A Formal Proof of Square Root and Division Elimination in Embedded Programs

Pierre Neron

A Formal Proof of Square Root and Division Elimination in Embedded Programs

Pierre NERON
École polytechnique - INRIA

The use of real numbers in a program can introduce differences between the expected and the actual behavior of the program, due to the finite representation of these numbers. Therefore, one may want to define programs using real numbers such that this difference vanishes. This paper defines a program transformation for a certain class of programs that improves the accuracy of the computations on real number representations by removing the square root and division operations from the original program in order to enable exact computation with addition, multiplication and subtraction. This transformation is meant to be used on embedded systems, therefore the produced programs have to respect constraints relative to this kind of code. In order to ensure that the transformation is correct, i.e. preserves the semantics, we also aim at specifying and proving this transformation using the Pvs proof assistant.

1. INTRODUCTION

Safety critical embedded systems, for instance in aeronautics, demand a very high level of reliability since any failure can have critical consequences. One approach to ensure such levels of safety is the use of proof assistants such as Pvs, Coq, Hol, to prove properties on these programs, ensuring that the program satisfies its specification. Some of these systems use computation over real numbers, this is the case of the conflict detection and resolution algorithms introduced in [NMD12, MBMD09] that have an extended use of solid geometry. However computing with real numbers is unsafe since concrete implementations of real numbers do not always behave as one would expect.

One of the main challenges is that real numbers can not be represented in an exact way in programs and we have to use different representations in order to make computation, *e.g.,* the floating point numbers defined by the IEEE754-Standard [IEE85] with a fixed number of bits. This kind of representation always introduces differences between the *expected* behavior (defined by the abstract semantics), where we assume that numbers are genuine real numbers, and the *actual* behavior (defined by a concrete semantics), happening when we run the program. Furthermore some programs, *e.g.,* air traffic management systems [NMD10], use tests on comparisons between real numbers and we may want to ensure some numerical stability on the programs that use these features since a tiny error in a test can make the actual behavior greatly diverge from the expected one.

Using formalizations of the floating point semantics [BM06, BF07, Har95, Min95] makes proofs of programs used in aeronautics difficult since most of the properties of real numbers and operations (*e.g.,* associativity) do not hold on such representations. Therefore many proofs are done on the abstract semantics which does not represent the actual behavior of the program. Differences between the abstract and the floating point semantics have been studied in a particularly efficient way

using static analysis by abstract interpretation with numerical abstract domains [CMC08] and interval arithmetic in order to provide numerical analysis of programs [DMM05]. These methods have been used to define a program transformation to improve accuracy [Mar09] but our goal is slightly different.

We aim at producing, for any program that computes a boolean with real arithmetic using $+, -, \times, /, \sqrt{}$, an equivalent program whose concrete semantics is equal to its abstract semantics on genuine real numbers. By introducing dynamic representations of real numbers, techniques have been developed to compute exactly. In 1980, Wiedmer studied the computation over infinite objects [Wie80] and introduced a representation with infinite decimal fraction. Then Boehm and Cartwright [BCRO86] both extended this representation as a sequence of fractions and introduced a representation using lazy evaluation of the digits representing the real number. Different constructions of real numbers have since been introduced, with redundant representation of continued fractions [Vui87] or with functional representation and lazy evaluation [Sim98, DGL04]. Some representations have even been formalized in the Coq system, a constructive construction of the real number field is presented in [O'C08, KS11] and a construction of the algebraic numbers [Bos03] is formalized in [Coh12]. However, in embedded systems we favor programs that can be executed in a fixed size memory, this prevents us from using these dynamic representations of real numbers since computation over it requires an unbounded amount of memory.

Exact computation over real numbers can also be done on addition and multiplication by using a fixed point representation with dynamic size since we are able to predict the sizes required by such computations using static analysis, the language we target being free of loops or recursion. Thus, we aim at defining a program transformation that removes square root and division operations in every boolean expression of the program, *e.g.,* transforms $\sqrt{x} > y$ into $y < 0 \vee x > y^2$, in order to use this exact computation with only addition and multiplication. Exactly computing the boolean values protects the control flow of the program from rounding errors.

As our goal is to improve the safety of the systems we target, we want to formally prove in Pvs that this transformation preserves the semantics. Therefore, by computing exactly with addition and multiplication, we can ensure that the path taken in each test is the same in both abstract and concrete semantics and that formal proofs of properties about boolean values done on the abstract semantics still hold on the concrete one. Indeed, if proofs on the original program using concrete semantics are difficult, so are proofs on the output of program transformations due to the size and the complexity of the produced code, this is the reason why the correction of the transformation needs to be formally proven.

The paper is an extended version of [Ner12],x it is organized as follows. First, we define the language on which the transformation applies and some general features about the Pvs formalization. Then we define the main transformation and two auxiliary methods that respectively remove square roots and divisions from boolean expression and from variable definitions. Finally, we present some experimental results using the OCaml implementation of the transformation which is almost an executable copy of Pvs formalization with few hand made extra features (Pvs and OCaml files are available on the author's web page).

2. PRESENTATION OF THE LANGUAGE

2.1 Language definition

In this section we define the syntax of the language the transformation applies to. This language, that embeds the core functionalities of our targeted programs, is a typed functional language that contains numerical ($\mathbb{R}$) and boolean constants, variable definitions (as let in instructions), tests (if then else), pairs and the usual arithmetic $+$, $-$, $\times$ (we also use $\cdot$ instead of $\times$), $/$, $\sqrt{}$, the comparisons $=$, $\neq$, $>$, $\geq$, $<$, $\leq$ and boolean operators ($\wedge$, $\vee$, $\neg$). Therefore we can define the syntax of our language as follows:

DEFINITION 2.1.1. **Syntax of the language**

Prog :=	Constant	| fst Prog	| Var
	| uop Prog	| snd Prog	| let Var = Prog in Prog
	| Prog op Prog	| (Prog, Prog)	| if Prog then Prog else Prog

where:

$$\begin{aligned} \text{Constant} &\subset \mathbb{R} \cup \{True,\ False\} \\ \text{op} &\in \{+, \times, /,\ =,\ \neq,\ >,\ \geq,\ <,\ \leq,\ \wedge,\ \vee\} \\ \text{uop} &\in \{\sqrt{},\ -,\ \neg\} \end{aligned}$$

In order to complete the description of the language, we define the type of a program, in a typing environment Γ that associates to every free variable its type.

DEFINITION 2.1.2. **Type system**

$$Type \ := \ \mathbb{R} \mid \mathbb{B} \mid Type \times Type$$

The rules are the usual ones for a functional language:

$$\frac{}{\Gamma,\mathsf{x}{:}A \vdash \mathsf{x}{:}A}$$

$$\frac{\Gamma \vdash e{:}\mathbb{B}}{\Gamma \vdash \neg e{:}\mathbb{B}}$$

$$\frac{\Gamma \vdash e{:}\mathbb{R}}{\Gamma \vdash uop\ e{:}\mathbb{R}} \qquad uop \in \{-,\ \sqrt{}\}$$

$$\frac{\Gamma \vdash e_1{:}\mathbb{R} \quad \Gamma \vdash e_2{:}\mathbb{R}}{\Gamma \vdash e_1\ op\ e_2{:}\mathbb{R}} \qquad op \in \{+, \times, /\}$$

$$\frac{\Gamma \vdash e_1{:}\mathbb{R} \quad \Gamma \vdash e_2{:}\mathbb{R}}{\Gamma \vdash e_1\ op\ e_2{:}\mathbb{B}} \qquad op \in \{=, \neq, >, <, \geq, \leq\}$$

$$\frac{\Gamma \vdash e_1{:}\mathbb{B} \quad \Gamma \vdash e_2{:}\mathbb{B}}{\Gamma \vdash e_1\ op\ e_2{:}\mathbb{B}} \qquad op \in \{\vee, \wedge\}$$

$$\frac{\Gamma \vdash e{:}T_1 \times T_2}{\Gamma \vdash \mathsf{fst}(e){:}T_1}$$

$$\frac{\Gamma \vdash e{:}T_1 \times T_2}{\Gamma \vdash \mathsf{snd}(e){:}T_2}$$

$$\frac{\Gamma \vdash e_1{:}T_1 \quad \Gamma \vdash e_2{:}T_2}{\Gamma \vdash (e_1,\ e_2){:}T_1 \times T_2}$$

$$\frac{\Gamma \vdash e_1{:}T_1 \quad \Gamma,\mathsf{x}{:}T_1 \vdash e_2{:}T_2}{\Gamma \vdash \mathsf{let\ x} = e_1\ \mathsf{in}\ e_2{:}T_2}$$

$$\frac{\Gamma \vdash f{:}\mathbb{B} \quad \Gamma \vdash e_1{:}T \quad \Gamma \vdash e_2{:}T}{\Gamma \vdash \mathsf{if}\ f\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2{:}T}$$

These types are used to identify the way a program has to be transformed. Indeed, the transformation is different for pure numerical expressions (*e.g.,* in a variable definition) and for the ones used in boolean expressions (*i.e.,* as arguments of a comparison). It is easy to define a type checking function in PVS, `type_infer(p,Γ)`,

that returns either a type or an undefined value ($\mathbb{U}$) if the program has no valid type in the given environment. $Ty_\Gamma(\mathsf{p})$ now denotes the result of `type_infer(Γ)(p)`.

Notation. In this paper we use the `teletype font` to represent PVS expressions. For clarity and conciseness, every element of Prog and subtypes of Prog will be written using the sans serif font in the concrete syntax (instead of their PVS abstract syntax,*e.g.,* we will write `e1 op e2` for `bop(op,e1,e2)`), some PVS expressions will be abstracted by their equivalent *mathematic expressions* (*e.g.,* $\forall\Gamma,\ Ty_\Gamma(\mathsf{p}) \neq \mathbb{U}$ stands for `FORALL tenv, type_infer(p,tenv) /= Undefined`) and we will not give the type when there is no ambiguity (Γ is always a typing environment *i.e.,* a function from Var to $Type$).

