École Doctorale de l'École Polytechnique

INRIA

THÈSE DE DOCTORAT

Présentée par

Pierre NERON

Pour obtenir le grade de

DOCTEUR de l'ÉCOLE POLYTECHNIQUE

Spécialité : **Informatique**

A Quest for Exactness: Program Transformation for Reliable Real Numbers

---

Directeurs de thèse:

M. Gilles DOWEK	Directeur de Recherche, INRIA
M. César MUÑOZ	Chercheur, NASA

Rapporteurs:

M. John HARRISON	Chercheur, Intel
Mme. Hélène KIRCHNER	Directeur de Recherche, INRIA

Examinateurs:

M. Yves BERTOT	Directeur de Recherche, INRIA
Mme. Sandrine BLAZY	Professeur, Université Rennes 1
M. David LESTER	Lecturer, University of Manchester
M. David MONNIAUX	Directeur de Recherche, CNRS

A Quest for Exactness: Program Transformation for Reliable Real Numbers

Pierre Néron

Remerciements

Il est de coutume d'entamer un mémoire de thèse par les remerciements; et c'est avec une grande joie que je me plie à cet exercice tant la présence et le soutient des personnes qui m'ont accompagné depuis de plus ou moins longues années me sont précieux.

Tout d'abord je tiens à remercier Gilles sans qui je ne serais surement pas ici, pour m'avoir permis de découvrir la logique, pour m'avoir encadré durant mes années à l'X, pour ses conseils d'orientation, de mon stage en Suède à mon prochain départ à Delft, pour sa disponibilité dont j'ai pu profiter voire abuser, pour les nombreux restaurants autour de la place d'Italie, pour ses encouragements et son enthousiasme, pour tout les conseils et brillantes idées dont il m'a fait part et pour sans doute une infinité d'autres raisons.

Je remercie César pour son accueil toujours très sympathique à Hampton, son enthousiasme pour ce projet et ses multiples propositions d'extensions et d'améliorations.

Je remercie également tout particulièrement Catherine Dubois, avec qui il est toujours très agréable de discuter, pour son animation de l'équipe et pour avoir toujours accepté le travail supplémentaire que je lui fournissais, en particuler la relecture en profondeur de cette thèse.

Je remercie Hélène Kirchner et John Harrison d'avoir accepté de rapporter cette thèse, pour leurs commentaires et leurs suggestions. Merci à Sandrine Blazy, Yves Bertot, David Lester et David Monniaux d'avoir accepté de composer le jury de ma thèse.

Durant ces dernieres années, j'ai également partagé de très bon moments et eu de passionnantes discussions dans les differentes équipes que j'ai fréquentées, au LIX avec Assia, Benjamin, Chantal, Cyril, Denis, Mathieu et Victor, puis a l'INRIA avec Alejandro, Ali, Benoit, Cécilia, David, Frederic, Guillaume, Hugo, Kailang, Melanie, Olivier, Pierre, Pierre-Nicolas, Quentin, Raphaël, Ronan et Simon. Je remercie également Raphaël pour son enthousiasme durant son stage avec moi ainsi qu'Hélène et Marine toujours promptes à rendre service. J'ai eu aussi l'occasion de rencontrer des personnes passionnantes au grès des conférences, visites ou séminaires, je pense en particulier à Olivier Danvy, Xavier Leroy, Nachum Dershowitz qui m'ont donné de précieux conseils et encouragé; ainsi qu'aux gens de la NASA, Alwyn, Anthony, Heber et Natasha qui m'ont fait découvrir Hampton et sa région.

J'ai également une pensée pour les personnes qui ont accompagné mes premiers pas dans la recherche, Arnaud, Thierry et Guilhem en Suède et Aline, Bruno, Boutheina, J-H, Julien, Huy, Laurent, Maria chez Gemalto, le tout dans la bonne humeur.

Bien entendu il faut aussi savoir décompresser et j'ai toujours eu l'occasion de le faire en la meilleure des compagnies avec une multitude d'amis, toujours présents pour toutes sortes de traquenards et autres activités pedagogiques, Régis, Alice, Yannick, Charlotte, Quentin, Julie, Thibault, Lucille, Julien, Faustine, Max, Chloé, Edouard, J-C, Pierre, Olivier, Alexander, Alexandre, David, Claire, Clement, Tristan, Solenne, Leo, Bogdan, Pimousse, Jeanne, Piste, Julien, Hugo, Thomas et tout ceux que j'oublie.

Je remercie Céline pour avoir supporté mon rythme décousu de thèsard et mes nombreuses absences, mais pas que.

Enfin je remercie ma famille, et en particulier mes parents, qui m'ont toujours poussé et soutenu afin que tout se passe dans des conditions idéales et que je puisse faire tout ce dont j'avais envie, ainsi que mon frère et Amélie pour leur joie de vivre.

There is nothing (right well beloved Students in the Mathematickes) that is so troublesome to Mathematicall practice, not that doth more molest and hinder Calculators, then the Multiplications, Divisions, square and cubical Extraction of great numbers, which besides the tedious expence of time, are for the most part subject to many slippery errors.

John NAPIER, *1614*

INTRODUCTION

COMPUTERS HAVE BEEN OF CRUCIAL IMPORTANCE in the realization of Jules Verne's dream on July 21th 1969. For two hundred years, machines have gradually replaced men for many complex tasks, from the assembly line of the Ford T to unmanned trains and aircrafts and, nowadays, computer programs are the heart of most of these systems. These programs have surpassed humans in many fields, they can be more reliable since they are more deterministic, subject neither to emotions nor fatigue, they can be more precise, if you want to compute the n-th digit of the number π, they can be faster, since they react in a fraction of a second under the influence of any event. However, the problem of software safety emerges from the increasing complexity of these systems, since one needs to ensure they effectively do what they are supposed to. And when the problem is the correctness of computations over real numbers, many troubles arise.

