Elimination of Square Roots and Divisions by Partial Inlining

Pierre Neron
TU Delft
p.j.m.neron@tudelft.nl

Abstract
Computing accurately with real numbers is always a challenge. This is particularly true in critical embedded systems since memory issues do not allow the use of dynamic data structures. This constraint imposes a finite representations of the real numbers, provoking uncertainties and rounding errors that might modify the actual behavior of a program from its ideal one. This article presents a solution to this problem with a program transformation that eliminates square roots and divisions in straight line programs without nested function calls. These two operations are the source of infinite sequences of digits in numerical representations, thus, eliminating these operations allows to compute exactly using for example a fixed-point number representation with a sufficient number of bits. In order to avoid an explosion of the size of the produced code this transformation relies on a particular anti-unification to realize a partial inlining of the variable and function definitions. This transformation targeting code for aeronautics certified in PVS, we want to prove the semantics preservation in this proof assistant. Thus we use both an OCaml implementation and the subtyping features of PVS to ensure the correctness of the transformation by defining a proof-producing (certifying) program transformation, providing a specific semantics preservation lemma for every definition in the transformed program.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; D.3.4 [Programming Languages]: Processors

Keywords Program Transformation, Real Numbers, Program Verification, Anti-unification, Embedded Systems.

1. Introduction
The problem of computation over real numbers is of critical importance since such computations cause many slippery errors. By using a finite number of bits to represent real number values, computations introduce roundoff errors whose consequences are not easy to determine. Moreover, a rounding error in the condition of a conditional expression can make the actual behavior of the program (the one on a concrete machine) to greatly diverge from its ideal one (the behavior one would expect on genuine real numbers) by executing a different part of the program.

This problem is particularly relevant for safety-critical embedded systems such as the one used in aeronautics (see [18, 34]) that, besides computing with real numbers, do not tolerate any failure. In order to ensure that the execution of a program does not fail due to the bounded memory of the system, the programs written for such systems use a restricted set of programming features. For example, dynamic data structure (e.g. lists) and unbounded behaviors (general recursion or while loops) are not allowed in such programs. Most of the time, real numbers are represented by floating-point numbers but such a representation always introduces a difference between the ideal and the actual behaviors of a program, see [17].

Many solutions to the problem of real number computations have been proposed and will be discussed in section 6. These solutions can be classified in two categories, there are solutions relying on the analysis of errors introduced by a finite representation and solutions using dynamic data structures (polynomial representation or streams) to provide exact computations. However analysis do not always provide sufficiently small error bounds (e.g. in comparison of really close values) to ensure the required safety properties and, as already mentioned, dynamic representations are not allowed in the embedded systems we target.

We propose a solution to the problem of real computation in a fixed-size representation with a program transformation that removes square roots and divisions, extending the work presented in [36]. Eliminating these two operations allows to compute exactly with addition and multiplication using a fixed-point representation of numbers. Indeed, while square root and division can generate infinite sequences of digits, the result of an addition or a multiplication can always be stored on a finite number of bits, and therefore, using a static analysis to determine this number, exactly computed. This transformation enables an exact computation of all the boolean values of a program which protects its control flow from any rounding errors, preventing any divergence between the program actual and ideal behaviors. However the input language of the transformation in [36] was restricted to the core language used to define the algorithms presented in [18, 34] and functions definitions and applications were not handled by the transformation.

This transformation relies not only on the elimination of square roots and divisions in boolean expressions, using a strategy similar to quantifier elimination, but also on a transformation of variable definitions. Figure 1 presents an example of such a transformation, the arrow $\rightarrow$ (or alternatively ↓) stands for our transformation, $A \rightarrow B$ means that the program A is transformed into B.

We transform the definitions in order to take the divisions and square roots out of them without using a variable inlining that would make the size of the produced code explode. The transformed definition defines variables whose values are independent from square root or division computations as shown in Figure 1.

In the previous example, the term $\sigma$ and two substitutions $\sigma_1$ and $\sigma_2$ such that $\sigma_1 = t_1$ and $\sigma_2 = t_2$. In the previous example, the term $(x_1 + \sqrt{x_2})/x_3$ (denoted $t$) is a template of $a + \sqrt{b \cdot c + d}$ and $e \cdot f/(g + h)$ since we have the

\[
t[x_1 \mapsto a; x_2 \mapsto b \cdot c + d; x_3 \mapsto 1] = a + \sqrt{b \cdot c + d}
\]

\[
t[x_1 \mapsto e \cdot f; x_2 \mapsto 0; x_3 \mapsto (g + f)] = e \cdot f/(g + f)
\]

As in the compilation case (see [16, 25]), to ensure the safety level required by the systems this transformation targets, it is required to prove that the transformation preserves the semantics, that is, that the program produced by such a transformation have the same ideal (real numbers) semantics than the input one. In [36] the transformation was certified in the PVS proof assistant, but the transformation of simple variable definition is excluded from this proof. To produce a program for which the semantics equivalence is formally proven, one would have to simply inline the variable and functions in a first step (and prove the correctness of the inlining) before applying the certified transformation. Relying on such inlining makes the size of the transformed program really explode (output programs are more than a hundred times bigger than the inputs) due to the code duplications induced by lifting conditional expression out of boolean expressions. Even using the non-certified variable transformation and a simple function inlining still produces programs that are more than fifteen times bigger than the input.

In this article, we extend the transformation to the program level, allowing function definitions and applications in the input programs to avoid the explosion in the size of the produced code. This transformation of function definitions reuses the constrained anti-unification algorithm that was used for the transformation of variable definitions. Such a transformation eliminates square roots and divisions from the function definitions but also from the arguments of the function calls. Being able to transform directly the functions instead of relying on an inlining pre-process helps to keep some of the structure and some brevity in the output program.

To ensure the semantics preservation, a modular certification mechanism for the complete transformation is also presented in this article. In [36], the semantics preservation has been ensured by defining a certified transformation in the PVS proof assistant [41], however that proof is not entirely complete and relies on the correctness of the anti-unification algorithm that is used to transform variable definitions. This anti-unification algorithm is not only used for the transformation of variable definitions but also, as we present in this article, to transform function definitions and applications. Anti-unification algorithms belong to this particular kind of programs that are difficult to prove to be correct but the correctness of the result is very easy to check; it only requires to apply a substitution to a term and verify that this application corresponds to the input term. This property of the anti-unification problem leads to change the verification paradigm from a certified transformation to a proof-producing transformation, i.e. a transformation that not only returns the transformed program but also a proof that this transformed program has the same semantics.

This proof-producing transformation allows us to formally prove that the output code is semantically equivalent and, unlike the inlining approach, this output code has a reasonable size (i.e., from 2 to 5 times bigger). Moreover, the certification process providing an equivalence lemma for every function definition this approach is much more modular than a global semantics preservation statement. We provide the following contributions:

- A description of the input and output language of the transformation, defining what is a square root and division free program and the semantics preservation property (Section 2)
- Transformation rules that removes square roots and divisions from function definition bodies and calls by realizing a partial inlining (Section 3)
- A global strategy, as a dependency order on the program, to ensure that the application of the transformation rules produces a square root and division free program (Section 4)
- The certification process providing semantics equivalence lemmas. It is implemented as a proof-producing transformation used to transform programs written in Pvs (Section 5)

2. Definition of the Transformation

The transformation presented in [36] aims at removing square roots and division in a program in order to make the compilation of Boolean expressions exact (in particular in conditional expressions) by using a fixed-point representation that allows exact computation with addition, subtraction and multiplication. Of course these operations can not be eliminated from any program, the program $\sqrt{2}$ will still return a rounded value. In this article we extend the language this transformation applies on to function definitions and applications. In [36], an algorithm eliminating square roots and divisions in Boolean expressions built with comparisons between arithmetic expressions is defined. Therefore, as we already explained, we could inline the function definitions and then eliminate the square roots and divisions using the already existing algorithm. However, this process would exponentially increase the size of the produced program and remove the intention of the programmer that introduced those functions for modularity or readability reasons.