Next, we define the denotational semantics of a program in the language, using an environment Env that associates to every variable its value. It is the usual semantics of a functional language. The $Fail$ value corresponds to the square roots of negative numbers, divisions by zero and unsound types cases.

DEFINITION 2.1.3. **Denotational semantics of the language**
The semantics is defined by the following induction rules:

$$\frac{}{Env \vdash [\![\mathsf{c}]\!]=\mathsf{c}} \qquad \mathsf{c} \in \mathsf{Constant}$$

$$\frac{}{Env,(\mathsf{x},e) \vdash [\![\mathsf{x}]\!]=e}$$

$$\frac{Env \vdash [\![\mathrm{E}]\!]=Fail}{Env \vdash [\![\mathrm{uop\ E}]\!]=Fail}$$

$$\frac{Env \vdash [\![\mathrm{E}]\!]=e}{Env \vdash [\![\mathrm{uop\ E}]\!]=uop(e)} \qquad \mathrm{uop} \in \{-,\neg\} \wedge e \neq Fail$$

$$\frac{Env \vdash [\![\mathrm{E}]\!]=e}{Env \vdash [\![\sqrt{\mathsf{E}}]\!]=Fail} \qquad e < 0 \vee e = Fail$$

$$\frac{Env \vdash [\![\mathrm{E}]\!]=e}{Env \vdash [\![\sqrt{\mathsf{E}}]\!]=\sqrt{e}} \qquad e \geq 0$$

$$\frac{Env \vdash [\![\mathrm{E}_1]\!]=e_1 \quad Env \vdash [\![\mathrm{E}_2]\!]=e_2}{Env \vdash [\![\mathrm{E}_1\ \mathrm{op}\ \mathrm{E}_2]\!]=e_1\ op\ e_2} \qquad e_1 \neq Fail \wedge e_2 \neq Fail \wedge (\mathrm{op} = / \Rightarrow e_2 \neq 0)$$

$$\frac{Env \vdash [\![\mathrm{E}_1]\!]=e_1 \quad Env \vdash [\![\mathrm{E}_2]\!]=e_2}{Env \vdash [\![\mathrm{E}_1\ \mathrm{op}\ \mathrm{E}_2]\!]=Fail} \qquad e_1 = Fail \vee e_2 = Fail \vee (\mathrm{op} = / \wedge e_2 = 0)$$

$$\frac{Env \vdash [\![\mathrm{E}_1]\!]=e_1 \quad Env \vdash [\![\mathrm{E}_2]\!]=e_2}{Env \vdash [\![(\mathrm{E}_1,\mathrm{E}_2)]\!]=Fail} \qquad e_1 = Fail \vee e_2 = Fail$$

$$\frac{Env \vdash [\![\mathrm{E}_1]\!]=e_1 \quad Env \vdash [\![\mathrm{E}_2]\!]=e_2}{Env \vdash [\![(\mathrm{E}_1,\mathrm{E}_2)]\!]=(e_1,e_2)} \qquad e_1 \neq Fail \wedge e_2 \neq Fail$$

$$\frac{Env \vdash [\![\mathrm{E}]\!]=(e_1,e_2)}{Env \vdash [\![\mathsf{fst}(\mathrm{E})]\!]=e_1} \qquad e_1 \neq Fail \wedge e_2 \neq Fail$$

$$\frac{Env \vdash [\![\mathrm{E}]\!]=(e_1,e_2)}{Env \vdash [\![\mathsf{fst}(\mathrm{E})]\!]=Fail} \qquad e_1 = Fail \vee e_2 = Fail$$

$$\frac{Env \vdash [\![\mathrm{E}]\!]=(e_1,e_2)}{Env \vdash [\![\mathsf{snd}(\mathrm{E})]\!]=e_2} \qquad e_1 \neq Fail \wedge e_2 \neq Fail$$

$$\frac{Env \vdash [\![\mathrm{E}]\!]=(e_1,e_2)}{Env \vdash [\![\mathsf{snd}(\mathrm{E})]\!]=Fail} \qquad e_1 = Fail \vee e_2 = Fail$$

$$\frac{Env \vdash [\![\mathrm{E}_1]\!]=e_1 \quad Env,(\mathsf{x},e_1) \vdash [\![\mathrm{E}_2]\!]=e_2}{Env \vdash [\![\mathsf{let\ x = E_1\ in\ E_2}]\!]=e_2} \qquad e_1 \neq Fail \wedge e_2 \neq Fail$$

$$\frac{Env \vdash [\![\mathrm{E}_1]\!]=e_1 \quad Env,(\mathsf{x},e_1) \vdash [\![\mathrm{E}_2]\!]=e_2}{Env \vdash [\![\mathsf{let\ x = E_1\ in\ E_2}]\!]=Fail} \qquad e_1 = Fail \vee e_2 = Fail$$

$$\frac{Env \vdash [\![\mathrm{F}]\!]=True \qquad Env \vdash [\![\mathrm{E}_1]\!]=e_1}{Env \vdash [\![\textsf{if F then E}_1 \textsf{ else E}_2]\!]=e_1}$$
$$\frac{Env \vdash [\![\mathrm{F}]\!]=False \qquad Env \vdash [\![\mathrm{E}_2]\!]=e_2}{Env \vdash [\![\textsf{if F then E}_1 \textsf{ else E}_2]\!]=e_2}$$
$$\frac{Env \vdash [\![\mathrm{F}]\!]=Fail}{Env \vdash [\![\textsf{if F then E}_1 \textsf{ else E}_2]\!]=Fail}$$

We now denote $[\![\mathsf{p}]\!]_{Env}$ the abstract semantics of p into an environment *Env* (*i.e.,* the v such that $Env \vdash [\![\mathsf{p}]\!] = v$, in PVS the corresponding function is called `semantics`). The language being defined, we now describe some subtypes of Prog, that represent restricted syntactic forms.

2.2 Program subtypes

2.2.1 *Normalized language.* The normalized language is the subtype on which our transformation applies, it is a subtype of the Prog type that has the following definition:

DEFINITION 2.2.1. **Expressions and programs normal form**
The unary expressions E_u *are built with operators and the expressions* E *with pairs.*

$$\begin{array}{rcl} \mathsf{E}_u & := & \mathsf{Var} \mid \mathsf{Constant} \mid \mathsf{uop}\ \mathsf{E}_u \mid \mathsf{E}_u\ \mathsf{op}\ \mathsf{E}_u \mid \mathsf{fst}\ \mathsf{E}_u \mid \mathsf{snd}\ \mathsf{E}_u \\ \mathsf{E} & := & (\mathsf{E}, \mathsf{E}) \mid \mathsf{E}_u \end{array}$$

The programs P *can also contain variable definitions and tests:*

$$\mathsf{P} := \mathsf{let\ Var} = \mathsf{P\ in\ P} \mid \mathsf{if\ P\ then\ P\ else\ P} \mid \mathsf{E}$$

Hence, every time we meet an arithmetic or boolean operator, we know there are neither definitions nor tests inside its arguments and fst and snd constructs can only be applied to variables (in well typed programs) since the pair constructs have been removed at top level in their arguments. Therefore they can not contain any square roots or divisions as subexpressions. In Section 3.5 the rules allowing the transformation of any program in Prog into a program in P is introduced.

2.2.2 *Target language.* Then we define the language that corresponds to programs from which divisions and square roots have been eliminated. Certainly, we can not eliminate all square roots and divisions from any program, *e.g.,* the execution of the program $\sqrt{2}$ will still return a rounded value of $\sqrt{2}$, but we are able to remove them from all the boolean values computations, particularly the boolean values of the tests. Therefore, it allows us to protect the control flow of the program from rounding errors introduced by these operations. We define new subtypes of expressions and programs, depending on which operators are allowed in different parts of the program.

DEFINITION 2.2.2. **Target language** *The Figure 1 describes the different set of operators, expressions and programs that are used to define the target language.*

—$\mathsf{N}_{\sqrt{},/}$ *is the set of numerical expression with square roots and divisions*
—N *is the set of square root and division free numerical expressions*
—*in* B_{let}*, we allow variable definitions of square root and division free boolean expressions in order to reduce the size of the output program as we will see in Section 5.*