The notion of real number is firmly related with the notion of infinity, which is not compatible with the finiteness of computers memory. This limitation has been overcome in different ways. Since the introduction of computable numbers by Alan Turing in 1936 [Tur36], many representations of real and computable numbers have been studied. The most common way to deal with real numbers in programming languages is to use the floating point numbers as described in the IEEE 754 standard [IEE85]. This standard defines a representation of numbers with a sequence of 32 or 64 bits, the $sign$, the $exponent$ and the $mantissa$ ($fraction$) representing $(-1)^{sign} \times 2^{exponent-bias} \times 1.mantissa$. For example, the 64 bits representation includes one bit for the sign, eleven for the exponent and fifty-two for the mantissa. However, this standard only represents a finite number of real numbers and therefore many rounding issues arise. In particular, none of the usual operations is always exact [Gol91, MBdD+10, Mon08] and therefore the result of the computation of an arithmetic expression with floating point numbers may differ from the value of this expression on real numbers. For example, the following assertions are true on the floating point numbers:

$$\mathsf{sqrt\ 2.0\ *\ sqrt\ 2.0} > 2.0 \qquad 0.2 + 0.1 > 0.15 + 0.15 \qquad \mathsf{1.0\ /\ 3.0} = .333333333333333315$$

Therefore many techniques have been developed to ensure the reliability of programs using floating point numbers. Static analysis techniques have also been developed to

handle the rounding errors introduced by floating point numbers [GP11, GMP02], it can be done by abstract interpretation [GMP02, Min04] using, for example, polyhedra domains [CMC08]. Interval arithmetic [Moo95, DMM05] is also widely used to prove the stability of programs using floating points numbers. Moreover, the floating point arithmetic has been specified in many proof assistants such as PVS [BM06, Min95], COQ [DRT01], HOL [Har95a, CM95] and HOL Light [Har97], it enables us to prove properties on the floating point implementations instead of the axiomatized real numbers. However proving properties on floating point numbers specifications tends to be quite troublesome since many of the usual properties of the real numbers (*e.g.*, associativity or distributivity) and thus many of the theorems commonly known do not hold anymore. Other representations such as the fixed point numbers [Obe07] have been used previously but they are not as efficient as the floating point one and have not been as deeply studied.

Using a fixed size representation for real numbers always enforces the use of rounding and thus exact computation is out of scope. However, 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 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 [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 [Coh12a, Coh12b].

There is at least one main reason why computations with real numbers have been so thoroughly studied. Real numbers are used to describe the physical world and many systems, namely *cyber-physical systems*, are used to control physical entities. From cars to airplanes, from medical robots to GPS chips, human develops thousands of such cyber-physical systems. Moreover, many of these systems are embedded and require a high level of safety since any failure may lead to dramatic consequences. Methods to ensure the safety of such *safety-critical embedded systems* has been widely studied and efficient tools have been developed for their analysis or development [BCC+03, CKK+12, BBF+00] but these systems do not provide exact computation mechanism.

In this thesis, we address the problem of exact computation with real numbers in safety-critical embedded systems. In such a setting, none of the exact representations of real numbers discussed above is suitable, because all of them require an unbounded amount of memory. Typical examples of embedded software using real numbers are implementing conflict detection and resolution algorithms for aircraft navigation [NMD12, MBMD09]. Not only these programs use solid geometry and therefore computes with real numbers but they also require to be executed as embedded systems. And such systems have constraints to ensure that the programs do not fail due to lack of memory. Yet, all the exact computation techniques we presented before, using either arbitrary precision, lazy evaluation or algebraic numbers, use dynamic data structures and may require an unbounded amount of memory.

Contributions

In this thesis, we investigate a solution to the problem of computation with real numbers based on program transformation. Program transformations may be used to improve the efficiency or safety of programs [PS83] or program analysis [DD03]. One main example regarding the correctness of computation with real number is a program transformation presented in [Mar07, Mar09] that improves the accuracy of computation over floating point numbers, limiting the rounding errors. However the exactness of the computation is still out of scope.

The transformation we propose aims at removing square root and division operations from straight line programs (*i.e.,* programs with no loops), such as those used in aeronautics, in order to allow exact computation over real numbers with the addition, subtraction and multiplication operations. These exact operations can be performed in embedded programs since static analysis allows us to predict the memory required for exact computation using a fixed point representation. This transformation does not allow to compute a real number with an arbitrary precision (the program $\mathsf{sqrt}(2)$ will still return a rounded value of $\sqrt{2}$), however it allows the system to compute exactly Boolean expressions that are built with comparisons between arithmetic expressions. Computing exactly Boolean expressions protects the control flow of the program from any rounding errors. This prevents the program effective behavior to diverge completely from its expected behavior, *i.e.,* the one assuming the numbers are genuine real numbers. Therefore the programs produced by our transformation are somehow continuous, the effective returned value being, in the worst case, a rounding of the expected one and, if the program returns a Boolean value, then this value is exact.

This transformation algorithm relies on two fundamental algorithms. The first is a particular case of quantifier elimination on real closed fields, it eliminates square roots and divisions in Boolean expressions. The second solves a specific anti-unification problem that we called *constrained anti-unification* in order to reduce the size of the produced code. This anti-unification algorithm uses the axioms of a theory of the arithmetic and a directed acyclic graph representation in order to compute common template that allow us to optimize the size of the produced code. The constrained anti-unification algorithm is also used to extend the transformation to a richer language allowing function definitions.

In order to still ensure the high level of safety required by the programs we are willing to transform, we also proved the correctness of this transformation in the PVS proof assistant. Indeed formal proof assistants enable the higher levels of safety and security for programs. They are used to prove properties about the behavior of these programs and ensure the correctness of such proofs more reliably than any human certification. Therefore we used the PVS proof assistant to show that not only the algorithm we presented effectively eliminates square roots and divisions but it also preserves the semantics of the program we transform. It allows us to ensure that the behavior of the transformed program is exactly the same as the expected behavior of the input program. Therefore, all the properties satisfied by this input program still hold on the transformed one.

While the complete algorithm is not entirely proven in the PVS proof assistant, its correctness only depends on that of the anti-unification algorithm. This proof is also sufficient to built a proof strategy that eliminates square roots and divisions in the formulas

used by the PVS proof checker. Moreover, we provide in our transformation scheme a mechanism to generate the correctness lemmas that state the semantics equivalences between input and output programs, these lemmas being quite easy to prove using the proof strategy we defined. In Chapter 8 we will present how our algorithm has been able to transform a complete PVS specification of a conflict detection algorithm for air traffic management introduced.

Organization of this thesis

This thesis is organized in two parts. The first introduces the theoretical work of this thesis, the second presents more practical issues.

Part I introduces the definitions of some properties of the algorithms that have been developed in this thesis to define the program transformation.

Chapter 1 presents the elimination of square roots and divisions in Boolean expressions in two different ways. The first transforms these operations elimination problem into a quantifier elimination problem while the second method eliminates these two operations directly in order to minimize the size of the transformed formula.

Chapter 2 presents a particular anti-unification problem that we define in this thesis called *constrained anti-unification*. It also presents an algorithm that solves this problem in a theory of arithmetic using directed acyclic graphs.

Chapter 3 presents the main program transformation algorithm. This algorithm transforms straight line programs (programs without loop) into equivalent ones where square roots and divisions have been eliminated from the Boolean computations using the algorithms introduced in Chapter 2 and 1. This work have been published in [Ner12].