Our goal is to have Boolean values completely independent from square roots or divisions computations but functions, whose calls may appear in these expressions, can contain these operations both in their definitions and in the arguments of the calls, e.g.,

\[
\text{l}(x,y) = (x + y) / y \quad \text{in} \quad \text{l}(a + \sqrt{b},c) > d
\]

In this case, how can we transform this function such that $\text{l}(a + \sqrt{b},c) > d$ does not depend on any square root or division? An inlining would transform this program into the following one:

\[
(a + \sqrt{b} + c) / c > d
\]

where square roots and divisions can be eliminated using the procedure on Boolean expressions. But this is not what we are looking for, we want to still have function definitions in the output code however, we have to ensure that the result of such function applications are completely independent from square root and division computations.

In this section, we present an algorithm that does not increase too much the size of the produced code but also preserves the structure of the program. It relies on a partial inlining that allows us to only manipulate sub-expressions that are square roots and divisions free as presented in Figure 2. The variables $f_1$, $f_2$ and $f_3$ being computed without any square root or divisions, we can now use the elimination algorithm for Boolean expressions introduced in [36] to get an equivalent program that does not use any square root or division.

The general idea of this transformation is to delay the computation of square root or division appearing in the definitions until their use in boolean expressions where they can be eliminated. To achieve this goal, one could think of a representation of real numbers using the abstract syntax structure of the arithmetic expres-
sions and only computing the square root and division free subtrees. Then, when such a real appear in a boolean expression, one could transform the expression and eliminate the square root and divisions at run-time before computing the value exactly. Such a representation would easily allow to transform programs with recursion and higher order functions. However, this representation is not suitable for embedded system since the representation is a tree (dynamic) and the transformation program is also a general recursive algorithm that does not terminate in a bounded time. The idea of the anti-unification solution is to find a representation that works in all cases and therefore does not have to be computed at runtime. Moreover, eliminating square roots and divisions sounds interesting because in a straight line program addition and multiplication can be computed exactly. Doing so in a richer language might require to use a dynamic representation of reals and existing solutions such as algebraic numbers might be much more efficient in that context.

Language We now describe the generalization of the previous example. The language the transformation applies on is a typed functional language that contains arithmetic and boolean operators, pairs and projections, variable definitions, conditional expressions and function definitions and applications. We restrict the transformation to programs that are in a certain normal form corresponding to the following mutually recursive definitions. We define different sets: unary expressions $E_u$, expressions $E$, programs without function definitions $P$ and eventually we add the function definitions that have to be defined at top level, this corresponds to the set $P^\top$: 

$E_u ::= X | C | uop E_u | E_u op E_u | fst E_u | snd E_u | X(P)$

$E ::= (E) | E | E_u$

$P ::= \lambda \gamma : \forall \gamma | \gamma P | \gamma$ if $P$ then $P$ else $P | E$

$P^\top ::= \forall \gamma : \forall \gamma | \gamma P | \gamma$

where $C$ is the set of constants (in $\mathbb{R}$ and $\mathbb{B}$), $X$ the set of variables and $\forall$ the set of tuples of variables (e.g., $(x,y)$).

The set of binary operators $op$ contains arithmetic operators

$Abop = \{+, -, \cdot, /\}$, Boolean operators $Bbop = \{\forall, \forall\}$ and comparison operators $Cbop = \{\leq, \geq, >, >=, <, \leq\}$ and the set of unary ones $uop = \{\forall, -\}$. $Type$ is the set of types of tuples of arithmetic and Boolean expressions ($Type ::= \mathbb{R} | \mathbb{B} | Type \times Type$), therefore only first order functions are allowed in the language. This language corresponds to the language used to define the conflict detection and resolution algorithm of the ACCoRD systems as presented in [18, 34].

Type and semantics We now define the type system and the semantics, these are the usual one for such a functional language with call by value. We only give the rules for function definition and application, other cases can be found in [36]. First we extend the set of types with functional types:

$Type_f ::= Type | Type \rightarrow Type$

and provide the corresponding typing rules:

$\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash e : A$

$\Gamma \vdash (f(e)) : B$

$\Gamma \vdash \text{let } x : A \rightarrow B = b ; \text{sc} : C$

the $\oplus$ operator corresponds to the addition of a multi-variable definition in the typing or semantics environment e.g., $Env \oplus ((x, y) : A \times B) = Env, (x : A), (y : B)$. Type inference and semantics fails when this operator fails e.g., $Env \oplus ((x, y) : \mathbb{R})$ fails since the type $\mathbb{R}$ does not correspond to a pair.

Then we define the big step semantics of functions using closures and call by value. The closure $<x, b, E>$ stores the bound variable $x$, the body of the definition $b$ and the current environment $E$ when a function is defined.

$E, (f, <x, b, E>) \vdash \text{sc} \quad \text{sc} \quad \text{v}$

$E \vdash f(x) \rightarrow Fail$

$E \vdash f(\text{e}) \rightarrow Fail$

We define the semantics of an ill-typed program as $Fail$, thus every program has a semantics in every environment. We denote by $\llbracket p \rrbracket_E$ the semantics of $p$ in the environment $E$ (i.e., the $v$ such that $E \vdash \llbracket p \rrbracket = v$).

Output language We now define the set of the output programs. In this set, we want not only the function bodies but also the function arguments to be square root and division free. Square roots and divisions can be eliminated locally in any Boolean expression using the algorithm described in [36] (e.g., $\sqrt{x} > b \rightarrow x > b \cdot b \forall b < 0$).

However in order to make the computation of Boolean expressions independent from square roots and divisions computations we have to ensure that the variables used in these expressions are not computed with these operations and the transformations of variable and function definition is not local. Therefore the transformation of a complete program is decomposed into 2 distinct steps, in the first one we transform the definitions of numerical variables and functions and in the second one we locally eliminate the remaining square roots and divisions in the Boolean expressions.

In order to specify the different steps of this transformation we introduce some intermediate languages that are used during this process. We define different set of terms, depending on where the square root and division operators are allowed:

- $E$ is $E$ where square root and division operators do not appear
- $E_B$ is $E$ where square root and division operators only appear in arguments of comparisons
- $E_N$ is $E$ where square root and division operators do not appear in arguments of comparisons

For example $(\sqrt{3} > x, 3) \in E_B$ but not in $E_N$ and $(2 > x, \sqrt{3})$ is in $E_N$ but not in $E_B$.