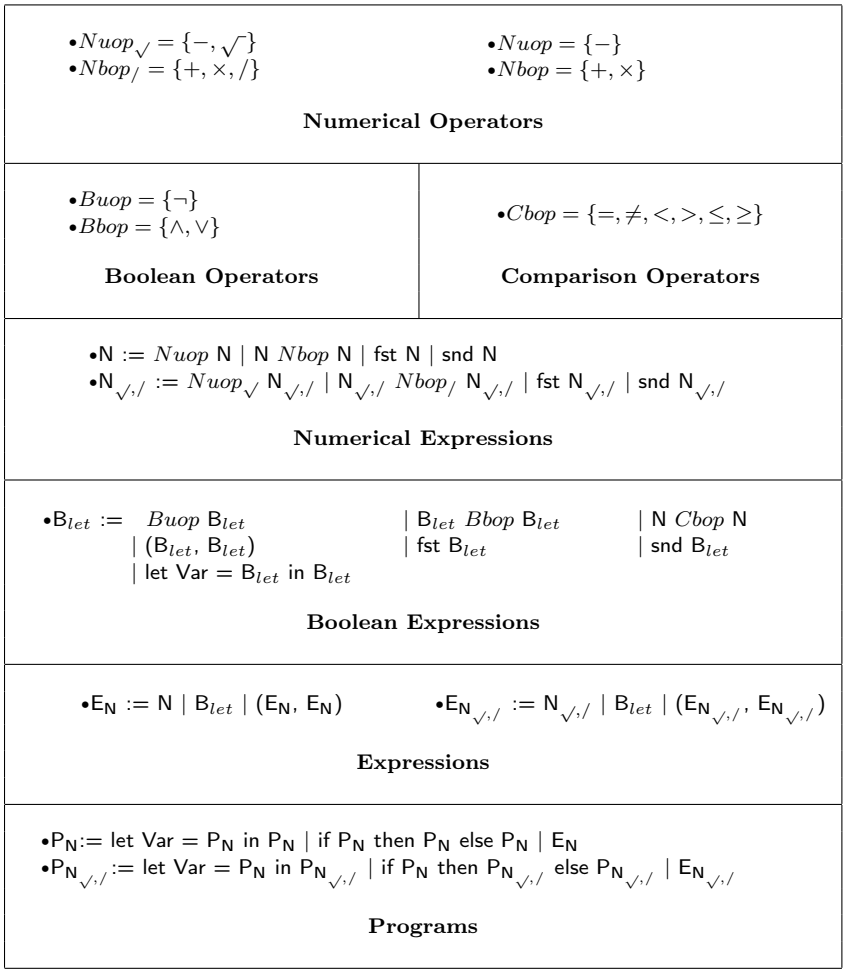

Fig. 1. Target language definitions

—$\mathsf{E_N}$ *is the set of expressions where square roots and divisions are not allowed*

—$\mathsf{E_{N_{\surd,/}}}$ *is the set of expressions where square roots and divisions are only allowed in numerical expressions that are not sub-terms of boolean expressions*

—$\mathsf{P_N}$ *is the set of programs that do not contain any square root or division*

—$\mathsf{P_{N_{\surd,/}}}$ *is the set of programs that can contain square roots or divisions only in the final numerical expressions (not in the body of any variable definition or any test).*

The language $\mathsf{P_{N_{\surd,/}}}$ *is the language the main transformation targets. For all programs in this language, the boolean sub-expressions and the variables they might depend on are free of square roots and divisions, they can be exactly computed with* $+, \times, -$*. Thus the control flow of such a program is protected from any rounding error.*

For example $(\sqrt{x}, a > b)$ is in $\mathsf{E}_{\mathsf{N}_{\sqrt{},/}}$ but not in E_{N} but $\sqrt{a} > b$ is in none of them. One can notice that any program in the targeted subtype that returns a boolean value is completely square root and division free.

PROPOSITION 2.2.1. **Output Boolean program** *A program in* $\mathsf{P}_{\mathsf{N}_{\sqrt{},/}}$ *returning a boolean value is in* P_{N}*:*

$$\forall\ \mathsf{p} \in \mathsf{E}_{\mathsf{N}_{\sqrt{},/}},\ \exists \Gamma,\ Ty_{\Gamma}(\mathsf{p}) = \mathbb{B}^n \Longrightarrow \mathsf{p} \in \mathsf{P}_{\mathsf{N}}$$

These definitions allow us to characterize the set of programs transformed by each step of our transformation and what kind of programs it produces.

3. PRELIMINARIES ON THE PROGRAM TRANSFORMATION

3.1 About the PVS Proof Assistant

3.1.1 PVS *sub-typing.* The specification of an algorithm and the related proofs in PVS can be expressed using the PVS predicate sub-typing. Given a type `T` and a predicate `P` of type `T -> 𝔹`, {`x: T | P(x)`} is the subtype of `T` of all elements `x` of type `T` that satisfy `P`, this type can also be denoted `(P)`. Then every definition of a function in PVS can be specified using these subtypes, *e.g.,*

```
f(x: (P)): {x': T' | P'(x')}
```

defines a partial function on `T` that takes only elements `x` of type `T` that satisfy `P` and returns elements of type `T'` that satisfy `P'`. When PVS typechecks such a function, it generates Type Check Conditions (TCC) where we have to prove that:

—`f` can be applied to every element of type `(P)` (*Completeness*)
—∀ `x: T, f(x): T' ∧ P'(f(x))` (*Soundness*)
—if `f` is recursive, then for every recursive call on `e`:
- `e : (P)` (*Recursive call correctness*)
- a measure decreases, according to a well founded order provided in the definition of `f` (*Termination*)

The type of a function can also be restricted using the `HAS_TYPE` judgement, *e.g.,* given two types `T` and `T'`, two subtypes `S` of `T` and `S'` of `T'` and a function:

```
f (x: T): T'
```

then we can state the following judgement:

```
f (x: S) HAS_TYPE S'
```

and once the corresponding TCC are proven, use either `T'` or `S'` as type of `f(x)` when `x` has type `S`. Therefore, in this paper, we will not give correctness lemmas of the functions we define, these lemmas will be encoded in the type of the function using the predicate sub-typing of PVS.

3.1.2 *Orders on abstract datatypes.* PVS requests us to prove the termination of every recursive function by giving a measure that decreases according to a well founded order in every recursive call. When using PVS abstract datatypes, most of the time the order used is the subterm relation (denoted $\ll$) but this measure is not always sufficient. Therefore, in order to prove the termination of some functions we had to define some lexicographical orders. To this purpose, we defined the following functions :

- `lexical_order_nat_based(m: [T -> ℕ], o1: (wf?[T])): (wf?[T])`
- `lexical_order(o1: (wf?[T]), o2: (wf?[S])): (wf?[[T,S]])`

where $wf?$`[T]` is the Pvs predicate `well_founded[T]` and

```
- lexical_order_nat_based(m,o)(x,y) =
            m(x) < m(y) ∨ (m(x) = m(y) ∧ o(x,y))
- lexical_order(o1,o2)((x1,x2),(y1,y2)) =
            o1(x1,y1) ∨ (x1 = y1 ∧ o2(x2,y2))
```

These orders will be used to extend the subterm relation in some particular cases.

3.2 No Fail Assumption

The transformation we define in this paper targets critical embedded systems, these programs are proved to be type safe and one can also prove that failure due to divisions by zero or square roots of negative number do not occur. Therefore we assume that the programs we want to transform are well typed (there exists a type environment that allows us to type the program) and we prove the type and semantics preservation only for every environments where the initial program is well typed and its semantics does not fail. Hence we will not have to force the failure cases that disappear when removing divisions and square roots, *e.g.,* we can transform $1/x > 0$ into $x > 0$ instead of if $x = 0$ then Fail else $x > 0$ even if in an environment where x is evaluated to 0 the program $1/x > 0$ fails whereas $x > 0$ returns false. The type and semantics preservations are defined in the language of Pvs by the predicate `sem_ty_nf_eq(p)(p')`, that is:

$$(\forall \Gamma,\ Ty_\Gamma(\mathsf{p}) \neq \mathbb{U} \implies Ty_\Gamma(\mathsf{p'}) = Ty_\Gamma(\mathsf{p})) \wedge$$
$$(\forall Env, [\![\ \mathsf{p}\ ]\!]_{Env} \neq Fail \implies [\![\ \mathsf{p'}\ ]\!]_{Env} = [\![\ \mathsf{p}\ ]\!]_{Env})$$

We consider that programs p and p' are *equivalent* when `sem_ty_nf_eq(p)(p')` holds. Given this type and semantics preservation predicate, we can now formally define our main goal which is to present a transformation `Elim` that has the following specification:

> Definition 3.2.1. `Elim` **specification**
> `Elim(Γ)(p: Prog | Ty_Γ(p) ≠ 𝕌) : { p': P_N_√,/ | sem_ty_nf_eq(p)(p')}`

The `Elim` function is defined by a sequence of elementary transformations, all of them preserving the semantics when the program does not fail. Every elementary transformation is formalized using a function whose output type is of the form:

`{ p': P' | sem_ty_nf_eq(p)(p')}` where P' is a subtype of Prog

3.3 Well typed program

In the previous definition, the typing environment Γ which is an argument of the function and proves the program is well typed, is needed to ensure some syntactic structure (*e.g.,* fst(a+b)) never happens. It allows us to prove type checking conditions when we define some partial transformations and to prove the transformed program is in $\mathsf{P}_{\mathsf{N}_{\surd,/}}$ but the transformed program does not depend on it.
Most of the time the predicate $wtp(\mathsf{p})$ (`well_typed_program` $= \exists \Gamma,\ Ty_\Gamma(\mathsf{p}) \neq \mathbb{U}$) would be sufficient but we need to have the environment explicit when we want to prove, for example, that $Ty_\Gamma(\mathsf{e}) \neq \mathbb{U} \wedge Ty_\Gamma(\mathsf{e'}) \neq \mathbb{U} \implies Ty_\Gamma((\mathsf{e},\mathsf{e'})) \neq \mathbb{U}$

3.4 Substitution

We now describe a function that defines a capture avoiding substitution of every free occurrence of a variable x in a program p by a given program e (also denoted p[x:=e]). The semantics and the types of p are preserved in every environment where e is well typed and does not fail. That gives the following specification:

`replace(x: Var,e,p: Prog):` $\{$p': Prog $\mid \forall \Gamma, Env,$

$[\![$ p' $]\!]_{Env} = [\![$ p $]\!]_{Env;(x,[\![e]\!]_{Env})} \ \wedge \ Ty_\Gamma(p') = Ty_{\Gamma,(x,Ty_\Gamma(e))}(p)\}$

This substitution renames bound variables in p that are free in e. If Pvs usual termination proofs on inductive structures relies on the well founded subterm relation, the variable renaming and substitution prevent us from using this order. Therefore, termination is ensured by the lexicographic order:

