_e(sDelt, vDelt : [real,real], DDelt : real) :
      {Delta : real | Delta = Delta((sDelt, vDelt, DDelt))} =
         DDelt * DDelt * (vDelt`1 * vDelt`1 + vDelt`2 * vDelt`2) -
           det_e((sDelt, vDelt)) * det_e((sDelt, vDelt))

    Theta_D_e(sThe, nzvThe : [real,real], eps, Dthe : real) :
      {Theta_D_n_1, Theta_D_n_2, Theta_D_d, sq_0 : real |
        (Theta_D_n_1 + Theta_D_n_2 * sqrt(sq_0)) / Theta_D_d =
           Theta_D((sThe, nzvThe, eps, Dthe))} =
        LET a =
              nzvThe`1 * nzvThe`1 + nzvThe`2 * nzvThe`2
        IN
        LET b =
              sThe`1 * nzvThe`1 + sThe`2 * nzvThe`2
        IN
        LET c =
              sThe`1 * sThe`1 + sThe`2 * sThe`2 - Dthe * Dthe
        IN (-b, eps, a, b * b - a * c)

    dtct_2D_e(s, v : [real,real], B, T, D, Entry, Exit : real) :
      {dtct_2D1_n_1, dtct_2D1_n_2,
        dtct_2D1_d, dtct_2D2_n_1,
         dtct_2D2_n_2, dtct_2D2_d, sq_7, sq_8 : real |
            ((dtct_2D1_n_1 + dtct_2D1_n_2 * sqrt(sq_8)) /
                dtct_2D1_d,
             (dtct_2D2_n_1 + dtct_2D2_n_2 * sqrt(sq_7)) /
                dtct_2D2_d) =
                   dtct_2D((s, v, B, T, D, Entry, Exit))} =
     IF zero_vect2?_e(v) AND horizontal_los?_e((s, D))
     THEN (B, 0, 1, T, 0, 1, 0, 0)
     ELSE
     IF Delta_e((s, v, D)) > 0
     THEN
     LET (Theta_D_n_1, Theta_D_n_2, Theta_D_d, sq_0) =
          Theta_D_e((s, v, Entry, D))
     IN
     LET (new_Theta_D_n_1, new_Theta_D_n_2, new_Theta_D_d, new_sq_0) =
          Theta_D_e((s, v, Exit, D))
     IN
     LET (maxmin_n_1, maxmin_n_2, maxmin_d, sq_3) =
          maxmin_e((new_Theta_D_n_1, new_Theta_D_n_2,
                    new_Theta_D_d, T, B, new_sq_0))
     IN
     LET (minmax_n_1, minmax_n_2, minmax_d, sq_6) =
          minmax_e((Theta_D_n_1, Theta_D_n_2, Theta_D_d, B, T, sq_0))
     IN (minmax_n_1, minmax_n_2, minmax_d, maxmin_n_1,
         maxmin_n_2, maxmin_d, sq_3, sq_6)
     ELSE (B, 0, 1, B, 0, 1, 0, 0)
     ENDIF
     ENDIF