Given these different sets of expressions, we define the similar sets of programs:

$P ::= \text{let } X : Type \rightarrow Type = P ; P^\top$

$P_E ::= \text{let } X : Type \rightarrow Type = P ; P^\top$

$P_B ::= \text{let } X : Type \rightarrow Type = P ; P^\top$

$P_N ::= \text{let } X : Type \rightarrow Type = P ; P^\top$
We introduce some notations relative to the substitu-
tion. Substitutions are usually denoted by $\sigma$. Given a substitution $\sigma = [x_1 \mapsto e_1; \ldots; x_n \mapsto e_n]$ and a term $t$, we denote by $t\sigma$ the application of $\sigma$ to $t$, $\varphi(\sigma)$ is the tuple of variables in the do-
main $i.e.,$ $(x_1, \ldots, x_n)$ and $\arg(\sigma)$ is the corresponding image $i.e.,$ $(e_1, \ldots, e_n)$. We assume that a substitution applied on a program handles conflicts with bound variable names.

We now formally define the transformations corresponding to these examples. Both of these transformations rely on the $\{\sqrt{.}\}$-constrained anti-unification of terms.

**Definition 2** (Constrained template). Given a finite set of terms $S$ in $E$ of identical type, a $\{\sqrt{.}\}$-constrained template of $S$ is a term $t$ in $E$ such that:

$$\exists x_1, \ldots, x_n \in \mathcal{X}, \forall s \in S, \exists e_1, \ldots, e_n \in E, t[x_1 \mapsto e_1; \ldots; x_n \mapsto e_n] = s$$

For example $(x \cdot \sqrt{y})/z$ is a constrained template of $a \cdot b \cdot \sqrt{c}$ and $d/(f+e)$.

$$a \cdot b \cdot \sqrt{c} = ((x \cdot \sqrt{y})/z) [x \mapsto a \cdot b; y \mapsto c; z \mapsto 1]$$

$$d/(f+e) = ((x \cdot \sqrt{y})/z) [x \mapsto d; y \mapsto 1; x \mapsto f + e]$$

Such a constrained template always exists and an algorithm computing a template with a small number of square roots and the corresponding substitutions can be found in [37]. Given this algorithm, we now aim at defining a transformation that removes square roots and divisions from the arguments of a function call, indeed in $\mathcal{P}^T$ the function calls have to be made on programs in $\mathcal{P}$, however as mentioned before, the elimination in Boolean expression will be done later. Therefore we want to define a transformation $\text{Elim}_{f, \text{fin}}$ that has the following specification:

**Definition 3** ($\text{Elim}_{f, \text{fin}}$ specification). Given $f \in \mathcal{X}$, $x \in \mathcal{Y}$, $A, B \in \text{Type}$, $b, sc \in \mathcal{P}$, we want the $\text{Elim}_{f, \text{fin}}$ function to compute $x', A', b'$ and $sc'$ such that:

- $\forall Env$, $\llbracket \text{let f x : A \rightarrow B = b; sc}\rrbracket_{Env} \neq \text{Fail} \implies \llbracket \text{let f x : A \rightarrow B = b; sc}\rrbracket_{Env} = \llbracket \text{let f x' : A' \rightarrow B= b'; sc'}\rrbracket_{Env}$

- $\forall f, p, (p) \leq sc' \implies p \in \mathcal{P}_B$

Where $\leq$ is the non-strict partial sub-term relation.

This means that the calls of the function are not made on numerical values computed with square roots or divisions. However, Boolean values are still allowed to contain these operations.

3. Function Definition Transformation

As mentioned previously, given a program that contains function definitions, we want to remove square roots and divisions both from the arguments of the function calls and from the bodies of the function definitions. We present a way to do both of these transformations completely independently. Figure 3 illustrates the elimination of square roots and divisions in the function calls where the term $x_1 + x_2 \sqrt{x_3}$ is a constrained template of $b$ and $c + d\sqrt{e}$, that is a term such that square root and division free substitutions applied to this term produce the terms it anti-unifies, i.e.,:

$$\llbracket x_1 + x_2 \cdot \sqrt{x_3}\rrbracket(x_1 \mapsto c; x_2 \mapsto d; x_3 \mapsto e) = c + d \cdot \sqrt{e}$$

$$\llbracket x_1 + x_2 \cdot \sqrt{x_3}\rrbracket(x_1 \mapsto b; x_2 \mapsto 0; x_3 \mapsto 0) = b$$

This transformation has eliminated all the square roots and divi-
sions that used to appear in the arguments of the calls of function $f$. Then Figure 4 illustrates the elimination from the body of the function where $y_1 + y_2 \cdot \sqrt{y_3} + \sqrt{y_4}$ and $z_1 + z_2 \cdot \sqrt{z_3} + \sqrt{z_4}$ are templates of $3 \cdot (x_1 + x_2 \cdot \sqrt{x_3}) + \sqrt{\gamma}$.

There are no more divisions in the body of the function, there-
fore the result of this new function $f$ (e.g., $y_1, y_2, y_3, y_4$) does not depend on square roots and divisions anymore.

**Notations** We introduce some notations relative to the substitu-
tions. Substitutions are usually denoted by $\sigma$. Given a substitution $\sigma = [x_1 \mapsto e_1; \ldots; x_n \mapsto e_n]$ and a term $t$, we denote by $t\sigma$ the application of $\sigma$ to $t$, $\varphi(\sigma)$ is the tuple of variables in the do-
main $i.e.,$ $(x_1, \ldots, x_n)$ and $\arg(\sigma)$ is the corresponding image $i.e.,$ $(e_1, \ldots, e_n)$. We assume that a substitution applied on a program handles conflicts with bound variable names.

We now formally define the transformations corresponding to these examples. Both of these transformations rely on the $\{\sqrt{.}\}$-constrained anti-unification of terms.

**Definition 1** (Transformation Specification). Given a program $p$ in $\mathcal{P}_T$, the transformation aims at producing a program $p'$ such that:

$$\llbracket p \rrbracket_E = \text{Fail} \implies \llbracket p' \rrbracket_E = \llbracket p \rrbracket_E$$

The transformed program has a control flow which is independent from any square root or division computation (1) and the semantics is preserved for every environment where the input program does not fail (2).

We now present the elimination of square roots and divisions in function definitions and applications using anti-unification.

4. Function Definition Transformation

As mentioned previously, given a program that contains function definitions, we want to remove square roots and divisions both from the arguments of the function calls and from the bodies of the function definitions. We present a way to do both of these transformations completely independently. Figure 3 illustrates the elimination of square roots and divisions in the function calls where the term $x_1 + x_2 \sqrt{x_3}$ is a constrained template of $b$ and $c + d\sqrt{e}$, that is a term such that square root and division free substitutions applied to this term produce the terms it anti-unifies, i.e.,:

$$\llbracket x_1 + x_2 \cdot \sqrt{x_3}\rrbracket(x_1 \mapsto c; x_2 \mapsto d; x_3 \mapsto e) = c + d \cdot \sqrt{e}$$

$$\llbracket x_1 + x_2 \cdot \sqrt{x_3}\rrbracket(x_1 \mapsto b; x_2 \mapsto 0; x_3 \mapsto 0) = b$$

This transformation has eliminated all the square roots and divi-
sions that used to appear in the arguments of the calls of function $f$. Then Figure 4 illustrates the elimination from the body of the function where $y_1 + y_2 \cdot \sqrt{y_3} + \sqrt{y_4}$ and $z_1 + z_2 \cdot \sqrt{z_3} + \sqrt{z_4}$ are templates of $3 \cdot (x_1 + x_2 \cdot \sqrt{x_3}) + \sqrt{\gamma}$.