Chapter 4 presents an extension of the algorithm introduced in Chapter 3. This extended transformation now handles programs using function definitions and transforms them directly, improving the size and the shape of the produced program.

Part II introduces the implementation of the transformation both as a PVS specification and as an OCaml program and some concrete applications.

Chapter 5 the PVS specification of the transformation algorithm introduced in Chapter 3. This specification proves that this transformation preserves the semantics of the programs it transforms, assuming the anti-unification is correct.

Chapter 6 presents the OCaml implementation of the anti-unification algorithm used to transform programs. It also defines the implementation of the transformation algorithm extended to program with function definition described in Chapter 4.

Chapter 7 presents the interfaces that have been implemented to transform real programs. These are a PVS strategy to transform goals in a proof context and the transformation of PVS specifications with generation of the correctness predicates. Work on the PVS strategy is going to be published [Ner13].

Chapter 8 presents the transformation of a PVS program for conflict detection called cd2d implemented for the ACCoRD system for aeronautics.

Contents

Introduction **1**

Contents **5**

I Transformation Algorithms **9**

1 Boolean Expressions **11**
1.1 Using Quantifier Elimination 12
1.2 Arithmetic expression normal form 13
1.3 Division Elimination 15
1.4 Square Root Elimination 16
1.5 Complexity and Examples 18

2 Constrained Anti-Unification **19**
2.1 Definition of the Constrained Anti-Unification 20
2.2 Anti-Unification Modulo an Equational Theory 22
2.2.1 Neutral Elements 23
2.2.2 The Switch Operator 23
2.2.3 Function Commutation and Normal Forms 25
2.3 Anti-Unification on Dag-like Terms 26
2.3.1 Dag Representation 26
2.3.2 Dag Constrained Anti-Unification 28
2.4 $\sqrt{\ }$ and / Anti-Unification 31
2.4.1 Theory of Arithmetic 31
2.4.2 Dag Representation 32
2.4.3 $\{\sqrt{\ }, /\}$-Anti-Unification 34
2.4.4 Dag Extension 36

3 Transformation of Programs **41**
3.1 Language . . . 41
3.2 Program subtypes . . . 44
3.2.1 Normalized language . . . 44
3.2.2 Target language . . . 45
3.3 Specification of the Transformation . . . 47
3.4 Reliable Computations over Real Numbers . . . 47
3.4.1 Exact Computation . . . 47
3.4.2 Program $\sqrt{}$ and / Continuity . . . 48
3.5 Transforming Programs . . . 51
3.5.1 Orders on Programs . . . 51
3.5.2 Program Normal Form P . . . 52
3.6 Boolean Expression Transformation . . . 53
3.7 Variable Definition Transformation . . . 54
3.7.1 Specification of Variable Definition Transformation . . . 55
3.7.2 Variable Definition Transformation . . . 56
3.7.3 Single Expression Decomposition . . . 58
3.7.4 Multiple Expression Decomposition . . . 59
3.8 Main Transformation . . . 61

4 Transforming Functions and Function Calls **63**
4.1 Language Extension . . . 64
4.2 Function Definition Transformation . . . 67
4.2.1 Function input transformation . . . 67
4.2.2 Function output transformation . . . 70
4.3 Dependency Graph . . . 72
4.4 Order for Variable Definition Transformation . . . 76
4.4.1 Variable inlining consequences . . . 77
4.4.2 Definition transformation iteration . . . 79
4.5 Towards Acyclic Graphs and Loops . . . 85
4.5.1 Function duplication . . . 85
4.5.2 Template fixpoint . . . 86
4.5.3 The Division Case . . . 87

II Implementations and Applications **89**

5 Formal PVS Proof **91**
5.1 The PVS Proof Assistant . . . 91
5.2 PVS Formalization . . . 93
5.3 Program normal form . . . 95
5.3.1 Substitution . . . 95
5.3.2 Program Normalization . . . 96
5.4 $Elim_{\mathbb{B}}$ proof . . . 97
5.4.1 Head Division Form . . . 98
5.4.2 Division Elimination . . . 99

5.4.3 Square Root Factorization . . . 99
5.4.4 Square root elimination . . . 101
5.5 Variable Definition Transformation . . . 102
5.5.1 Template . . . 102
5.5.2 Decomposition . . . 104
5.6 Main elimination . . . 106

6 OCaml Implementation **107**
6.1 Simplification . . . 108
6.2 Anti-unification Algorithm . . . 109
6.2.1 Dag construction . . . 109
6.2.2 Dag Anti-unification . . . 111
6.2.3 Template Computation Extension . . . 114
6.3 Program with functions Transformation . . . 115
6.3.1 $Elim_{fin}$ and $Elim_{fout}$ implementation . . . 115

7 Interfaces **117**
7.1 Parsing and printing . . . 117
7.2 PVS Strategy . . . 118
7.2.1 Deep embedding . . . 119
7.2.2 Strategy definition . . . 120
7.3 PVS Theory Transformation . . . 123
7.3.1 PVS to OCaml . . . 123
7.3.2 Specification with subtyping . . . 125
7.3.3 Proving the equivalence . . . 127
7.3.4 From comparison operator to function . . . 128
7.4 Yices . . . 130

8 Applications **131**

Conclusion **137**

Bibliography **139**

Part I

Transformation Algorithms

1

Elimination in Boolean Expressions

Square roots and divisions can be eliminated from any Boolean expression. These expressions are built with comparisons operators, arithmetic expressions and Boolean operators. This elimination is a particular case of the quantifier elimination on real closed fields. We first describe how the general quantifier elimination could be used to eliminate square roots and divisions by transforming every quantifier free expression with square root and divisions into a quantified expression that is free of divisions and square roots. Then we present our elimination algorithm. This more efficient algorithm normalizes the arithmetic expressions and recursively eliminates divisions and square root to get an equivalent Boolean expression. We assume that the expressions we want to transform are well formed, they do not contain divisions by zero or square root of negative numbers, the equivalence only holds under such hypothesis.