    detect?_e(st, vt : [real,real], Bt, Tt, Dt, Entryt, Exitt : real) :
      {res : bool | res = detect?((st, vt, Bt, Tt, Dt, Entryt, Exitt))} =
    LET (dtct_2D1_n_1, dtct_2D1_n_2, dtct_2D1_d,
      dtct_2D2_n_1, dtct_2D2_n_2, dtct_2D2_d, sq_7, sq_8) =
              dtct_2D_e((st, vt, Bt, Tt, Dt, Entryt, Exitt))
    IN lt3_e((dtct_2D1_n_1, dtct_2D1_n_2, dtct_2D1_d,
         dtct_2D2_n_1, dtct_2D2_n_2, dtct_2D2_d, sq_8, sq_7))
END cd2d_elim
```

Fig. 9. Transformed `cd2d` Pt. 2

isons are replaced by functions such as `gt_0_e` or `lt_1_e` used in the `minmax` and `maxmin` function. Their use is factorized since both `minmax_e` and `maxmin_e` use the same comparison functions. These comparison functions are defined in a separate file, namely `cd2d_operators.pvs`, as introduced in Section 4.2.

As one can notice, the number of lines in the output program is more than twice the length of the input one. However this is mainly due to the length of the subtyping predicates associated to the transformed functions. All of these subtyping predicates can be proven by first unfolding the functions of the input program and then using the `(grind-reals)` strategy.

This transformed program is therefore equivalent to the input one according to the type predicates embedded in the type of the functions and it does not use square roots or divisions anymore except in these predicates. In particular, the last function, `detect?_e`, returning a Boolean value has not only the same signature but also the same behavior then the corresponding input function, namely `detect?`, but its result does not depend on any square root or division computation. Therefore, being able to construct an exact implementation of real numbers computations with addition, subtraction and multiplication would enable an exact execution of this program.

Conclusion

We have presented a program transformation that eliminates square roots and divisions from straight line programs in order to allow exact computation with addition and multiplication. This transformation embeds a certifying mechanism that is used to prove the semantics preservation between the definitions of the input and the output program. Eliminating square roots and divisions in a proof assistant also allows the use of some decision procedure that were not handling such operations in a first place.

Future work on this transformation includes the extension of the language this transformation applies to by including loops and more generally some kind of recursion that are not supported by the anti-unification and inlining approach. The transformation of PVS types could also be improved by transforming the PVS subtypes predicates of the input programs into the equivalent ones about the definitions of the output program.

Acknowledgment

I would like to thanks César Muñoz for helpful discussions on this project. This work was supported by the Assurance of Flight Critical System's project of NASA's Aviation Safety Program at Langley Research Center under Research Cooperative Agreement No. NNL09AA00A awarded to the National Institute of Aerospace.

References

[1] S. Boldo and J.-C. Filliâtre. Formal verification of floating-point programs. In P. Kornerup and J.-M. Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[2] C. Cohen. Construction of real algebraic numbers in Coq. In L. Beringer and A. Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, Princeton, États-Unis, Aug. 2012. Springer.

[3] M. Daumas, G. Melquiond, and C. Muñoz. Guaranteed proofs using interval arithmetic. In *IEEE Symposium on Computer Arithmetic*, pages 188–195, 2005.

[4] A. Goodloe, C. Muñoz, F. Kirchner, and L. Correnson. Verification of numerical programs: From real numbers to floating point numbers. In G. Brat, N. Rungta, and A. Venet, editors, *Proceedings of the 5th NASA Formal Methods Symposium (NFM 2013)*, volume 7871 of *Lecture Notes in Computer Science*, pages 441–446, Moffett Field, CA, May 2013.

[5] E. Goubault and S. Putot. Static analysis of finite precision computations. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 232–247, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] L. Lensink, S. Smetsers, and M. van Eekelen. Generating verifiable java code from verified PVS specifications. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, pages 310–325, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[8] C. Muñoz. Rapid prototyping in PVS. Contractor Report NASA/CR-2003-212418, NASA, Langley Research Center, Hampton VA 23681-2199, USA, May 2003.

[9] P. Neron. Elimination of square roots and divisions by partial inlining. To appear in PPDP 2014.

[10] 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 *Lecture Notes in Computer Science*, pages 256–272. Springer, 2012.

[11] P. Neron. *A Quest for Exactness: Program Transformation for Reliable Real Numbers*. PhD thesis, Ecole polytechnique, 2013.

[12] P. Neron. Square root and division elimination in PVS. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 457–462. Springer, 2013.

[13] R. O'Connor. Certified exact transcendental real number computation in Coq. In O. Mohamed, C. César, Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 246–261. Springer Berlin Heidelberg, 2008.

[14] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[15] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine intelligence*, 5(1):135–151, 1970.

[16] Z. Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

[17] J. Vuillemin. Exact real arithmetic with continued fractions. Rapport de recherche RR-760, INRIA, 1987.

[18] E. Wiedmer. Computing with Infinite Objects. *Theoretical Computer Science*, 10:133–155, 1980.