There are no more divisions in the body of the function, there-
fore the result of this new function $f$ (e.g., $y_1, y_2, y_3, y_4$) does not depend on square roots and divisions anymore.
such a decomposition respects the following equality:
\[
\text{let } f \ (x,y) : B \times R \rightarrow R = \\
\text{if } \sqrt{x} > t \text{ then } (\sqrt{y} > c, d, \sqrt{e}) \text{ else } (\text{False, 1, 1}) > 0
\]

**Scope decomposition**  
Such a transformation relies on a decomposition of the function scope. We have to extract the different function calls of \( f \) assuming there are no nested calls of this function, indeed in this case we will not be able to transform the program as we will see in Section 4.

**Definition 4** (Scope f-decomposition). Given \( f \) a function variable and a program \( p \) in \( P^T \) that does not contain any nested call of \( f \), this program can be decomposed into:

- a scope program part metafunction \( scf \) of type \( P^n \rightarrow P^T \) that represents the structure of \( p \) where the function calls of \( f \) have been abstracted.
- a list of programs \( (c_1, ..., c_n) \) in \( P \) corresponding to the different calls of \( f \).

This decomposition has to respect the following equality:
\[
scf((c_1), ..., (c_n)) = p
\]

For example, the following program:
\[
\text{let } x = f(\text{if } c \text{ then } \sqrt{a} \text{ else } b) \text{ in } f(d/e) + x
\]

admits the following f-decomposition:
\[
scf := (fun \ (y, z) \rightarrow \text{let } x = y \text{ in } z + x)
\]
\[
c_1 := \text{if } c \text{ then } \sqrt{a} \text{ else } b
\]
\[
c_2 := d/e
\]

The next step of the decomposition is the extraction of the final expressions from the different call arguments, corresponding to the different cases of the conditional expressions.

**Definition 5** (P decomposition). A program \( p \) in \( P \) can be decomposed into:

- a program part metafunction \( Pp \) of type \( E^n \rightarrow P \) that represents the different if then else constructors and variable definitions
- a list of final expressions \( e_1, ..., e_m \) in \( E \) corresponding to the different expressions that can be returned by the program.

Such a decomposition respects the following equality:
\[
Pp(e_1, ..., e_m) = p
\]

This decomposition is done by the algorithm in figure 5. The final expressions that correspond to the different cases of the different calls can be anti-unified as introduced in definition 2. It produces a template \( T \) and substitutions of same domain such that \( \forall i \leq m, \forall j \leq m, T \sigma_{i,j} = e_{i,j} \). Therefore given a function variable \( f \), we decompose every program by combining both decomposition from definitions 4 and 5 and the template computation.

**Lemma 1** (Program Decomposition). Every program \( p \) in \( P^T \) can be decomposed in the following form:
\[
scf((Pp_1(T \sigma_{1,1}, ..., T \sigma_{1,m_1})), ..., (Pp_n(T \sigma_{n,1}, ..., T \sigma_{n,m_n})))
\]

In [37] a property of the \( \text{Decompose} \) function is introduced that allows that the template and the program part to commute under certain hypothesis on the variables that are used in the template. Given a program \( p \) we denote by \( FV(p) \) the free variables in \( p \) and by \( BV(p) \) the variables that are bound in \( p \), we also use \( BV \) for program part and scope program part to represent the variable that are bound in such metafunctions \( (BV(Pp_i) = BV'(Pp_i(x_1, ..., x_m))) \).

**Lemma 2** (Template and program part commutativity). Given a term \( T \), substitutions \( \sigma \) and a program part \( Pp \), if \( \text{FV}(T) \cap \text{BV}(Pp) = \emptyset \) then we have the following semantics equality:
\[
Pp(T \sigma_{1,1}, ..., T \sigma_{n,m}) \text{ } \stackrel{\text{seq}}{\Rightarrow} \text{ let } \text{var}(\sigma) = Pp(\text{arg}(\sigma_1), ..., \text{arg}(\sigma_m)) \text{ in } T
\]

For example, the following transformation preserves the semantics:
\[
\text{if } F \text{ then } a = u \text{ in } a \cdot \sqrt{b} / d \text{ else } c \cdot \sqrt[3]{a}
\]
\[
\text{let } (x,y,z) = \\
\text{if } F \text{ then } a = u \text{ in } (a, b, 1) \text{ else } (c, 1, d)
\]
\[
in x \cdot \sqrt[3]{z}
\]

We can extend such commutativity to the function definition level. It enables the square roots and divisions elimination in the function arguments (the expressions used in the substitutions are square root and division free). It relies on a commutativity rule between the template and the scope program part:

**Definition 6** (Function input transformation). Given a function definition \( \text{let } f : A \rightarrow B = b; \text{ sc } \) and the decomposition of \( \text{sc} \) using Lemma 1, we define a rule that inlines the template of the calls in the body of the function definition:
\[
\text{let } f : A \rightarrow B = b; \\
\text{scf}((Pp_1(T \sigma_{1,1}, ..., T \sigma_{1,m_1})), ..., (Pp_n(T \sigma_{n,1}, ..., T \sigma_{n,m_n})))
\]

\[
\text{let } f \text{ var}(\sigma_1) : A' \rightarrow B = b[ x \mapsto T ]; \\
\text{scf}((Pp_1(\text{arg}(\sigma_{1,1}), ..., \text{arg}(\sigma_{1,m_1}))), ..., (Pp_n(\text{arg}(\sigma_{n,1}), ..., \text{arg}(\sigma_{n,m_n}))))
\]

where \( A' \) is the type of the \( \text{arg}(\sigma_{i,j}) \).

In order to prove that this transformation preserves the semantics, we have to enforce some conditions on the variable names to avoid conflicts. The following lemma states the correctness of this transformation rule.

**Lemma 3** (Function input transformation correctness). When the following properties hold:

\[
\begin{align*}
\text{Decompose}(p) &= \\
&\text{ if } p \in E \text{ then } ((fun \ x \rightarrow x), p) \\
&\text{ if } p = \text{let } y = a \text{ in } p' \text{ then } \\
&\quad (pp.ep) := \text{Decompose}(p') \\
&\quad \text{return: } ((fun \ x \rightarrow \text{let } y = a \text{ in } pp(x)), ep) \\
&\text{ if } p = \text{if } B \text{ then } p_1 \text{ else } p_2 \text{ then } \\
&\quad (pp_1.ep_1) := \text{Decompose}(p_1) \\
&\quad (pp_2.ep_2) := \text{Decompose}(p_2) \\
&\quad \text{return: } \\
&\quad ((fun \ (x_1, x_2) \rightarrow \text{if } B \text{ then } pp_1(x_1) \text{ else } pp_2(x_2)), (ep_1, ep_2))
\end{align*}
\]
that do not contain square root or division operations, these
later. Therefore we want to define a transformation

\[ \text{BV} \]

\[ \text{tics}. \]

\[ \text{B} \]

\[ \text{to compute} \]

\[ \text{Definitions} \]

\[ \text{from the arguments of function calls. We now introduce the} \]

\[ \text{ments of the calls of the function} \]

\[ \text{Definition 6 enables the elimination of all square roots and divi-
}\]

\[ \text{sions from the numerical expressions corresponding to the argu-
}\]