`lexical_order_nat_based(let_number,`$\ll$`)`

`let_number(p)` being the number of variable definitions in p. The `replace` function also has several types that ensure the preservation of program subtypes when we have some hypothesis on e, *e.g.,*

`replace(x: Var, e:` E_u`, p: P) HAS_TYPE P`

Correctness of the substitution is proven using the two following lemmas that we will use every time we have to rename variables. We denote $FV(p)$ the set of the free variables of p, then:

- `no_FV_no_change: LEMMA =`
  $\forall$ x, p, x $\notin FV(p) \Longrightarrow \forall$ Env, v, $[\![p]\!]_{Env} = [\![p]\!]_{Env;(x,v)}$
- `no_FV_no_change_type: LEMMA =`
  $\forall$ x, p, x $\notin FV(p) \Longrightarrow \forall$ Γ, S, $Ty_\Gamma(p) = Ty_{(\Gamma;(x,S))}(p)$

3.5 Program normalization

The core of the transformation operates on programs in P, however since we aim at transforming every program in Prog we first present a transformation that transforms every program in Prog into an equivalent program of type P. This transformation can be done using the following set of reduction rules that takes the tests and variable definitions out of the binary operators and pairs and reduces the projections:

—inversions between variable definitions and other kinds of expressions:

- uop (let x = e1 in e2) $\longrightarrow$ let x = e1 in (uop e2)
- (let x = e1 in e2) op e3 $\longrightarrow$
  let x' = e1 in (e2[x:=x'] op e3) x' $\notin FV$((e2,e3))
- e1 op (let x = e2 in e3) $\longrightarrow$
  let x' = e2 in (e1 op e3[x:=x']) x' $\notin FV$((e1,e3))
- ((let x = e1 in e2),e3) $\longrightarrow$
  let x' = e1 in (e2[x:=x'],e3) x' $\notin FV$((e2,e3))
- (e1,let x = e2 in e3) $\longrightarrow$
  let x' = e2 in (e1,e3[x:=x']) x' $\notin FV$((e1,e3))
- fst (let x = e1 in e2) $\longrightarrow$ let x = e1 in fst (e2)
- snd (let x = e1 in e2)) $\longrightarrow$ let x = e1 in snd (e2))

—inversions between tests and other kinds of expressions:
- uop (if f then e1 else e2) ⟶
  if f then (uop e1) else (uop e2)
- (if f then e1 else e2) op e3 ⟶
  let xi = if f then e1 else e2 in (xi op e3) xi $\notin FV$(e3)
- e1 op (if f then e2 else e3) ⟶
  let xi = if f then e2 else e3 in (e1 op xi) xi $\notin FV$(e1)
- ((if f then e1 else e2),e3) ⟶
  let xi = if f then e1 else e2 in (xi,e3) xi $\notin FV$(e3)
- (e1,(if f then e2 else e3)) ⟶
  let xi = if f then e2 else e3 in (e1,xi) xi $\notin FV$(e1)
- fst (if f then e1 else e2) ⟶ if f then fst (e1) else fst (e2)
- snd (if f then e1 else e2) ⟶ if f then snd (e1) else snd (e2)

—projections reductions:
- fst (e1,e2) ⟶ e1
- snd (e1,e2) ⟶ e2

The transformation rules for a test inside a binary operator or a pair use a variable definition in order to avoid duplicating the other arguments, a process that can strongly increase the size of the transformed code.

Pvs *specification.* Creating new variable definitions makes the direct termination proof more complicated. Therefore we first define the following set of functions that normalize the application of operators, pair or projections to a program in normal form:

```
- uop_P_switch(uop)(p:P | wtp(p)): {p':P | sem_ty_nf_eq(uop p)(p')}
- fst_P_switch(p:P | wtp(p)): {p':P | sem_ty_nf_eq(fst(p))(p')}
- snd_P_switch(p:P | wtp(p)): {p':P | sem_ty_nf_eq(snd(p))(p')}
- op_P_switch(xi)(op)(p1,p2:P | wtp(p1 op p2)):
                              {p':P | sem_ty_nf_eq(p1 op p2)(p')}
- pair_P_switch(xi)(p1, p2:P | wtp((p1,p2))) :
                              {p':P | sem_ty_nf_eq((p1,p2))(p')}
```

The unary operator switches terminate using the usual subterm order. Binary operator switches involving variable renaming, the termination proof of the last two functions is done using the depth of the program. The correctness is proven by using the `no_FV_no_change` lemmas introduced in Section 3.4. Using these definitions we can define the main function that transforms any program into an equivalent normalized program by iterating these functions:

```
p_norm_reduction(xi:Var)(p:(wtp)): {pp:P | sem_ty_nf_eq(p)(pp)} =
CASES p OF
      uop e1: uop_P_switch(uop)(p_norm_reduction(xi)(e1))...
```

Since we can transform every well typed program in Prog into an equivalent one in P, we now describe the major function `Elim` of the transformation that transforms any well typed program in P into an equivalent one in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$.

4. NORMALIZED PROGRAM TRANSFORMATION

This `Elim` function is a recursive program transformation algorithm eliminating square roots and divisions. It uses two functions, `Elim_bool` and `Elim_let`, that

have the following specifications:

DEFINITION 4.1. `Elim_bool` **and** `Elim_let` **specifications**
`Elim_bool` *transforms any comparison (the* `comparison_expression` *subtype will be formally defined in Section 5) into an equivalent boolean program without any square root or division and* `Elim_let` *transforms a variable definition into an equivalent one that contains neither square root nor division in its body. Formally these functions have the following definitions:*

```
Elim_bool(Γ)(xsq: Var)(c: comparison_expression(Γ)) :
                                    { e: B_let | sem_ty_nf_eq(c)(e) }
Elim_let(Γ)(x: Var, p1: P_N√,/, p2: P | Ty_Γ(let x = p1 in p2) ≠ 𝕌):
      { x': Var, p1': P_N, p2': P |
                  sem_ty_nf_eq(let x = p1 in p2)(let x' = p1' in p2') &
                  if_letin_number(p2') <= if_letin_number(p2) }
```

`xsq` *is a variable used to name some boolean expressions in order to avoid formula duplications.*

REMARK 1. *Given an expression in* E*, then by applying the* `Elim_bool` *function to every comparison we can find in this expression, we can define the* `Elim_bool` *function that transforms every well typed expression in* E *into an equivalent one in* $\mathsf{E}_{\mathsf{N}_{\surd,/}}$ *that does not contain any square root or division.*

The function `if_letin_number(p)` gives the number of variable definitions and tests which occur in p, it will be used to prove the termination of the main algorithm. The `Elim_bool` function will be described in Section 5, and the `Elim_let` one in Section 6. Therefore using these two functions, we can define the recursive algorithm that transforms any program in P in an equivalent one in $\mathsf{P}_{\mathsf{N}_{\surd,/}}$:

DEFINITION 4.2. **The** `Elim` **function**
The transformation is a recursive algorithm on p*:*

```
Elim(Γ, xsq)(p:P | Ty_Γ(p) ≠ Fail): {p':P_N√,/ | sem_ty_nf_eq(p)(p')}=
     CASES p OF
       let x = p1 in p2:
         LET pn1 = Elim(Γ,xsq)(p1) IN
         LET (x',p1',p2') = Elim_let(Γ)(x,pn1,p2) IN
                  let x' = p1' in Elim(Γ,xsq)(x', Ty_Γ(p1'))))(xsq)(p2'),
       if f then p1 else p2:
         if Elim(Γ,xsq)(f) then Elim(Γ,xsq)(p1) else Elim(Γ,xsq)(p2)
     ELSE Elim_bool_expr(Γ)(xsq)(p)
     ENDCASES
     MEASURE (if_letin_number(p)) BY <
```

`elim_bool_expr` applies `elim_bool` on all the subterms of an expression that are comparison expressions. According to the typing predicate of the `Elim_let` function, the termination is ensured by proving that the number of variable definitions and tests strictly decreases. The correction is ensured by both `Elim_bool` and `Elim_let` typing predicates. Therefore, applying the `main_elim` function, which is the composition of both `p_norm_reduction` and `Elim` functions to any program in Prog, we can state that every program in Prog has an equivalent one in $\mathsf{P}_{\mathsf{N}_{\surd,/}}$:

THEOREM 1. Prog **is equivalent to** $\mathsf{P}_{\mathsf{N}_{\surd,/}}$

$\forall\ \Gamma,\ \mathsf{xsq},\ \mathsf{ifname},$

$\mathtt{main\_elim}(\Gamma)(\mathsf{xsq},\mathsf{ifname})(\mathsf{p}\colon\ \mathsf{Prog}\ \mid\ Ty_\Gamma(\mathsf{p}) \neq Fail)\colon$
$\{\mathsf{p'}\colon\ \mathsf{P}_{\mathsf{N}_{\surd,/}}\ \mid\ \mathtt{sem\_ty\_nf\_eq}(\mathsf{p})(\mathsf{p'})\}$

And its corollary, using Proposition 2.2.1, stating that every boolean program (whose type is in $\mathbb{B}^n$) has an equivalent division and square root division free program:

COROLLARY 1. **Elimination in boolean programs**

$\forall\ \Gamma,\ \mathsf{xsq},\ \mathsf{ifname},$

$\mathtt{main\_elim}(\Gamma)(\mathsf{xsq},\mathsf{ifname})(\mathsf{p}\colon\ \mathsf{Prog}\ \mid\ Ty_\Gamma(\mathsf{p}) \in \mathbb{B}^n)\ \mathtt{HAS\_TYPE}\ \mathsf{P}_\mathsf{N}$

Once we have seen the general structure of our transformation, let us focus on the `Elim_bool` function in Section 5 and on the `Elim_let` function in Section 6.