This section presents the algorithm in a very general way and we only give outlines of the proofs. The complete algorithm and formalization in PVS along with the according proof will be presented in Section 5. Let us define the Boolean expressions we consider, they are based on comparisons between arithmetic expression that are defined by the following grammar:

Definition 1.1 (Arithmetic expressions). Given a set of variables $\mathcal{X}$ and a set of real constants $\mathcal{C} \subseteq \mathbb{R}$, we define the following set of terms:

$$\begin{array}{lll} \mathcal{A} ::= & \mathcal{X} & \mid \mathcal{A} + \mathcal{A} \\ & \mid \mathcal{C} & \mid \mathcal{A} \times \mathcal{A} \\ & \mid -\mathcal{A} & \mid \mathcal{A} \,/\, \mathcal{A} \\ & \mid \sqrt{\mathcal{A}} & \end{array}$$

The Boolean expressions $\mathcal{B}$ are based on relations between arithmetic expressions:

Definition 1.2 (Boolean expressions). The Boolean expressions are defined by the following grammar:

$$\begin{array}{llll} \mathcal{B} ::= & \mathbb{B} & \mid \mathcal{A} > \mathcal{A} & \mid \mathcal{A} = \mathcal{A} \\ & \mid \neg\mathcal{B} & \mid \mathcal{A} \geq \mathcal{A} & \mid \mathcal{A} \neq \mathcal{A} \\ & \mid \mathcal{B} \wedge \mathcal{B} & \mid \mathcal{A} < \mathcal{A} & \\ & \mid \mathcal{B} \vee \mathcal{B} & \mid \mathcal{A} \leq \mathcal{A} & \end{array}$$

We also use $-$ as a binary operator, $a - b$ is an abbreviation for $a + (-b)$, ab is the abbreviation for $a \times b$, e^2 denotes the square operation, *i.e.*, $e \times e$ and $\frac{a}{b} = a/b$. We do not mark the parenthesis on the left *e.g.*, $a + b + c = (a + b) + c$ and we use the usual priorities of operations *e.g.*, $a + b \times c = a + (b \times c)$. We respectively denote $\mathcal{A}^{\surd}$ and $\mathcal{B}^{\surd}$ the sets corresponding to $\mathcal{A}$ and $\mathcal{B}$ that do not contain the $\surd$ and $/$ constructors. Given a Boolean formula, we call atom every comparison relation between arithmetic expressions (*e.g.*, $\mathcal{A} > \mathcal{A}$) that is a sub-term of this formula.

1.1 Square Roots and Divisions Elimination using Quantifiers Elimination

In this section we present how we can transform every formula in $\mathcal{B}$ into an equivalent one in $\mathcal{B}^{\surd}$ by using quantifier elimination. To this purpose we extend the set of Boolean expression with quantifiers:

Definition 1.3 (Boolean expression with quantifiers). The following set adds the existential constructor to the set of square root and division free Boolean expressions $\mathcal{B}^{\surd}$:

$$\mathcal{B}^{\exists} ::= \ \mathcal{B}^{\surd} \mid \exists\, \mathcal{X},\ \mathcal{B}^{\exists}$$

Using the characterization of the square root and divisions, we can transform every formula in $\mathcal{B}$ into an equivalent one in $\mathcal{B}^{\exists}$ by introducing a new quantifier for every division or square root:

Example 1.1 (Quantifier introduction).

$$\sqrt{a} = b/c \qquad \text{becomes} \qquad \exists\, s,\ \exists\, d,\ d \times c = b \wedge s \geq 0 \wedge s^2 = a \wedge s = d$$

Every occurrence of square root or division is replaced by the characterization of the function:

Proposition 1.1 (Square root and divisions specification).

$$\forall\, a, b, x \in \mathbb{R}^3,\ b \neq 0 \quad \Longrightarrow \quad a/b = x \Leftrightarrow a = b \times x$$

$$\forall\, a, x \in \mathbb{R}^2,\ a \geq 0 \quad \Longrightarrow \quad \sqrt{a} = x \Leftrightarrow x \geq 0 \wedge x \times x = a$$

We denote $\lll$ the sub-term partial order, $\ll$ the strict one and $T[e \mapsto f]$ the term T where every sub-term equal to e is replaced by f. We define the following algorithm for quantifier introduction:

Algorithm 1.4 (Elimination using quantifiers). Given a Boolean expression f in $\mathcal{B}$, we define the following recursive algorithm:

```
elim_quant(f) := if ∃ a,b ∈ 𝒜, a/b ⋘ f then
                     choose x ∈ {y ∈ 𝒳 | ¬ y ⋘ f};
                     return ∃ x, elim_quant(b × x = a ∧ f[a/b ↦ x]);
                 else if ∃ a ∈ 𝒜, √a ⋘ f then
                     choose x ∈ {y ∈ 𝒳 | ¬ y ⋘ f};
                     return ∃ x, elim_quant(x ≥ 0 ∧ x × x = a ∧ f[√a ↦ x]);
                 else return f;
```

This algorithm terminates since the cardinal of the following finite set:

$$\{e \in \mathcal{A} \mid e \lll f \wedge (\exists\, a, b,\ e = a/b \ \vee\ \exists\, a,\ e = \sqrt{a})\}$$

strictly decreases for each recursive call. Indeed each iteration eliminates one element of that set. Therefore we are able to transform any formula in $\mathcal{B}$ into an equivalent formula in $\mathcal{B}^{\exists}$.

EXAMPLE 1.2 (Elimination using quantifiers).
elim_quant($\sqrt{a} > b/c \wedge \sqrt{a}/d < e$) =

$$\exists\, x, \exists\, y, \exists\, z, x \times c = b \wedge y \times d = z \wedge z \geq 0 \wedge z \times z = a \wedge z > x \wedge y < e$$

We can now use a quantifier elimination algorithm to eliminate the quantifiers from such formulas and therefore obtain an equivalent formula in $\mathcal{B}^{\surd}$ which is free of square roots and divisions.

Quantifier elimination procedures on real closed fields have first been introduced by Tarski in [Tar51] followed by Seidenberg [Sei54] and Cohen [Coh69] who developed the same idea. Quantifier elimination can be used to proove that the theory of real closed fields is decidable, *i.e.*, there exists an algorithm that is able to decide for every formula in that theory if this formula is either true or false. However, these elimination algorithms were more theoretical proofs of the existence of such procedure than effective decision procedures, due to their huge complexity. In 1976, Collins proposed a much more efficient quantifier elimination using cylindrical algebraical decomposition [Col76]. Among others versions of this algorithm has been implemented in the Redlog system [DS96b, DS96a] or QEPCAD project [Bro03]. It has also been formally proved in the Coq proof assistant, see [CM10]. However the complexity of these general procedure depends on the number of free variables in the quantified expressions and therefore, in the context of a program transformation, it would provoke an explosion of the size of the output.

The particular quadratic (and cubic cases) have been studied by Weispfenning in [Wei94, Wei97] but we now present a square root and division elimination procedure that allows us to eliminate nested square roots, all occurrences of the same square roots being handled in one step. This transformation relies on a reduction of arithmetic expression to a particular normal form introduced in Section 1.2 and on a mutually recursive elimination of divisions (see Section 1.3) and square roots (Section 1.4). In Chapter 5 we present a complete formal proof in PVS of this transformation.