\[ \text{ments of the calls of the function} \]

\[ \text{such that:} \]

\[ \text{And} \]

\[ \text{Conditions for the semantics preservation are handled by the} \]

\[ \text{template and substitutions computations. Application of the rule of} \]

\[ \text{Definition 6 enables the elimination of all square roots and divi-
}\]

\[ \text{sions from the numerical expressions corresponding to the argu-
}\]

\[ \text{ments of the calls of the function} \]

\[ \text{such that:} \]

\[ \text{Definition 7 (Elim}} \]

\[ \text{The} \]

\[ \text{function is the composition of} \]

\[ \text{steps:} \]

\[ \text{The decomposition of the scope of the function according to the} \]

\[ \text{Definition 4 with respect of the conditions (Hp), (HT) and (Hx')} \]

\[ \text{introduced in Lemma} \]

\[ \text{Definition} \]

\[ \text{We have defined the elimination of the square roots and divi-
}\]

\[ \text{sions from the arguments of function calls. We now introduce the} \]

\[ \text{transformation of the body of the function definition to remove} \]

\[ \text{square roots and divisions from it since, in} \]

\[ \text{the function bodies} \]

\[ \text{have to be programs in} \]

\[ \text{As in the input transformation the elimination of these operations in} \]

\[ \text{Boolean expressions will be done later. Therefore we want to define a} \]

\[ \text{transformation} \]

\[ \text{that has the following specification:} \]

\[ \text{Definition 8 (Elim}_ \]

\[ \text{Given} \]

\[ \text{for every call of} \]

\[ \text{and for every environment} \]

\[ \text{we have we have:} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{And} \]

\[ \text{This means that the new function can only return numerical val-
}\]

\[ \text{that do not contain square root or division operations, these} \]

\[ \text{these operations are still allowed in Boolean expressions as presented} \]

\[ \text{in Figure}. \]

\[ \text{As in the elimination in function arguments, the trans-
}\]

\[ \text{formation relies on a decomposition using metafunctions and anti-
}\]

\[ \text{unification. This time the body of the function definition,} \]

\[ \text{is decomposed with a program part} \]

\[ \text{and anti-unification, produc-
}\]

\[ \text{ing a template} \]

\[ \text{and a list of substitution} \]

\[ \text{such that:} \]

\[ \text{Given this decomposition we define the following transformation:} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{Figure 6. Square root in Boolean value} \]

\[ \text{Definition 9 (Function output transformation). Given a function} \]

\[ \text{definition let} \]

\[ \text{for every call of} \]

\[ \text{and for every environment} \]

\[ \text{we have we have:} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{let} \]

\[ \text{where} \]

\[ \text{and} \]

\[ \text{and} \]

\[ \text{and} \]

\[ \text{where} \]

\[ \text{Once again we have to enforce some conditions on the variables} \]

\[ \text{names that are used in the template in order to allow the elements} \]

\[ \text{to commute.} \]

\[ \text{Lemma 4 (Function output transformation correctness). When the} \]

\[ \text{following conditions hold:} \]

\[ \text{where} \]

\[ \text{Proof} \]

\[ \text{Proof} \]

\[ \text{is similar to the one of Lemma 3 by induction on the} \]

\[ \text{scope. (Hp) allows the commutation of} \]

\[ \text{and} \]

\[ \text{and} \]

\[ \text{and} \]

\[ \text{where} \]

\[ \text{This rule eliminates the square roots and divisions from the} \]

\[ \text{of the body. It enables us to define the} \]

\[ \text{function that respects the specification introduced in Definition 8:} \]

\[ \text{Definition 10 (Elim}_ \]

\[ \text{The} \]

\[ \text{function is the composition of} \]

\[ \text{steps:} \]

\[ \text{Decompose the body according to the} \]

\[ \text{with respect of the conditions (Hp), (HT), (HT2) and (HF)} \]

\[ \text{Decompose the scope according to the} \]

\[ \text{Apply the rule of Definition} \]

\[ \text{9} \]
The transformations that eliminate square roots and divisions in function arguments and function definition bodies are defined in Definitions 7 and 10. The simple variable case has been defined also with decomposition of the body (with the Decompose function) and template inlining in [36] with the Elim rule:

\[
\text{let } x = P p(e_1, \ldots, e_n) \quad \rightarrow \quad \text{let } \var{\sigma_1} = P p(\text{arg}(\sigma_1), \ldots, \text{arg}(\sigma_n)) \text{ in } p2[x \mapsto T]
\]

Given the two functions, Elim \text{fin} from Definition 7 and Elim \text{fout} from Definition 10 and the simple variable case, we now define a strategy that removes square roots and divisions from all the definitions of a program.

4. The Inlining Order

Application of the rules relative to variable and function definitions is not as straightforward as in the case without functions. We have to find the right order to transform the function and variable definitions in order to be sure that eventually neither the function calls and definitions nor the variable definitions contain square roots or divisions. The transformation Elim \text{fin} depends on the calls of this function and these calls might depend on variables that are defined in the scope of the function definition. Thus the inlining of templates used to transform these variable definitions might create new square roots or divisions in these calls. Given the following program:

\[
\begin{align*}
&\text{let } f x = x/2; \\
&\text{let } y = \sqrt{a} \text{ in } \\
&f(y) > 0
\end{align*}
\]

by following the program subterm order and first eliminating square roots and divisions in \( f \) and then in \( y \) we would get the following program:

\[
\begin{align*}
&\text{let } f x = (x,2); \\
&\text{let } y = a \text{ in } \\
&\text{let } (f_1, f_2) = f(\sqrt{y}) \text{ in } f_1/f_2 > 0
\end{align*}
\]

The argument of a call of \( f \) still contains a square root. This means that when we apply a definition transformation rule (i.e., variable, function input or function output) we have to be sure that all the variables and functions the corresponding template depends on have already been transformed (that are the variables and function in the expressions that are anti-unified). In order to be sure that we apply the rules in the right order we construct a dependency graph. The nodes of the graph represent the different variable and function transformations that have to be completed before eliminating square roots and divisions in Boolean expressions. These transformations correspond to the variable definitions identified by the variable name, plus, for every function \( f \) the transformation of its input, identified by \( f_i \), and output, identified by \( f_o \). We call these identifiers the transformation items. The dependency graph associates to every transformation item all the transformation items depending on the former one.