5. $\surd$ AND / ELIMINATION IN BOOLEAN EXPRESSIONS: `Elim_bool`

In this section, we describe the function `Elim_bool` that transforms a well typed boolean expression built with comparisons into an equivalent expression which is free of divisions and square roots (*i.e.,* of class B_{let}). The elimination of square roots and divisions in boolean formulas is a particular case of the quantifier elimination over real closed fields *e.g.,* the formula $\sqrt{x} + \frac{3}{\sqrt{y}} > 4$ can be rewritten as:

$$\exists\ x',\ y',\ y'',\ x'^2 = x\ \wedge\ y'^2 = y\ \wedge\ y'' \times y' = 3\ \wedge\ x' + y'' > 4$$

Then by eliminating the three quantifiers, we get an equivalent boolean formula without any division or square root. The general quantifier elimination with cylindrical algebraic decomposition [Tar51, Col76] has been implemented as QEPCAD[1] but this algorithm does not succeed in most of our cases due to the large number of free variables in the context of program transformation. The elimination of square roots and division in formulas has also been studied by V. Weispfenning in its restriction of the quantifier elimination to the quadratic case [Wei97]. However we do not need a general quantifier elimination algorithm nor even the quadratic case restriction for the elimination of square roots and division. Thus, we define a specific transformation for square root and division elimination that allows us to eliminate all occurrences of one square root in a single step and produce more concise code.

5.1 Expressions subtyping

As introduced in Section 2, we describe the input and output program type of the transformations using the subtypes of Prog. `Elim_bool` is a partial function that only applies to a subtype of the general programs that is:

$\mathtt{comparison\_expression}(\Gamma)\colon\ \mathtt{TYPE} =$
$\{\ \mathsf{e1}\ \mathsf{op}\ \mathsf{e2}\colon\ \mathsf{Prog}\ \mid\ Ty_\Gamma(\mathsf{e1}\ \mathsf{op}\ \mathsf{e2}) = \mathbb{B}\ \ \&\ \ \mathsf{op} \in Cbop\ \ \&\ \ \mathsf{e1}\colon\ \mathsf{N}_{\surd,/}\ \ \&\ \ \mathsf{e2}\colon\ \mathsf{N}_{\surd,/}\}$

We also define the type of well typed numerical expressions:

$\mathtt{wt\_num\_expr}(\Gamma)\colon\ \mathtt{TYPE} = \{\ \mathsf{e}\colon\ \mathsf{N}_{\surd,/}\ \mid\ Ty_\Gamma(\mathsf{e}) = \mathbb{R}\}.$

[1] see `http://www.usna.edu/cs/~qepcad/B/QEPCAD.html`

It is more convenient to explicitly have the syntactic form of the arguments e1 and e2 using the type $\mathsf{N}_{\surd,/}$, it allows us to prove completeness of the function without extracting their syntactic structure from the type. However this needs to be done when we invoke the `Elim_bool` itself using the following lemma :

`expr_wt_num_is_num_expr: LEMMA` $\forall$ `(e:E |`$\exists\Gamma,\ Ty_\Gamma(\mathsf{e}) = \mathbb{R}$`) HAS_TYPE` $\mathsf{N}_{\surd,/}$

The well typed hypothesis is also used to ensure arguments of projections are either projections or variables (since the projections on pair constructors has already been reduced and no constant has type pair):

`no_op_st_fst_wt_num: LEMMA` $\forall$ `(x: {p:` $\mathsf{N}_{\surd,/}$ `|` $wtp(\mathsf{p})$`}), (y: Prog):`
`(fst?(x) OR snd?(x)) & y «` $\mathsf{x} \Rightarrow$ `fst?(y) OR snd?(y) OR Var?(y)`

where `fst?`, `snd?` and `Var?` are predicates that are true if the element has the corresponding constructor as head symbol.

Using these type definitions we can define the first step of elimination of square roots and divisions which is a reduction to a normal form for numerical expressions $\mathsf{N}_{\surd,/}$ with division as head operator.

5.2 One division normal form

The elimination of the division is in fact quite simple since every numerical expression on $+$, $-$, $\times$, $/$, $\surd$ can be represented with only one division at head. Therefore the expression we get by reducing expressions to the same denominator without splitting square roots corresponds to the following normal form:

Definition 5.2.1. **Division and polynomial normal forms**

$$\begin{array}{lllll} \mathsf{DNF} := \mathsf{PNF} \mid \frac{\mathsf{PNF}}{\mathsf{PNF}} & & & & \\ \mathsf{PNF} := \mid \mathsf{PNF} + \mathsf{PNF} & \mid \mathsf{PNF} \times \mathsf{PNF} & \mid -\,\mathsf{PNF} & \mid \sqrt{\mathsf{N}_{\surd,/}} & \\ \quad \mid \mathsf{fst\ PNF} & \mid \mathsf{snd\ PNF} & \mid \mathsf{Constant} & \mid \mathsf{Var} & \end{array}$$

Reduction to division normal form is done by the `to_dnf` function that transforms any well typed numerical expression into an equivalent one with only one division (if there is any) as head operation. This corresponds to the following specification:

```
to_dnf(Γ)( e: wt_num_expr(Γ) ):
 { eout: DNF | sem_ty_nf_eq(e)(eout) & sq_number_eq(e,eout) }
```

which implements the following set of rules :

Definition 5.2.2. **Head division reduction**

$$\begin{array}{llll} \mathsf{e}_1 + \frac{\mathsf{e}_2}{\mathsf{e}_3} \longrightarrow \frac{\mathsf{e}_3.\mathsf{e}_1 + \mathsf{e}_2}{\mathsf{e}_3} & \frac{\mathsf{e}_1}{\mathsf{e}_2} + \mathsf{e}_3 \longrightarrow \frac{\mathsf{e}_1 + \mathsf{e}_2.\mathsf{e}_3}{\mathsf{e}_2} & \textit{(HD +)} \\ -\frac{\mathsf{e}_1}{\mathsf{e}_2} \longrightarrow \frac{-\mathsf{e}_1}{\mathsf{e}_2} & -(-\mathsf{e}_1) \longrightarrow \mathsf{e}_1 & \textit{(HD -)} \\ \mathsf{e}_1 \times \frac{\mathsf{e}_2}{\mathsf{e}_3} \longrightarrow \frac{\mathsf{e}_1 \times \mathsf{e}_2}{\mathsf{e}_3} & \frac{\mathsf{e}_1}{\mathsf{e}_2} \times \mathsf{e}_3 \longrightarrow \frac{\mathsf{e}_1 \times \mathsf{e}_3}{\mathsf{e}_2} & \textit{(HD $\times$)} \\ \frac{\frac{\mathsf{e}_1}{\mathsf{e}_2}}{\mathsf{e}_3} \longrightarrow \frac{\mathsf{e}_1}{\mathsf{e}_2.\mathsf{e}_3} & \frac{\mathsf{e}_1}{\frac{\mathsf{e}_2}{\mathsf{e}_3}} \longrightarrow \frac{\mathsf{e}_1.\mathsf{e}_3}{\mathsf{e}_2} & \textit{(HD /)} \end{array}$$

Remark 2. *This transformation is only correct in the context of the no fail assumption since the semantics equivalence of the rule* $\frac{\frac{\mathsf{e1}}{\mathsf{e2}}}{\mathsf{e3}} \longrightarrow \frac{\mathsf{e1.e3}}{\mathsf{e2}}$ *only holds when* $[\![\mathsf{e3}]\!] \neq 0$.

REMARK 3. `sq_number_eq(e,eout)` *is a predicate that will be used to prove the termination of the* `Elim_bool` *function. It states that every square root that appears in the output was already in the input:*

$$\texttt{sq\_number\_eq(e,eout)} = \forall\ \mathsf{sq},\ \sqrt{\mathsf{sq}} \lessdot\!\!\lessdot \mathsf{eout} \Rightarrow \sqrt{\mathsf{sq}} \lessdot\!\!\lessdot \mathsf{e}$$

where $\mathsf{x} \lessdot\!\!\lessdot \mathsf{y}$ *means that* x *is a subterm of* y *or is equal to* y.

5.3 Division elimination rules

Once we have a comparison between two DNF form, the head division can be easily eliminated by multiplying both arguments by the denominators.

DEFINITION 5.3.1. **Elimination of division with cases**

—*when* $\Re \in \{=, \neq\}$*:* $\frac{\mathsf{e1}}{\mathsf{e2}}\ \Re\ \frac{\mathsf{e3}}{\mathsf{e4}} \longrightarrow\ \mathsf{e1.e4} - \mathsf{e3.e2}\ \Re\ 0$

—*when* $\Re \in \{>, <, \geq, \leq\}$*:* $\frac{\mathsf{e1}}{\mathsf{e2}}\ \Re\ \frac{\mathsf{e3}}{\mathsf{e4}} \longrightarrow$
$(\mathsf{e1.e4} - \mathsf{e3.e2}\ \Re\ 0 \wedge \mathsf{e2.e4} > 0) \vee (\mathsf{e3.e2} - \mathsf{e1.e4}\ \Re\ 0 \wedge \mathsf{e2.e4} < 0)$

But since we want to avoid the creation of new comparisons due to the case distinction depending on the sign of the denominators when multiplying, we will sometimes prefer the following elimination rule that multiplies both arguments by the square of the denominators:

DEFINITION 5.3.2. **Elimination of division**

—*when* $\Re \in \{=, \neq\}$*:* $\frac{\mathsf{e1}}{\mathsf{e2}}\ \Re\ \frac{\mathsf{e3}}{\mathsf{e4}} \longrightarrow\ \mathsf{e1.e4} - \mathsf{e3.e2}\ \Re\ 0$

—*when* $\Re \in \{>, <, \geq, \leq\}$*:* $\frac{\mathsf{e1}}{\mathsf{e2}}\ \Re\ \frac{\mathsf{e3}}{\mathsf{e4}} \longrightarrow\ \mathsf{e1.e2.e4}^2 - \mathsf{e3.e4.e2}^2\ \Re\ 0$

REMARK 4. *The application of this rule eliminates the last division that is not inside a square root. Therefore the expressions produced are boolean combinations of relations between numerical expressions in* PNF *and* 0.