1.2 ARITHMETIC EXPRESSION NORMAL FORM

The first step of this transformation relies on a reduction to division normal form, where the head operation is a division, if there is one at top level in the expression. In this section, we say that an operation is at top level if it appears in the expression without being argument of a square root ($\sqrt{a/b} + c$ does not have a division at top level). We define the following normal form that corresponds to this head division reduction:

DEFINITION 1.5 (Division and polynomial normal forms). We define the *Head Division Form* along with the *Polynomial Form*:

$$\begin{aligned} HDF &= PF \mid PF/PF \\ PF &= \mathcal{C} \mid \mathcal{X} \mid PF + PF \mid -PF \mid PF \times PF \mid \sqrt{\mathcal{A}} \end{aligned}$$

This means that any division which is not the head constructor of an expression in HDF is a sub term of a square root argument. The square roots being able to contain any arithmetic expressions, these expressions will be normalized after the elimination of their head square root symbol when they appear at top level.

This reduction to the Head Division Form can be done by commuting the division with all the other operations. The corresponding rules are given in the following definition.

DEFINITION 1.6 (Head division reduction).

$$\begin{array}{rclrclr} e_1 + \dfrac{e_2}{e_3} & \longrightarrow & \dfrac{e_3 \times e_1 + e_2}{e_3} & \dfrac{e_1}{e_2} + e_3 & \longrightarrow & \dfrac{e_1 + e_2 \times e_3}{e_2} & (HD\ +) \\ -\dfrac{e_1}{e_2} & \longrightarrow & \dfrac{-e_1}{e_2} & -(-e_1) & \longrightarrow & e_1 & (HD\ -) \\ e_1 \times \dfrac{e_2}{e_3} & \longrightarrow & \dfrac{e_1 \times e_2}{e_3} & \dfrac{e_1}{e_2} \times e_3 & \longrightarrow & \dfrac{e_1 \times e_3}{e_2} & (HD\ \times) \\ \dfrac{\frac{e_1}{e_2}}{e_3} & \longrightarrow & \dfrac{e_1}{e_2 \times e_3} & \dfrac{e_1}{\frac{e_2}{e_3}} & \longrightarrow & \dfrac{e_1 \times e_3}{e_2} & (HD\ /) \end{array}$$

***Remark* 1.1.** Since most of the arithmetic expressions we are dealing with contain free variables, we can not take the divisions out of the square roots using the following rule:

$$\sqrt{\frac{e_1}{e_2}} \quad \longrightarrow \quad \frac{\sqrt{\epsilon_1 e_1}}{\sqrt{\epsilon_2 e_2}} \qquad \text{with } \epsilon_1, \epsilon_2 \in \{+, -\}^2$$

Indeed the free variables prevent us from guessing the sign of the expressions e_1 and e_2 and therefore to chose the right epsilons.

***Remark* 1.2.** Note that the rules $HD\ /$ only holds when we suppose that the original expression does not fail, if e_2 in the first or e_3 in the second is equal to zero then the left side of the rule fails due to a division by zero whereas the right one does not.

This set of rewriting rules terminates and transforms any expression in $\mathcal{A}$ in an expression in HDF:

PROPOSITION 1.2. *Normalization using the reduction rules defined in Definition 1.6 transforms any term in $\mathcal{A}$ into a formula in HDF.*

Proof. This reduction terminates using the multiset order on the depth of the division operators. All the rules define switches between division and all the operations except square root, therefore if $a/b \ll e$ there exists sq such that $a/b \ll \sqrt{sq} \leqslant e$ All these rules trivially preserve the semantics. ◀

We now present how, given a comparison between two arithmetic expressions in HDF, we are able to transform them by eliminating the top level square root.

1.3 Division Elimination

We want to transform an atomic proposition $e_1 \ \mathcal{R} \ e_2$, where $\mathcal{R} \in \{=, \neq, >, <, \geq, \leq\}$ and $e_1, e_2 \in DNF^2$, into a Boolean formula that only contains relations between PF expressions. When $\mathcal{R} \in \{=, \neq\}$ this can be easily done by multiplying both sides of the equation by the denominator, the product of two PF expressions being in PF. We only describe the case when both sides have divisions as head constructors, rules when only one arithmetic operation has a division being similar:

Definition 1.7 (Division elimination with equality).

$$a_1/a_2 = b_1/b_2 \quad \longrightarrow \quad a_1 \times b_2 = b_1 \times a_2$$

$$a_1/a_2 \neq b_1/b_2 \quad \longrightarrow \quad a_1 \times b_2 \neq b_1 \times a_2$$

However the usual elimination of division rule in comparisons enforces a case distinction on the signs of the denominators:

Definition 1.8 (Division Elimination in Comparisons with Cases).

$$a_1/a_2 > b_1/b_2 \ \longrightarrow \ (a_2 \times b_2 \geq 0 \wedge a_1 \times b_2 > b_1 \times a_2) \vee (a_2 \times b_2 \leq 0 \wedge a_1 \times b_2 > b_1 \times a_2)$$

$$a_1/a_2 \geq b_1/b_2 \ \longrightarrow \ (a_2 \times b_2 \geq 0 \wedge a_1 \times b_2 \geq b_1 \times a_2) \vee (a_2 \times b_2 \leq 0 \wedge a_1 \times b_2 \geq b_1 \times a_2)$$

$$a_1/a_2 < b_1/b_2 \ \longrightarrow \ (a_2 \times b_2 \geq 0 \wedge a_1 \times b_2 < b_1 \times a_2) \vee (a_2 \times b_2 \leq 0 \wedge a_1 \times b_2 < b_1 \times a_2)$$

$$a_1/a_2 \leq b_1/b_2 \ \longrightarrow \ (a_2 \times b_2 \geq 0 \wedge a_1 \times b_2 \leq b_1 \times a_2) \vee (a_2 \times b_2 \leq 0 \wedge a_1 \times b_2 \leq b_1 \times a_2)$$

Using such rules to eliminate the division not only increase the sizes of the comparisons but also the number of comparisons in a formula. In order to avoid this case distinction, we prefer to multiply both sides of the comparison by the square of the denominators. Indeed, if multiplying by the square create even bigger comparisons, it avoids the case distinction since square are always positives:

Definition 1.9 (Division Elimination in Comparisons with Squares).