The dependency in a variable \( x \) bound by a function \( f \) is a dependency in the function input and not the variable itself since all these bound variables are transformed at the same time when using the Elim \text{fin} rule. Moreover since we want a transformation item to represent its definition we only work on programs that only have unique variable definitions (no variable is redefined). Every program can be easily transformed into a program with unique variable definition using variable renaming. Given such a program with unique variable definition, we define the transformation item associated to every variable:

**Definition 11** (Corresponding transformation item). *Given a program \( p \) with unique variable definition, for every variable \( x \in \mathcal{X} \), we define \( x^\top \) the associated transformation item that corresponds to the only occurrence of its definition:*

\[
\begin{align*}
&\text{if } \exists v, b, \text{ sc, let } v = b \text{ in } \text{ sc } \subseteq p \land x \subseteq v \text{ then } x^\top = v \\
&\text{if } \exists f, v, b, \text{ sc, let } f v = b \text{ in } \text{ sc } \subseteq p \land x \subseteq v \text{ then } x^\top = f \\
&\text{if } \exists f, b, \text{ sc, let } x v = b \text{ in } \text{ sc } \subseteq p \text{ then } x^\top = x_o
\end{align*}
\]

Therefore given such a program with unique variable definition we now present the construction of the dependency graph. This graph is represented as list of edges and we compute the graph in a list of contexts that represents the definitions that are depending on the current expression:

**Definition 12** (Definition Context). *The different contexts are:*

- Variable definition context: \( V D(x) \) with \( x \in \mathcal{V} \)
- Function definition context: \( F D(f, x) \) with \( (f, x) \in \mathcal{X} \times \mathcal{V} \)
- Function application context: \( F A(f) \) with \( f \in \mathcal{X} \)
- Boolean context: \( B C \)
1. \( \mathcal{G} := (p, \emptyset)^\forall \), check that \( \mathcal{G} \) is acyclic
2. while \( \mathcal{G} \) is not empty do
   • Choose a root item \( t \)
   • Transform the definition corresponding to \( t \) using the according Elim\textsubscript{let}, Elim\textsubscript{fin}, or Elim\textsubscript{fout} rule
   • \( \mathcal{G} := \{ (y, z) \in \mathcal{G} \mid y \neq t \} \)
3. Transforms the variables that were not in \( p^\forall \), e.g., variables that are only used in Boolean expressions

Figure 8. Definition Transformation Elim\textsubscript{Var}

The dependency graph is then used in the following algorithm to transform all the variable and function definitions that appear in the input program. This algorithm only works on acyclic dependency graphs. The case of cyclic dependency graph is discussed at the end of this section. We define in Figure 8 the Elim\textsubscript{Var} algorithm which transforms all the variable definitions and function calls. In order to state the correctness of this algorithm we introduce a few definitions that represent the evolution of the program during the transformation.

Definition 13. The relation Elim\textsubscript{t}(p, p') where \( p \) and \( p' \) are programs and \( t \) is a transformation item, states that \( p' \) is \( p \) where the transformation corresponding to \( t \) has been applied. We extend the relation to formalize the iteration of such transformations:

\[
\text{Elim}_{\text{t}}(p, p') = \exists p'' \quad \text{Elim}_{\text{t}}(p', p'') \land \text{Elim}_{\text{t}}(p, p'')
\]

The set \( \mathcal{D}(p) \) is the set of transformation items whose corresponding definition body or function arguments are not in \( p^\forall \).

The set \( IT(p) \) is the set of transformation items of the program \( p \).

Given these definitions we can state the following lemma on the iteration of such transformations:

Lemma 5 (Transformation order). Given a program \( p \) and its dependency graph \( p^\forall \), then for all program \( p' \) and list of transformation items \( l \), if

1. \( p' \) is the result of the application of transformations in \( l \) to \( p \)
2. all the transformation items in \( l \) are from the input program (we do not transform definitions produced by the transformations)
3. the order of the variable transformations has followed the dependency graph order

then the definition and function arguments that are not in \( p^\forall \) are the ones of the input program that have not been transformed yet:

\[
\forall p', l,
\begin{align*}
(a) \quad & \text{Elim}_t(p, p') \implies \\
(b) \quad & \forall x \in l, x \in IT(p) \implies \\
(c) \quad & (\forall x, t', (x :: t') \subseteq l \implies \{ y \mid (y, x)^* \in p^\forall \} \subseteq t' \land x \notin t' \} \implies \\
& \mathcal{D}(p') \subseteq IT(p)\backslash \mathcal{IT}(p)
\end{align*}
\]

where \( (y, x)^* \in p^\forall \) denotes the existence of a path from \( y \) to \( x \) in the graph \( p^\forall \).

Proof. The detailed proof of this property can be found in section 4.4 of [37].

This lemma allows us to state the correctness of the global transformation algorithm

Lemma 6 (Elim\textsubscript{Var}, correctness). The Elim\textsubscript{Var} algorithm preserves the semantics and produces a program where all the definition bodies and function arguments are in \( p^\forall \).

Proof. Preservation of the semantics is ensured by the correctness of rules Elim\textsubscript{let}, Elim\textsubscript{fin}, and Elim\textsubscript{fout}. Since we only transform roots, the hypothesis of Proposition 5 is verified: one node becomes a root only when all the variables it depends on have been transformed. Since we transform all the variable definitions and function definitions and calls from the input program, when the algorithm terminates the set \( IT(p)\backslash l \) is empty, therefore all the definition bodies and function calls are in \( p^\forall \).

Cyclic dependency graph The behavior of the algorithm presented in Figure 8 when the graph is acyclic is not specified and in this case the implementation of the algorithm returns an error message. Indeed in such a case we will not be able to eliminate the square roots and divisions from all the function and variable definitions. We already mentioned the fact that we are not able to transform programs that have nested function calls and in fact a program with a cyclic dependency graph is a program that hides a nested function call as in the following program:

\[
\text{let } f \times = 2 \times; \\
\text{let } y = \sqrt{f(b)} \text{ in } f(y) > b
\]

This program has the following dependency graph:

Thus eliminating the square root in \( y \) will introduce one in the call \( f(y) \), then a second elimination will introduce one in the body of \( f \) and then the elimination from the body of \( f \) will create one in \( y \). Following the algorithm in Figure 8 we will not be able to choose a root and if we start from another node then we will end up with a program that still contains square roots or divisions in one of the
Main algorithm Once all the variable and function definitions have been transformed, square roots and divisions are eliminated from the purely numerical expressions that appear in their bodies and calls. The square roots and divisions only remain in expression that are arguments of comparison operators (or in the final expression returned by the program where we can not eliminate them).

Thus the last step of the transformation is to eliminate the square roots and divisions in the boolean expressions. This is done with the elimination algorithm presented in [36], then none of the boolean expressions of the program depends on any square root or division computation and, assuming addition, multiplication and subtraction can be exactly computed, the control flow of the program does not depend on any rounding errors and thus the actual behavior of the program will not diverge from the abstract one. In the following section, we will see how this transformation is applied to PVS programs and how we can formally certify that the semantics of the output is equivalent to the one of the input.

5. A Proof-Producing Transformation

The transformation algorithm with functions has been implemented in OCaml and we aim at using this algorithm to transform the programs for aeronautics, however this OCaml implementation is not certified. These programs for aeronautics are written and proved correct in PVS and then extracted to real embedded code. The subset of PVS used to write these programs exactly matches the language P ⊤ , i.e., they only use arithmetic and Boolean operators, projection, variable and function definitions and conditional expressions. Thus we can use the P ⊤ language as a deep embedding of PVS in OCaml to transform the PVS programs into square root and division free ones. However, we also want to prove the semantics equivalence between the input and output programs.

Certifying Anti-unification The OCaml implementation is not certified, the main reason is that the anti-unification algorithm that is used to compute the template for the transformation of definition is quite complex and optimized to minimize the size of the template that has to be inlined. However, if proving the correctness of this anti-unification algorithm is out of scope as of today, it is quite simple to check that the result of an anti-unification algorithm satisfies its specification e.g., that the input terms are really instances of the template with the given substitutions.