These elimination rules are implemented in PVS with the following function:

```
elim_div_rule1(Γ)(p: comparison_expression(Γ)):
    { e1 op e2: comparison_expression(Γ) | sq_number_eq(p,e1) &
          sem_ty_nf_eq(p)(e1 op e2) & PNF?(e1) & e2 = 0 }
```

The OCaml implementation gives the option to use either this rule or the usual one (with case distinction) but only this one was proven in PVS since it gives better results for the size of the produced code.
The head division being eliminated, we have to eliminate a top level square root that will make the divisions in the arguments of square roots appear at top level and allow us to continue the elimination of both operations.

5.4 Square root elimination

For every comparison between a PNF form and 0 we choose one square root that is at top-level, that means this square root does not appear in the argument of another square root.

DEFINITION 5.4.1. **Top level square root** *Given* e *an arithmetic expression, an expression* q *is a top level square root of* e *when:*

$$\sqrt{\mathsf{q}} \lessdot\!\!\lessdot \mathsf{e}\ \wedge\ \neg\exists\ \mathsf{q'},\ \sqrt{\mathsf{q'}} \lessdot\!\!\lessdot \mathsf{e}\ \wedge\ \sqrt{\mathsf{q}} \ll \sqrt{\mathsf{q'}}$$

Given a top level square root Q we factorize the PNF expression in the form ($\mathsf{P}.\sqrt{\mathsf{Q}}+\mathsf{R}$) where $\sqrt{\mathsf{Q}}$ does not appear in P or R:

```
factorize_sqrt(Γ)(e:wt_N_expr(Γ) | PNF?(e))(q:wt_N_expr(Γ)):
  { p,r: wt_N_expr(Γ) | sem_ty_nf_eq(e)(p.√q + r)
      sq_number_but_q_eq(e)(q)(p) & sq_number_but_q_eq(e)(q)(r)}
```

where `sq_number_but_q_eq` states that every square root of an output was already in an input and is different from `q`. Then by splitting cases depending on the signs of P and R we get a new formula equivalent to $\mathsf{P}.\sqrt{\mathsf{Q}}+\mathsf{R}>0$ under the condition that $\mathsf{Q}\geq 0$, *e.g.,* when $\Re$ is $>$, we have the following rule:

$$(\mathsf{P}.\sqrt{\mathsf{Q}}+\mathsf{R})>0 \quad\longrightarrow\quad \begin{array}{l}(\mathsf{P}>0\ \wedge\ \mathsf{R}>0)\ \vee\\ (\mathsf{P}>0\ \wedge\ \mathsf{P}^2.\mathsf{Q}-\mathsf{R}^2>0)\ \vee\\ (\mathsf{R}>0\ \wedge\ \mathsf{R}^2-\mathsf{P}^2.\mathsf{Q}>0)\end{array}$$

In the last expression, one can notice some of the comparisons that are present in this formula are equal. Thus, in order to reduce the size of the final expression, we name them and share their different occurrences instead of duplicating them, this is the reason why we allowed variable definitions of boolean expressions in the B_{let} subtype, therefore the complete elimination rule of one square root is:

$$(\mathsf{P}.\sqrt{\mathsf{Q}}+\mathsf{R})>0 \longrightarrow \begin{array}{l}\mathsf{let\ xsq}=((\mathsf{P}>0,\mathsf{R}>0),(\mathsf{P}^2.\mathsf{Q}-\mathsf{R}^2>0,\mathsf{P}^2.\mathsf{Q}-\mathsf{R}^2\neq 0))\ \mathsf{in}\\ \quad \mathsf{fst(fst(xsq))}\wedge\mathsf{snd(fst(xsq))}\vee\\ \quad\ \mathsf{fst(fst(xsq))}\wedge\mathsf{fst(snd(xsq))}\vee\\ \quad\quad \mathsf{snd(fst(xsq))}\wedge\neg\ \mathsf{fst(snd(xsq))}\wedge\mathsf{snd(snd(xsq))}\end{array}$$

This corresponds to the elimination of one square root in a $>$ comparison, similar rules can be defined for the other comparison operators.

Since the transformation we are defining in Pvs only applies on comparisons, we first have to recursively eliminate all the square roots and divisions from the four produced comparisons before combining them to produce a boolean expression equivalent to the input comparison. Therefore the implementation of these rules relies on composition functions such as

```
name_comp(xsq: Var, e1, e2, e3, e4: Prog): Prog =
    let xsq = ((e1,e2),(e3,e4)) in
          fst(fst(xsq)) ∧ snd(fst(xsq)) ∨
            fst(fst(xsq)) ∧ fst(snd(xsq)) ∨
              snd(fst(xsq)) ∧¬ fst(snd(xsq)) ∧ snd(snd(xsq))
```

and on elimination rules such as

```
elim_sqrt_rule_gt(Γ)(p,q,r: wt_num_expr(Γ)) :
 { e1, e2, e3, e4: comparison_expression(Γ) |
   ∀(x: Var): sem_ty_nf_eq(p.√q + r > 0)(name_comp(x,e1,e2,e3,e4))} =
     (p > 0,r > 0,p².q − r² > 0,p².q − r² ≠ 0)
```

Using reduction to DNF, square root factorization and the elimination rules for division and square root, we can define the main algorithm which transforms a comparison into an equivalent fragment of code that contains neither division nor square root. This algorithm is a recursive combination of these four transformations:

DEFINITION 5.4.2. `elim_bool` **description**
While the comparisons contains divisions or square roots, do:
—*Reduce to DNF*
—*Eliminate the head division*
—*Factorize using one top level square root*
—*Eliminate that square root*

The full transformation of the comparisons is defined by the `elim_bool` function whose specification was given in Definition 4.1. There are two distinct kinds of TCC that have to be proven:

—`p:` B_{let}, comes from the fact that the `elim_bool` functions returns either a PNF comparison whose square root list is empty or a composition (such as `name_comp`) of expressions in B_{let}

—`sem_ty_nf_eq`, this is proven by composition of the `sem_ty_nf_eq` predicates of the reductions and elimination rules

In order to ensure the termination of the algorithm, we prove that the number of distinct square roots that are sub-terms of the comparison strictly decreases. Indeed, after applying the 4 steps, every square root in the output comparisons was already in the input and is different from the one we eliminated, which was also in the input.

EXAMPLE 5.4.1. / **and** $\sqrt{\ }$ **elimination**

$$\frac{x+\sqrt{y}}{z} > 0 \longrightarrow \begin{array}{l}\mathsf{let\ xsq} = ((z>0,x.z>0),(z^2.y-(x.z)^2>0,z^2.y-(x.z)^2\neq 0))\ \mathsf{in}\\ \quad \mathsf{fst(fst(xsq))} \wedge \mathsf{snd(fst(xsq))} \vee\\ \quad\ \mathsf{fst(fst(xsq))} \wedge \mathsf{fst(snd(xsq))} \vee\\ \quad\ \mathsf{snd(fst(xsq))} \wedge \neg\ \mathsf{fst(snd(xsq))} \wedge \mathsf{snd(snd(xsq))}\end{array}$$

Complexity. In a comparison A, if we write $|A|_{\sqrt{\ }}$ the number of distinct square roots, the number of comparisons produced by eliminating the square roots in this comparison is bounded by $4^{|A|_{\sqrt{\ }}}$. We did not study the exact complexity of the algorithm since the reduction into DNF and the factorization may also increase the size of the formula. Nevertheless, experimentally our specialized algorithm is able to transform bigger formulas than the QEPCAD implementation of the quantifier elimination by cylindrical algebraic decomposition, *e.g.,*

$$\exists sq, sq^2 = q \wedge sq \geq 0 \wedge (p_1.sq + r_1).(p_2.sq + r_2).(p_3.sq + r_3) > 0$$

Indeed, the complexity of our transformation does not depend on the number of variables of the formula but only on the number of square roots and divisions.

The elimination of square roots and divisions being defined, we detail in the following section the second transformation used in the main transformation. It transforms a variable definition into an equivalent one without square roots or divisions.

6. TRANSFORMATION OF VARIABLES DEFINITIONS: `Elim_let`

In this section, our goal is to define how to transform a variable definition in order to take the divisions and square roots out of the body. We will describe a function that

corresponds to the specification given in Definition 4.1. We noticed that inlining the variable definitions satisfies this specification but it leads to an explosion of the size of the code therefore we will now present a way to inline only the divisions and square roots as in Example 6.1.

The intuitive idea for suppressing these operations in a variable definition is to only name its subexpressions that are free of them. Then we replace the variable in the scope with an expression depending on these new defined variables, as in the following example:

EXAMPLE 6.1.
$\mathsf{let\ x = if\ y > 0\ then\ (a1 + a2)/b\ else\ c + \sqrt{d1.d2}\ in\ P} \longrightarrow$
$\mathsf{let\ (x0,x1,x2) = if\ y > 0\ then\ (a1 + a2,b,0)\ else\ (c,1,d1.d2)\ in\ P[x:= \frac{x0+\sqrt{x2}}{x1}]}$

Square roots and divisions that were used to define x are now explicit in P. In this section, we only use the *mathematic* font in order to describe the transformation and for clarity we use multiple variable definitions.