$$a_1/a_2 > b_1/b_2 \ \longrightarrow \ (a_1 \times a_2 \times b_2 \times b_2 > b_1 \times b_2 \times a_2 \times a_2)$$

$$a_1/a_2 \geq b_1/b_2 \ \longrightarrow \ (a_1 \times a_2 \times b_2 \times b_2 \geq b_1 \times b_2 \times a_2 \times a_2)$$

$$a_1/a_2 < b_1/b_2 \ \longrightarrow \ (a_1 \times a_2 \times b_2 \times b_2 < b_1 \times b_2 \times a_2 \times a_2)$$

$$a_1/a_2 \leq b_1/b_2 \ \longrightarrow \ (a_1 \times a_2 \times b_2 \times b_2 \leq b_1 \times b_2 \times a_2 \times a_2)$$

All these transformations eliminate the head division if it exists and therefore transform every relation between HDF terms into a formula that only embed PF expressions. Let elim_div be the function that implements rules of definitions 1.7 and 1.9, depending on the comparison operator, we have the following proposition:

Proposition 1.3 (From HDF to PF). *Given a relation $e_1 \ \mathcal{R} \ e_2$, where $\mathcal{R} \in \{=, \neq, >, <, \geq, \leq\}$ and $e_1, e_2 \in DNF^2$, by eliminating the head division we get a Boolean formula f that only uses relations between PF expressions:*

$$\mathsf{elim_div}(e_1 \ \mathcal{R} \ e_2) = f \implies$$
$$\forall a_1, a_2 \in \mathcal{A}, \Re \in \{=, \neq, >, <, \geq, \leq\}, \ a_1 \ \Re \ a_2 \ \lll \ f \implies (a_1, a_2) \in PF^2$$

The head division being eliminated, there is no division left at top level of the arithmetic expressions. Given a relation between PF expressions we now introduce the elimination of one square root.

1.4 Square Root Elimination

As for the division elimination, the elimination of one square root symbol relies on a standardization of the expression. First step of this normalization is to chose a non-nested square root expression. Indeed, we want to avoid eliminating several times the same square root expression, thus we chose a square root that is not nested, which means that this square root is not a sub-term of any other square root expression.

Definition 1.10 (Top level square root). Given an arithmetic expression e and an expression a, we call $\sqrt{a}$ a *non-nested square root* when

$$\sqrt{a} \lll e \wedge \forall\, y \in \mathcal{A},\ \sqrt{y} \lll e \Rightarrow \neg\, \sqrt{a} \ll \sqrt{y}$$

Proposition 1.4. *While there is at least one square root, such a top level square root always exists:*

$$\forall\, e \in PF,\ \exists\, s \in \mathcal{A},\ \sqrt{s} \lll e \implies \exists\, a \in \mathcal{A},$$
$$\sqrt{a} \lll e \wedge \forall\, y \in \mathcal{A},\ \sqrt{y} \lll e \Rightarrow \neg\, \sqrt{a} \ll \sqrt{y}$$

Proof. Since $\ll$ is a well-founded order, a maximum of $\{y \in \mathcal{A} \mid \sqrt{y} \lll e\}$ for this order has the property we want. ◀

This allows us to transform any expression with square roots in the following form:

Proposition 1.5 (Square root factorization). *Given $e \in PF$ and sq a top level square root of e:*

$$\exists\, p, r \in PF,\ e = p \times \sqrt{sq} + r \ \wedge\ \neg\sqrt{sq} \lll p \ \wedge\ \neg\sqrt{sq} \lll r$$

This transformation can be done using the following set of rules:

Definition 1.11 (Top level square root factorization).

$$\frac{x \in \mathcal{X}}{x \hookrightarrow 0 \times \sqrt{sq} + x} \qquad \frac{c \in \mathcal{C}}{c \hookrightarrow 0 \times \sqrt{sq} + c}$$

$$\frac{e_1 \hookrightarrow p_1 \times \sqrt{sq} + r_1 \qquad e_2 \hookrightarrow p_2 \times \sqrt{sq} + r_2}{e_1 + e_2 \hookrightarrow (p_1 + p_2) \times \sqrt{sq} + (r_1 + r_2)} \qquad \frac{e_1 \hookrightarrow p_1 \times \sqrt{sq} + r_1}{-e_1 \hookrightarrow (-p_1) \times \sqrt{sq} + (-r_1)}$$

$$\frac{e_1 \hookrightarrow p_1 \times \sqrt{sq} + r_1 \qquad e_2 \hookrightarrow p_2 \times \sqrt{sq} + r_2}{e_1 \times e_2 \hookrightarrow (p_1 \times r_2 + r_1 \times p_2) \times \sqrt{sq} + (p_1 \times p_2 \times sq + r_1 \times r_2)} \qquad \frac{a \neq sq}{\sqrt{a} \hookrightarrow 0 \times \sqrt{sq} + \sqrt{a}}$$

$$\frac{}{\sqrt{sq} \hookrightarrow 1 \times \sqrt{sq} + 0}$$

Since sq is a top level square root, neither p nor q contains any $\sqrt{sq}$.

Now given a relation between two PF forms $e_1 \mathcal{R} e_2$ being the result of the division elimination introduced in Section 1.3. We can transform this relation by factorizing the expression $e_1 - e_2$ with a top level square root. Therefore our new relation has the following form $p \times \sqrt{sq} + r \mathcal{R} 0$. We transform this relation into a new formula that do not

contain $\sqrt{sq}$ as a sub-term anymore. This transformation relies on a case distinction on the signs of p and r. The following arrays present, depending on the sign of p and r, formulas where $\sqrt{sq}$ does not appear and which are equivalent to $p \times \sqrt{sq} + r \mathcal{R}\ 0$ when $\mathcal{R}$ is $=$ or $>$

- Transformation of $p \times \sqrt{sq} + r = 0$:

$r \backslash p$	$-$	0	$+$
$-$	$\bot$	$\bot$	$p^2 \times sq - r^2 = 0$
0	$sq = 0$	$\top$	$sq = 0$
$+$	$p^2 \times sq - r^2 = 0$	$\bot$	$\bot$

- Transformation of $p \times \sqrt{sq} + r > 0$:

$r \backslash p$	$-$	0	$+$
$-$	$\bot$	$\bot$	$p^2 \times sq - r^2 > 0$
0	$\bot$	$\bot$	$sq \neq 0$
$+$	$r^2 - p^2 \times sq > 0$	$\top$	$\top$

We can define equivalent case distinctions for the other operators, *i.e.*, $\geq$, $<$, $\leq$ and $\neq$. Therefore every Boolean formula of the form $p \times \sqrt{sq} + r \mathcal{R}\ 0$ can be transformed into an equivalent one that do not contain $\sqrt{sq}$ anymore:

DEFINITION 1.12 (Square root elimination). The following rules eliminate one top level square root:
$p \times \sqrt{sq} + r = 0 \longrightarrow p \times r \leq 0 \wedge p^2 \times sq - r^2 = 0$
$p \times \sqrt{sq} + r > 0 \longrightarrow (p \geq 0 \wedge r \geq 0) \vee (p \geq 0 \wedge p^2 \times sq - r^2 > 0) \vee (r \geq 0 \wedge p^2 \times sq - r^2 < 0)$ And we define similar rules for the other operators $\geq$, $<$, $\leq$ and $\neq$.