Moreover, if the complete transformation algorithm has not been proven correct, a formalization of the part of the transformation that is the elimination of square roots and divisions in boolean expressions exists in PVS and it has also been turned into a PVS strategy, namely elim-sqrt, see [38]. This strategy allows the elimination of square roots and divisions in boolean expressions based on arithmetic expressions, in particular the equality. Therefore, given a set of arithmetic expressions e1, ..., en and the template T and substitutions σ1, ..., σn computed by the anti-unification algorithm, this strategy allows us to eliminate square roots and divisions from all the equalities Tσi = ti. The equalities produced by this strategy only rely on additions, multiplications and subtraction and this is a decidable problem using quantifier elimination on real closed fields (see [45]). The research of proofs of equality of arithmetic terms is handled in PVS by the grind-reals strategy. Therefore checking that the a term is a template of another term can be automatically done in PVS. We will rely on this proof strategies to prove that the output program produced by the OCaml implementation has effectively the same semantics as the input one.

A Proof-Producing Transformation The main difference between the P ⊤ language and the PVS programs we target is that in P ⊤ functions and variable definitions always have a scope. This is not the case for the PVS programs that can only be a list of definitions, therefore proving the correctness of the transformation of such programs as a global semantics preservation (as stated in Definition 1) is not possible anymore.

There is one case where the equivalence between the input program and the output program can be quite easy to state. This case is when the specification’s last definition is a Boolean value or a function returning Boolean values. Indeed, if the transformation changes the type when we compute with numerical sub-expressions, the Boolean ones are not changed since square roots and divisions can be eliminated locally. If the last definition of the targeted PVS program is a function then it is never called and thus the input type does not change either as shown by the example in figure 10 thus we could state that the transformation preserves this semantics. However transforming the functions that return numerical values changes the types and thus the semantics can not be directly compared. Moreover, when these functions are used into other functions the relation between the definitions of the output program and the ones of the input one are completely lost.

The idea to certify the correctness of the transformation is to provide a specific correctness lemma that specify the behavior of each definition of the output program with respect to its corresponding definition in the input one. Moreover since the target language of the transformation is PVS we can embed these correctness lemmas directly in the transformed program.

To do so, we decided to use the sub-typing features of PVS to encode this correctness lemmas as a typing predicate of the transformed functions. In fact this predicate only represents the templates that have been used to transform the body and the calls of the function. In order to realize such a transformation, in the OCaml implementation, we add a sub-typing predicate field to the function definition constructor. We denote by \{ x : T | P(x) \} where x ∈ V the type of the elements x of type T that verify P(x), e.g., \{ x : \mathbb{R} | x ≥ 0 \} is the set of non-negative real numbers. Such a type exactly corresponds to the PVS subtype \{ x : T | P(x) \}.

Using this predicate, we are able to transform a function and specify the behavior of the transformed function using the input function, an example of such a transformation is presented in figure 11. Generalizing this example, we can give the new transformation rules for the function input and output. Figures 12 and 13 extends

\begin{figure}
\centering
\begin{verbatim}
f_test(x,y : real, z : nnreal) : bool = LET sq = 2 * y * sqrt(z) IN x + sq > 0
\end{verbatim}
\caption{Boolean function transformation}
\end{figure}
\[ f(t : \text{real}) : \text{real} = \frac{(t + 1)}{(t - 1)} \]

\[ \text{res} : \text{bool} = f(a + b + \sqrt{c}) > 0 \]

\[ f_e(t_1, t_2 : \text{real}) : \text{real|} \]

\[ \begin{align*}
& \{ f_{n_1}, f_{n_2}, f_{d_1}, f_{d_2} : \text{real} | \\
& (f_{n_1} + f_{n_2} \cdot \sqrt{c}) / (f_{d_1} + f_{d_2} \cdot \sqrt{c}) \\
& = f(t_1 + t_2 \cdot \sqrt{c}) \} = \\
& (1 + t_1, t_2, -1 + t_1, t_2) \]
\]

\[ \text{res} : \text{bool} = \]

\[ \text{LET} (f_{n_1}, f_{n_2}, f_{d_1}, f_{d_2}) = \\
\begin{align*}
& f_e(x, y + 1) \\
& f_e(a, b) \\
& \text{IN} \ldots
\end{align*} \]

\[ \text{Figure 11. Output specification} \]

\[ \text{let } f_x : A \rightarrow B = \]

\[ \text{b;} \]

\[ \text{scf}(f(P_{P_1}(T_{in}\sigma_{1,1}, \ldots, T_{in}\sigma_{m,n})), \ldots, f(P_{P_n}(T_{in}\sigma_{1,1}, \ldots, T_{in}\sigma_{m,n}))) \]

\[ \downarrow \]

\[ \text{let } f_e x' : A' \rightarrow \{ y : B | y = f(T_{in}) \} = \]

\[ b[x \mapsto T_{in}]; \]

\[ \text{scf}(f_e(P_{P_1}(\text{arg}\sigma_{1,1}), \ldots, \text{arg}\sigma_{1,m})), \ldots, f_e(P_{P_n}(\text{arg}\sigma_{1,1}, \ldots, \text{arg}\sigma_{1,m}))) \]

\[ \text{where } A' \text{ is the type of the } \text{arg}\sigma_{1,1} \text{ and } x' = \text{var}\sigma_{1,1} \]

\[ \text{Figure 12. Output specification with subtype} \]

\[ \text{let } f_e x : A \rightarrow \{ y : B | y = f(T_{in}) \} = \]

\[ P_{P_1}(T_{out}\sigma_{1,1}, \ldots, T_{out}\sigma_{n}); \]

\[ \text{scf}(f(a_1), \ldots, f(a_m)) \]

\[ \downarrow \]

\[ \text{let } f_e x : A \rightarrow \{ \text{var}\sigma : B' | T_{out} = f(T_{in}) \} = \]

\[ P_{P}(\text{arg}\sigma), \ldots, \text{arg}\sigma); \]

\[ \text{scf}(\text{let } x' = f_e(a_1) \text{ in } T_{out}, \ldots, \text{let } x' = f_e(a_1) \text{ in } T_{out}) \]

\[ \text{where } B' \text{ is the type of the } \text{arg}\sigma \text{ and } x' = \text{var}\sigma. \]

\[ \text{Figure 13. Function output transformation with subtype} \]

\[ f(x_1, y_1 : \text{posreal}) : \text{real} = x_1/y_1 \]

\[ g(t : \text{bool}, x, y : \text{posreal}) : \text{real} = \]

\[ \text{IF } t \text{ THEN } f(x, (y + 1)) \text{ ELSE } \sqrt{x} + y \text{ ENDIF} \]

\[ \downarrow \]

\[ f_e(x_1, y_1 : \text{real}) = \]

\[ \{ f_{n, f_d} : \text{real} | f_{n}/f_{d} = f((x_1, y_1)) \} = \\
\begin{align*}
& (x_1, y_1) \\
& \text{IF } t \text{ THEN} \\
& \text{LET } (f_{n}, f_{d}) = f_e((x, y + 1)) \\
& \text{IN } (f_{n}, 0, f_{d}, 0) \]

\[ \text{ELSE } (y, 1, 1, x) \]

\[ \text{ENDIF} \]

\[ \text{Figure 14. Transformation example} \]

\[ g_e.TCC2 : \text{OBLLIGATION} \]

\[ \text{FORALL} (t : \text{bool}, x, y : \text{real}) : \\
\begin{align*}
& t \text{ IMPLIES} \\
& (\text{FORALL} (f_{n} : \text{real}, f_{d} : \text{real}): \\
& \quad f_{d} = f_e(x, y + 1)^2 \text{ AND} \\
& \quad f_{n} = f_e(x, y + 1) \text{ IMPLIES} \\
& \quad (f_{n} + 0 * \sqrt{0}) / f_{d} = g(t, x, y); \\
\end{align*} \]

\[ g_e.TCC3 : \text{OBLLIGATION} \]

\[ \text{FORALL} (t : \text{bool}, x, y : \text{real}) : \\
\begin{align*}
& \text{NOT } t \text{ IMPLIES} \\
& \quad (y + 1 * \sqrt{x}) / 1 = g(t, x, y); \\
\end{align*} \]

\[ \text{Figure 15. TCC generation} \]