6.1 Transformation of the variable definition code

The elimination of square roots and divisions from a variable definition (let x = p1 in p2 where x: Var, p1: $\mathsf{P_{N_{\surd,/}}}$ and p2: P) relies on a decomposition of its body p1 in:

—the *program part* Pp of type $\mathsf{E^n_{N_{\surd,/}}} \longrightarrow \mathsf{P_{N_{\surd,/}}}$ and $\mathsf{E^n_N} \longrightarrow \mathsf{P_N}$

—the list of expressions $(\mathsf{e}_1, ..., \mathsf{e}_n)$ of $\mathsf{E^n_{N_{\surd,/}}}$ such that $Pp(\mathsf{e}_1, ..., \mathsf{e}_n) = \mathsf{p1}$, that decomposes itself in:
- the *template* $Temp$ of type $\mathsf{E^k_N} \longrightarrow \mathsf{E_{N_{\surd,/}}}$
- the list of k-tuples of division and square root free *subexpressions* $\mathsf{se}_1, ..., \mathsf{se}_n$ of type $\mathsf{E^k_N}$ such that: $\forall i \in [1...n], Env,\ [\![Temp(\mathsf{se}_i)]\!]_{Env} = [\![\mathsf{e}_i]\!]_{Env}$

Given such a decomposition, we have that:

$$\forall Env, [\![\mathsf{p1}]\!] = [\![Pp(Temp(\mathsf{se}_1), ..., Temp(\mathsf{se}_n))]\!]_{Env}$$

and we transform the variable definition in the following way:

DEFINITION 6.1.1. **Variable definition transformation**
A variable definition is transformed by commuting elements of its decomposition:

$$\begin{array}{l}\mathsf{let\ x} = Pp(Temp(\mathsf{se}_1), ..., Temp(\mathsf{se}_n)) \\ \mathsf{in\ p2}\end{array} \longrightarrow \begin{array}{l}\mathsf{let\ } (\mathsf{x}_{\epsilon_1}, ..., \mathsf{x}_{\epsilon_n}) = Pp(\mathsf{se}_1, ..., \mathsf{se}_n) \\ \mathsf{in\ p2[\ x} := Temp(\mathsf{x}_{\epsilon_1}, ..., \mathsf{x}_{\epsilon_n})\ ]\end{array}$$

The is not a unique solution for the template computation and due to the complexity of the elimination of square roots and divisions we really focus on minimizing the number of square roots that appear in it (the template is then inlined). Therefore, for genericity reasons, the template computation is only axiomatized in PVS (we assume that such a function exists) but the OCaml program implements one version.

We now begin the description of the two decompositions. In order to simplify the presentation, we introduce the following notation. For every program p, $(fun\ (x_1, ..., x_n) \to \mathsf{p})$ is a function of $\mathsf{P}^n \longrightarrow \mathsf{P}$, such that:

$$(fun\ (x_1, ..., x_n) \to \mathsf{p})(u_1, ..., u_n) = \mathsf{p}[x_1 := u_1; ...; x_n := u_n]$$

The first decomposition transforms the body of the definition that is in $\mathsf{P_{N_{\surd,/}}}$ into

a *program* part (the part that contains the local variable definitions and tests) and the *expression part*.

6.2 Program and expression part decomposition

We define the following recursive algorithm *Decompose*, that computes from a program p in $\mathsf{P_{N_{\surd,/}}}$, its *program part* and its *expression part*.

DEFINITION 6.2.1. **Program and expression part decomposition**
Decompose(p*) =*
—*if* $\mathsf{p} \in \mathsf{E_{N_{\surd,/}}}$ *then* $((fun\ x \to x), [\mathsf{p}])$
—*if* p = let y = a in p' *then*
—*(pp,ep) := Decompose(*p'*)*
—*return* $((fun\ x \to$ let y = a in $pp(x)), ep)$
—*if* p = if B then $\mathsf{p_1}$ else $\mathsf{p_2}$ *then*
—*(*pp_1*,*ep_1*) := Decompose(*$\mathsf{p_1}$*)*
—*(*pp_2*,*ep_2*) := Decompose(*$\mathsf{p_2}$*)*
—*return* $((fun\ (x_1, x_2) \to$ if B then $pp_1(x_1)$ else $pp_2(x_2)), ep_1 @ ep_2)$

where @ denotes the concatenation of lists.

EXAMPLE 6.2.1.
Decompose(if F then let y = a in a + $\sqrt{\mathsf{b}}$ else c) =
$(fun\ (x, y) \to$ if F then let y = a in x else y, (a+$\sqrt{\mathsf{b}}$,c))

The program p being in $\mathsf{P_{N_{\surd,/}}}$, neither the local variable definitions bodies nor the boolean arguments of the tests can contain division or square root. Therefore if we apply Pp to a tuple of expressions in $\mathsf{E_N}$, the result does not contain any divisions or square roots.

LEMMA 6.2.1. **Program part restriction**

$$\forall\ e_1, ..., e_k :\ \mathsf{E_N} \Longrightarrow Pp(e_1, ..., e_k) \in \mathsf{P_N}$$

Now we will see how we can decompose the expression part in order to remove square roots and divisions from it.

6.3 Expression decompositions

Initially we will see how to decompose a single expression by assuming that the body of the definition does not contain any test, thus the expression part is a list of one element, then we will extend this definition to any list of expressions.

6.3.1 *Template of an expression.* The idea for transforming a variable definition with an expression is to transform this definition into several variable definitions that correspond to the subexpressions that are free of square roots and divisions. Therefore we need to introduce this decomposition of an expression as the application of a function to square roots and divisions free expressions:

DEFINITION 6.3.1. **Template of expression**
Given an expression $\mathsf{e} \in \mathsf{E_{N_{\surd,/}}}$*, a template for* e *is a function* t *of* $\mathsf{E_N^k} \longrightarrow \mathsf{E_{N_{\surd,/}}}$ *such that:*

$$\exists\ \mathsf{e}_1, ..., \mathsf{e}_k :\ \mathsf{E_N},\ \mathsf{e} = t(\mathsf{e}_1, ..., \mathsf{e}_k)$$

The template and subexpression tuple of a unique expression can be computed by giving the tuple of square roots and division free subexpressions.

EXAMPLE 6.3.1.

$\frac{a_1.a_2+b\sqrt{c}}{d.\sqrt{e_1+e_2+\sqrt{c}}} = (fun\ (\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3,\mathsf{sq}_1,\mathsf{sq}_{21}) \to \frac{\mathsf{x}_1+\mathsf{x}_2\sqrt{\mathsf{sq}_1}}{\mathsf{x}_3\sqrt{\mathsf{sq}_{21}+\sqrt{\mathsf{sq}_1}}})(a_1.a_2, b, d, c, e_1+e_2)$

We use special names for square roots because we want to avoid creating different names for the same square root.

EXAMPLE 6.3.2. **Unique square root name**

$$\begin{array}{l} \mathit{let}\ x = 2.\sqrt{a}\ \mathit{in} \\ \mathit{let}\ y = 5 + \sqrt{a}\ \mathit{in}\ P \end{array} \longrightarrow \begin{array}{l} \mathit{let}\ (x_1, sq_1) = (2, a)\ \mathit{in} \\ \mathit{let}\ (y_1) = 5\ \mathit{in} \\ P[x := \mathsf{x}_1.\sqrt{\mathsf{sq}_1};\ y := \mathsf{y}_1 + \sqrt{\mathsf{sq}_1}] \end{array}$$

Spotting the identical square roots is essential in order to preserve a reasonable size for the produced code.

6.3.2 *Common template of expressions.* Given the definition of a template of one expression, we will now extend this decomposition (template, square root and division free sub expressions) to programs that contain tests. If the body contains a test as in Example 6.1, its *expression part* is a list of expressions (*i.e.,*$[(\mathsf{a1} + \mathsf{a2})/\mathsf{b}; \mathsf{c} + \sqrt{\mathsf{d1.d2}}]$). The objective is now to decompose these expressions using the same template (*i.e.,*$\frac{\mathsf{x0}+\sqrt{\mathsf{x2}}}{\mathsf{x1}}$), we call it the common template:

DEFINITION 6.3.2. **Common template of expressions** *A common template of two expressions e_1 and e_2 is a function t that is a template of both e_1 and e_2, that means t satisfies:*

$$\exists\ se_1, se_2 :\ \mathsf{E}^k_{\mathsf{N}}, e_1 = t(se_1)\ and\ e_2 = t(se_2)$$

Existence: Let $x_1, ...x_k$ be the local variables corresponding to e_1 and $x'_1, ...x'_{k'}$ the one to e_2, then if we define t as:

$$fun\ (s, x_1, ..., x_k, x'_1, ..., x'_{k'}) \to (s \times e_1 + (1-s) \times e_2)$$

we have: $t(1, x_1, ...x_k, 0, ..., 0) = e_1\ \wedge\ t(0, 0, ..., 0, x'_1, ...x'_{k'}) = e_2$, it is a common template of e_1 and e_2. This is only an example since we use a more compact form in our transformation in order to limit the number of square roots produced by the template but we will not describe the common template computation in this paper. Thus the transformation of a variable definition with tests has the following form:

EXAMPLE 6.3.3. **Declaration with test**

$$\begin{array}{l} \mathsf{let\ x} = \\ \quad \mathsf{if\ F} \\ \quad\quad \mathsf{then}\ \mathsf{a}_1 + \mathsf{a}_2.\sqrt{\mathsf{b}_1 + \mathsf{b}_2.\sqrt{\mathsf{b}_3}} \\ \quad\quad \mathsf{else}\ \frac{\mathsf{c}_1}{\mathsf{c}_2+\mathsf{c}_3.\sqrt{\mathsf{d}_1}} \\ \quad \mathsf{in\ P} \end{array} \longrightarrow \begin{array}{l} \mathsf{let}\ (\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3,\mathsf{x}_4,\mathsf{sq}_1,\mathsf{sq}_2,\mathsf{sq}_3) = \\ \quad \mathsf{if\ F} \\ \quad\quad \mathsf{then}\ (\mathsf{a}_1, \mathsf{a}_2, 1, 0, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_3) \\ \quad\quad \mathsf{else}\ (\mathsf{c}_1, 0, \mathsf{c}_2, \mathsf{c}_3, 0, 0, \mathsf{d}_1) \\ \quad \mathsf{in\ P}[\mathsf{x} := t(\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3,\mathsf{x}_4,\mathsf{sq}_1,\mathsf{sq}_2,\mathsf{sq}_3)] \end{array}$$

where $t = fun\ (x_1, x_2, x_3, x_4, sq_1, sq_2, sq_3) \to \frac{x_1+x_2.\sqrt{sq_1+sq_2.\sqrt{sq_3}}}{x_3+x_4.\sqrt{sq_3}}$ is the common template of $\mathsf{a}_1 + \mathsf{a}_2.\sqrt{\mathsf{b}_1 + \mathsf{b}_2.\sqrt{\mathsf{b}_3}}$ and $\frac{\mathsf{c}_1}{\mathsf{c}_2+\mathsf{c}_3.\sqrt{\mathsf{d}_1}}$.