This transformation might introduce new divisions at top level (divisions that are in sq), therefore before being able to remove another top level square root, we have to normalize into HDF form and eliminate the division if there is one. Therefore we define the following algorithm:

ALGORITHM 1.13 (Square roots and divisions elimination). While the comparisons contain divisions or square roots, do:

i) Reduce to HDF form using rules of Definition 1.6

ii) Eliminate head division using one rule of Definition 1.9 or 1.8

iii) Factorize using one top level square root with rules of Definition 1.11

iv) Eliminate this square root using one of the rules of Definition 1.12

This algorithm terminates using the multiset of square root sub-terms given the following lemmas:

LEMMA 1.6 (Square roots sub-terms conservation). *For every rule from definitions 1.6, 1.9, 1.8 and 1.11, transforming e into f, the following property holds:*

$$\forall a \in \mathcal{A}, \sqrt{a} \lll f \Longrightarrow \sqrt{a} \lll e$$

LEMMA 1.7 (Square roots elimination). *The rules of Definition 1.12 transforming $p \times \sqrt{sq} + r \; \mathcal{R} \; 0$, where sq is a top level square root, into f have the following sub-term property:*

$$\neg \sqrt{sq} \lll f \;\wedge\; \forall a \in \mathcal{A}, \sqrt{a} \lll f \Longrightarrow \sqrt{a} \lll e$$

It ensures that, after applying the four step of the algorithm, all the remaining square roots were already in the input term and are different from the one we eliminated. Therefore, even if the last rule produces 6 relations from one, the set of the square roots appearing in each relation strictly decreases and therefore enforce the termination of this algorithm.

1.5 COMPLEXITY AND EXAMPLES

Using this algorithm the size of the transformed formula quickly grows when the number of square roots in the expression increase. Indeed as mentioned earlier, every elimination of square roots can produce up to 6 relations, and these relations might be bigger than the original one ($p^2 \times sq - r^2$ is bigger than $p \times \sqrt{sq} + r$). Given $\#_{\sqrt{}}(e)$ the number of square roots appearing in the expression e (*i.e.*, the cardinal of $\{a \mid \sqrt{a} \lll e\}$), then the number of atoms in the formula produced by this elimination on e is bounded by $6^{\#_{\sqrt{}}}(e)$. Using variable definitions as it will be introduced in Section 3.6, this exponential can be brought to $4^{\#_{\sqrt{}}}(e)$, however this complexity can still easily makes the size of the output explodes. Before presenting the constrained anti-unification problem in the next section we first give some examples of square roots and division elimination in Boolean formulas using this algorithm:

EXAMPLE 1.3 (Quadratic function root comparison).

$(-b + \sqrt{(}\Delta))/(a) > c \longrightarrow$

$$\begin{aligned}
&a > 0 \wedge -b \times a - c \times a \times a > 0 \vee \\
&a > 0 \wedge a \times a \times \Delta - (-b \times a - c \times a \times a) \times (-b \times a - c \times a \times a) > 0 \vee \\
&-b \times a - c \times a \times a > 0 \wedge a \times a \times \Delta - (-b \times a - c \times a \times a) \times (-b \times a - c \times a \times a) < 0
\end{aligned}$$

EXAMPLE 1.4 (Quadratic function roots comparisons).

$(-b + \sqrt{\Delta})/a > (-b - \sqrt{\Delta})/a \longrightarrow$

$$a \times a \times a + a \times a \times a > 0 \wedge (a \times a \times a + a \times a \times a) \times (a \times a \times a + a \times a \times a) \times \Delta > 0$$

EXAMPLE 1.5 (Square root sum).

$a \times \sqrt{sa} + b \times \sqrt{sb} > 0 \longrightarrow$

$$\begin{aligned}
&a > 0 \wedge b > 0 \wedge b \times b \times sb > 0 \vee \\
&a > 0 \wedge a \times a \times sa - b \times b \times sb > 0 \wedge (a \times a \times sa - b \times b \times sb) \times (a \times a \times sa - b \times b \times sb) > 0 \vee \\
&b > 0 \wedge b \times b \times sb > 0 \wedge a \times a \times sa - b \times b \times sb < 0 \wedge \\
&\qquad\qquad (a \times a \times sa - b \times b \times sb) \times (a \times a \times sa - b \times b \times sb) > 0
\end{aligned}$$

2

Constrained Anti-Unification

The anti-unification problem was introduced independently by J.C. Reynolds [Rey70] and G.D. Plotkin [Plo70] in 1970. This problem is the dual of the unification problem. While unification aims at computing a common instance of two terms, the anti-unification computes a template (also called generalization or anti-unifier) of two terms such that substitutions applied to this template produce the input terms:

Definition 2.1 (Anti-unification). Given two terms t_1 and t_2, the anti-unification problem consists in finding a term t and two substitutions σ_1 and σ_2 such that:

$$t_1 = t\sigma_1 \qquad \text{and} \qquad t_2 = t\sigma_2$$

Such a term t is called an anti-unifier or template of t_1 and t_2.

Unification has been widely studied in the fields of proof theory and rewriting (see [BS01] for a survey) and therefore many papers have presented efficient solutions [Rob65] where equality is considered modulo various equational theories (*e.g.*, [Hue76]).

Algorithms for first order anti-unification have been introduced in [Rey70, Plo70, Hue76] and one for higher-order anti-unification, in the Calculus of Constructions, is presented in [Pfe91]. These algorithms are used, for instance, for proof generalization and, more recently, termination [AEMO08] using generalization modulo some equational theories [Pot89]. Anti-unification has also been used to find general properties or solutions of algebraic expressions [LMM88, OSW05] or to detect code duplication [BKZ09, KLV11]. All of these applications focus on computing the most specific (or least general) template which is the dual of the most general unifier, that is:

Definition 2.2 (Most specific template). Given two terms, t_1 and t_2, a most specific template is a template t such that:

For all s, if s is a template of t_1 and t_2 then exists σ such that $t = s\sigma$

The importance of this most specific template comes from the fact that one of the main purposes of usual anti-unification is to factorize terms, the template being the common part of all the terms while the substitutions capture their differences. This factorization is then used to define a variable corresponding to that template and reuse this variable in all the terms it anti-unifies.