In figure 14. The different cases corresponding to the conditional constructor (if then else) are already decomposed, thus the only predicates to prove are equalities between arithmetic expressions. Yet, the expression comparison does not involve the common template computation and therefore these equalities corresponds to the transformation that have been proved correct in PVS. Therefore, after using the type predicate of the other functions (e.g. \( f_e \)) the current function is depending on, we can use the \((\text{elim-sqrt})\) strategy presented in [38] to prove such equalities. The \( g_e.TCC2 \) obligation requires to prove that:

\[ \quad (f_{n} + 0 * \sqrt{0}) / f_{d} = g(t, x, y) \]

The proof of this obligation is done by the following strategy:

1. By unfolding \( g \) knowing that \( t \) is true, the new goal is the following formula:

\[ \quad (f_{n} + 0 * \sqrt{0}) / f_{d} = f(x, 1 + y) \]

2. Then by adding the type predicate of \( f_e \) we have the following hypothesis

\[ \quad f_{n} / f_{d} = f(x, 1 + y) \]

3. A simple simplification using the rules of arithmetic finishes the proof

This predicate is quite simple and PVS solves it directly using for example the \((\text{assert})\) strategy. However, more complicated predicates require to invoke the subtyping of many other functions and use the \((\text{elim-sqrt})\) strategy, in particular when square roots are involved in Boolean expressions. Thus, in figure 16, we present the
general scheme of the proofs of the correctness relations between the input and the output functions.

I. Unfold the input function that corresponds to the current TCC
II. Introduce all the typing predicates corresponding to the functions that are called in the new expression
III. Prove the Boolean equivalence to reduce the unfolded input function to the case corresponding to the current TCC of the output one
IV. Prove the expression equality with either native arithmetic strategies or the (elim-sqrt) one for more complicated cases

---

Figure 16. TCC proof scheme

Therefore, even if the OCaml transformation is not proven correct in PVS, we are still able to prove the equivalence between the input and the output programs. These equivalence proofs can be done quite easily by using both the powerful type checking condition generation in PVS and the (elim-sqrt) strategy, that corresponds to the elimination of Boolean expressions.

This proof-producing program transformation is able to transform PVS programs from the ACCoRD system for aeronautics and to certify that the transformed program has a semantics equivalent to the input program. The tool implementation and transformation of a complete conflict detection algorithm from this system, namely ed2d, are presented in [39]. For programs from the ACCoRD system, the size of the transformed programmed is usually between 1.5 and 4 times (from 2-3kB to 3-10) the size of the input where the output programs also contain the equivalence lemmas embedded in the types.

6. Related Work

The differences between the concrete implementations of real numbers and the expected abstract behavior has been widely studied. Most implementations use a finite representations for real numbers, in particular the floating-point one, and try to bound the errors of such computations or to ensure some stability and continuity on the program behavior. Some dynamic representations have also been developed to perform exact computation, however due to the limitation of safety critical embedded systems such as bounded memory and computation time, these representations are not suitable for use in the aeronautics systems we target.

Finite representations The most commonly used representation of real numbers is the floating-point representation on 32 or 64 bits [23]. The floating-point number semantics has been widely studied and many program analysis have been developed to ensure the numerical stability of programs using this representation [31, 33].

Abstract interpretation can be used to predict overflow or to bound the roundoff introduced by the representation [1, 9, 19, 20, 29]. A program transformation rearranging numerical expressions to improve the accuracy of floating-point number computations has also been developed by Martel [26, 27].

The problem of diverging behavior due to roundoff in conditional statements has been studied by Chaudhuri in [7, 8] by presenting a continuity analysis. In a continuous program, if a different branch than the ideal semantics is taken due to a roundoff error then the error between the two semantics remains small. Continuity being a property of the program and not all programs are continuous.

Interval arithmetic [32] has also been used both for analysis and as representation for real number computations [13]. Darulova and Kuncak also presented a programming model and a compilation scheme that allows to measure and bound the uncertainties of computations [12]. However exact computation is not possible using such finite representations and the different analysis can not always ensure the stability of the programs, in particular when values on both sides of a comparison are close.

Another approach with the floating-point numbers is to directly prove the correctness of the program without assuming it computes with real numbers but on the representation that is used. Floating-point number semantics has been formalized in many proof assistants such as PVS [4, 30], Hol [6], Coq [14, 28], Hol Light [21] and these formalization have been used to prove the correctness of some programs [3, 22]. However these kinds of proof tend to be extremely tedious since even some simple properties such as the associativity do not hold on these representations, therefore proving the correctness of complex algorithms for aeronautics seems to be out of this scope.

Dynamic representations Some dynamic representations have been developed to perform exact computation with real numbers or some subsets of them. In 1980, Wiedmer studied the computation over infinite objects [47] and introduced a representation with infinite decimal fraction. Then Boehm and Cartwright [2] both extended this representation as a sequence of fraction and introduced a representation using lazy evaluation of the digits representing the real number. Different constructions of real numbers have then been introduced, with redundant representation of continued fractions [46] or with functional representation and lazy evaluation [15, 44]. Some representations have even been formalized in the Coq system, a constructive construction of the real number field is presented in [24, 40] and a construction of the algebraic numbers [5] is formalized in [10, 11]. However these representations do not provide bounds on the required memory or computation time which makes them not suitable for safety critical embedded systems.

Conclusion

In this article, we have presented how the constrained anti-unification introduced in [37] is used to handle function definitions and applications in a program transformation that removes square roots and divisions in straight line programs. This transformation realizes a partial inlining in the sense that the corresponding variables and function calls are replaced by a particular term. This term, that is used to replace these variables and calls is a template of the different expressions that can be returned by the body of the definition and therefore it is most of the time much more smaller than the effective body. By using such partial inlining we are able to not only maintain the size of the transformed program into reasonable bounds but also to preserves the structure of the input program.

This structure preservation allows us to define a correspondence between the elements defined in the input program and the output one. We use this correspondence to state the semantics equivalence between the input and the output programs by stating, for every function in the input program, a relation between the result of this input function and the result of the output one. These relations are specified in the type of the transformed functions and they can easily been proven since the proof mainly relies on a verification of the result of the anti-unification algorithm. We provide a proof scheme to prove it in PVS.

Therefore even if the transformation is not certified, we are able to reach the level of safety required by the systems we target by providing a formal proof of the semantics preservation between the input and output programs.

Acknowledgment

I would like to thank César Muñoz and Gilles Dowek for their ideas on this project and Olivier Danvy and the anonymous reviewers for their very useful comments. This work was supported by the Assurance of Flight Critical System’s project of NASA’s Aviation Safety
References