The definition of a common template of two expressions extends naturally to several expressions. Therefore we are able to build a common template for any list of expressions in $\mathsf{E}_{\mathsf{N}_{\sqrt{},/}}$.

6.4 Correctness

We can prove that this transformation is correct by using the following semantical equivalences (with some conditions on the variable names that enforce some renaming).

LEMMA 6.4.1. **Semantics equivalences**

let x = let y = e in t(y) in P	$\longleftrightarrow$	*let y = e in let x = t(y) in P*	*(lift-let)*
if F then t(e_1) else t(e_2)	$\longleftrightarrow$	*t(if F then e_1 else e_2)*	*(lift-if)*
let x = e in p	$\longleftrightarrow$	*p[x := e]*	*(inline)*

These rules allow us to prove that the rule used to transform variable definitions introduced in Definition 6.1.1 preserves the semantics. This transformation being described, it completes the description of the transformation algorithm. The last section presents some remarks about the effective implementation of this transformation.

7. PRACTICAL ASPECTS OF THE PROGRAM TRANSFORMATION

Exact computation with $+, -, \times$. The air traffic management system introduced in [NMD10] computes with distances and other geometric quantities. Therefore it has use of the square root and division operations. There is a fundamental difference between division and square roots on one side and the three other arithmetic operations that are addition multiplication and subtraction on the other side. Indeed, let us now introduce the subset $\mathbb{D}$ of $\mathbb{R}$. $\mathbb{D}$ is the set of dyadic rational numbers, the rational numbers which denominator is a power of 2. Therefore every element of $\mathbb{D}$ can be exactly represented using a finite sequence of bits and this set $\mathbb{D}$ is closed under addition, multiplication and subtraction, whereas division and square roots can not be precisely defined (1/5 has no finite binary representation) and will force us to use round offs.

Computing in $\mathbb{D}$ with addition, multiplication and subtraction can be done exactly by using a dynamic representation of real numbers which will allow us to use all the bits we need to avoid losing accuracy during computation (*e.g.,* the product of two numbers of size n can be stored in a number of size $2.n$). Certainly, this kind of computation does not respect the constraint of embedded systems that requires to know at compile time the memory the program will use at run time. But, since our language does not contain loop or recursion, a simple static analysis can give us the number of bits required by every computation depending the number of bits of the inputs. Being able to compute exactly with addition, multiplication and subtraction is the reason why we want to eliminate square root and division operations.

Experiments. We have tested this transformation on several functions defined by the NASA Langley Formal Method Team in their Airborne Coordinated Conflict Resolution and Detection (ACCoRD) framework. Since our language does not support function definitions yet, we had to inline the calls by hand in the target program. We tried our transformation on the `cd3d` program, which is an improved

version of the `cd2d` function defined in [MBMD09], this function contains only one distinct square roots which is used many times along with several divisions. We also tested the `track_line` function, defined in [NMD10], this function contains two distinct and dependent square roots and also several divisions. On these two functions we computed the transformation and therefore we can give the size of the produced code but we also computed an estimation of the required memory size if we want to compute on that produced code using exact computation with $+, -, \times$. Assuming the inputs are represented on 64 bits, that gives the following results:

Function	Input code	Output code	Required memory
`cd3d`	2,3 KB	13 KB	15 Kb
`trackline`	1 KB	13 KB	57 Kb

The size and the memory required by the output programs quickly growing with the number of square roots and division, it is absolutely critical to keep that number as low as possible during the transformation process in order to get reasonable sizes.

Conclusion

In this paper, we described a way to transform any program built with variable definitions and tests into another semantically equivalent one where the control flow never depends on the result of a square root or division, thus protecting this control flow from rounding errors introduced by these operations. This transformation allows proofs done on the abstract semantics to still hold on the concrete one. The programs produced by this transformation also respects the constraints of embedded code since they do not use any dynamic structures.

The main issues of this transformation were not only to define procedures that remove divisions and square roots from nearly every part of the program but also to keep the size of the produced code in an acceptable range. This is the reason why we did not define the computation of the template, since several complex algorithms may be used to generate the most appropriate template in order to minimize the size of the produced code. Nevertheless, validity of the proof only relies on the correctness of the template used in this optimization. This work led us to define the problem of template generation that may be of general interest and will be discussed in future work.

Most of the program generic transformations such as the elimination of square roots and divisions in expressions have already been formalized and proved in the Pvs proof assistant. Both the Pvs development and the OCaml implementation can be found on the web site of the author. The Pvs specification has also been used to define a Pvs strategy using computational reflection as presented in [Ner13]. Future work includes extending this transformation to more complex languages which contain structures such as function definitions and bounded loops and keep reducing the size of the produced code by using static analysis techniques such as using the information given by the tests values in the corresponding branches during the transformation.

Acknowledgement. I would like to thank both my PhD. advisors Gilles Dowek and Cesar Muñoz for this topic proposal and the helpful discussions and Catherine Dubois, Cyril Cohen and the anonymous reviewers for their useful comments on

this paper. 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

[BCRO86] Hans-Juergen Boehm, Robert Cartwright, Mark Riggle, and Michael J. O'Donnell. Exact real arithmetic: a case study in higher order programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 162–173, New York, NY, USA, 1986. ACM.

[BF07] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In Peter Kornerup and Jean-Michel Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[BM06] Sylvie Boldo and César Muñoz. A formalization of floating-point numbers in PVS. Report NIA Report No. 2006-01, NASA/CR-2006-214298, NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, 2006.

[Bos03] Alin Bostan. *Algorithmique efficace pour des opérations de base en Calcul formel*. Ph.d. thesis, Ecole polytechnique, 2003.

[CMC08] Liqian Chen, Antoine Miné, and Patrick Cousot. A sound floating-point polyhedra abstract domain. In G. Ramalingam, editor, *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2008.

[Coh12] Cyril Cohen. Construction of real algebraic numbers in Coq. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, Princeton, États-Unis, August 2012. Springer.

[Col76] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis. *SIGSAM Bull.*, 10:10–12, February 1976.

[DGL04] Pietro Di Gianantonio and Pier Luca Lanzi. Lazy algorithms for exact real arithmetic. *Electron. Notes Theor. Comput. Sci.*, 104:113–128, November 2004.

[DMM05] Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In *IEEE Symposium on Computer Arithmetic*, pages 188–195, 2005.

[Har95] John Harrison. Floating point verification in HOL. In Phillip J. Windley, Thomas Schubert, and Jim Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 186–199, Aspen Grove, Utah, 1995. Springer.

[IEE85] IEEE. *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[KS11] Robbert Krebbers and Bas Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1), 2011.

[Mar09] Matthieu Martel. Program transformation for numerical precision. In *PEPM*, pages 101–110, 2009.

[MBMD09] Jeffrey Maddalon, Ricky Butler, César Muñoz, and Gilles Dowek. Mathematical basis for the safety analysis of conflict prevention algorithms. Technical Memorandum NASA/TM-2009-215768, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2009.

[Min95] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS, 1995.

[Ner12] Pierre Neron. A formal proof of square root and division elimination in embedded programs. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 256–272. Springer, 2012.

[Ner13] Pierre Neron. Square root and division elimination in PVS. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 457–462. Springer, 2013.

[NMD10] Anthony Narkawicz, César Muñoz, and Gilles Dowek. Formal verification of air traffic prevention bands algorithms. Technical Memorandum NASA/TM-2010-216706, NASA, 2010.

[NMD12] Anthony Narkawicz, Céar Muñoz, and Gilles Dowek. Provably correct conflict prevention bands algorithms. *Science of Computer Programming*, 77(1–2):1039–1057, September 2012.

[O'C08] Russell O'Connor. Certified exact transcendental real number computation in Coq. In OtmaneAit Mohamed, César César, Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 246–261. Springer Berlin Heidelberg, 2008.

[Sim98] Alex K. Simpson. Lazy functional algorithms for exact real functionals. In Luboš Brim, Jozef Gruska, and Jiří Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 456–464. Springer Berlin Heidelberg, 1998.

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry.* Univ. of California Press, 2nd edition, 1951.

[Vui87] Jean Vuillemin. Exact real arithmetic with continued fractions. Rapport de recherche RR-760, INRIA, 1987.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, 8(2):85–101, 1997.

[Wie80] Edwin Wiedmer. Computing with Infinite Objects. *Theoretical Computer Science*, 10:133–155, 1980.