In this chapter we present a variant of the anti-unification problem. The *constrained anti-unification* problem consists in computing a template with the constraint that the language of the terms allowed in the substitutions is a subset of that of the input terms.

We give a formal definition of this constrained anti-unification problem, state some general properties of this problem in different equational theories and introduce how directed acyclic graph can be used to compute a specific template. All of these features are then used to provide a tailored constrained anti-unification algorithm modulo an arithmetic theory.

2.1 Definition of the Constrained Anti-Unification

In this section we first describe the problem of constrained anti-unification on tree-like terms. Given a set of variables $\mathcal{X}$ and a signature Σ which is a set of function symbols, we define the following notations:

- $\Sigma^{(m)}$ is the set of functional symbols in Σ of arity m.
- $\mathcal{T}(\Sigma, \mathcal{X})$ (or simply $\mathcal{T}$) is the set of terms t over Σ and $\mathcal{X}$ defined by the following grammar:
  $t ::= x \mid f(t_1, ..., t_n)$ where $x \in \mathcal{X}$ and $f \in \Sigma^{(n)}$.
- A substitution $\sigma \in \mathfrak{S}$, is a partial mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}(\Sigma, \mathcal{X})$.
- The domain of the substitution σ is denoted by $\mathcal{D}(\sigma)$.
- $\mathcal{I}(\sigma)$ is the image of σ, *i.e.*, $\{t \in \mathcal{T}(\Sigma, \mathcal{X}) | \exists x \in \mathcal{D}(\sigma), t = \sigma(x)\}$.
- $[x \mapsto a, y \mapsto b]$ is the substitution that replaces x by a and y by b.
- $t\sigma$ is the application of σ to a term t.
- $\sigma_1\sigma_2$ is the composition of σ_1 and σ_2, *i.e.*, the substitution such that:
  $\forall\, t,\ t(\sigma_1\sigma_2) = (t\sigma_1)\sigma_2$.
- When $\mathcal{D}(\sigma_1) \cap \mathcal{D}(\sigma_2) = \emptyset$, $\sigma_1 \| \sigma_2$ is the parallel substitution of σ_1 and σ_2, *i.e.*, the substitution such that:
  if $x \in \mathcal{D}(\sigma_1)$ then $(\sigma_1 \| \sigma_2)(x) = \sigma_1(x)$ else $(\sigma_1 \| \sigma_2)(x) = \sigma_2(x)$
- $\ll$ is the strict sub-term order on terms, that is, the inductive relation defined by:
  $a \ll f(t_1, .., t_n) \quad \equiv \quad (\exists\, i,\ a = t_i \vee a \ll t_i)$
  and the associated partial order $\leqslant$, such that $s \leqslant t \Leftrightarrow s = t \vee s \ll t$

We usually denote by x, y, z... the variables in $\mathcal{X}$, by a, b, c... the constants in $\Sigma^{(0)}$ and by f, g, h... the other symbols in Σ. Given these definitions and notations, we now define the constrained anti-unification:

Definition 2.3 (Template with constraint). Given $\overline{\Sigma} \subseteq \Sigma$ and a term s in $\mathcal{T}(\Sigma, \mathcal{X})$, a term t in $\mathcal{T}(\Sigma, \mathcal{X})$ is a $\overline{\Sigma}$-template of s, denoted $s \preccurlyeq_{\overline{\Sigma}} t$ when:

$$\exists\, \sigma,\ t\sigma \ = \ s \ \wedge \ \mathcal{I}(\sigma) \subseteq \mathcal{T}(\Sigma \backslash \overline{\Sigma}, \mathcal{X})$$

Symbols in $\overline{\Sigma}$ are the *forbidden symbols*. An example of such a $\overline{\Sigma}$-template is:

EXAMPLE 2.1. If $\Sigma = \{f, g, a, b\}$ and $\overline{\Sigma} = \{f, a\}$ then $f(x, y, a)$ is a $\overline{\Sigma}$-template of $f(x, z, a)$ and $f(b, g(x), a)$ with the substitutions $[y \mapsto z]$ and $[x \mapsto b;\ y \mapsto g(x)]$.

The usual anti-unification problem is a way to factorize terms, the most specific template represents the common part of the input terms and the substitutions embed the differences. The constrained anti-unification has a different goal, it aims at factorizing the terms depending on the symbol they use, the template has to contain all the forbidden symbols that were used in the input terms whereas the substitution contains the rest of the term. Unlike the usual anti-unification, we do not aim at computing the most specific template but we aim at computing a large set of templates and select the best according to a criterion that is completely different. Indeed, we will use the template to replace some variables in Boolean expressions before eliminating square roots and divisions with the algorithm introduced in chapter 1. Thus, the template is used many times and we prefer a small template and large substitutions, this means that we might even chose one of the less specific templates.

Let us first extend this definition to define the template of a set of terms:

DEFINITION 2.4 (Template of finite set). Given $\overline{\Sigma} \subseteq \Sigma$ and a finite set of terms $\mathcal{S}$ included in $\mathcal{T}(\Sigma, \mathcal{X})$, a term t in $\mathcal{T}(\Sigma, \mathcal{X})$ is a template of $\mathcal{S}$, when for all s in $\mathcal{S}$, $s \preccurlyeq_{\overline{\Sigma}} t$.

***Remark* 2.1.** For unconstrained problems, there is always a template: the (fresh) variable x, since, for every term $t = x[x \mapsto t]$. This is no longer the case when we add constraints since the $\overline{\Sigma}$-template of a set of expressions does not always exist. As soon as one of the terms contains a forbidden symbol, a simple fresh variable is not a $\overline{\Sigma}$-template anymore, *e.g.*, given Σ and $\overline{\Sigma}$ from Example 2.1, $f(x, y, a)$ and $g(b)$ do not have a common $\overline{\Sigma}$-template.

We aim at computing a common template of a finite set of terms. Since the $\overline{\Sigma}$-anti-unification relation is transitive, a template of a set of terms can be recursively computed by using anti-unification on pairs of terms.

PROPOSITION 2.1 ($\overline{\Sigma}$-template transitivity). *If r is a $\overline{\Sigma}$-template of s and s a $\overline{\Sigma}$-template of t then r is a $\overline{\Sigma}$-template of t*

Proof. $s = r\sigma \wedge t = s\sigma' \Longrightarrow t = r(\sigma\sigma')$ and $\mathcal{I}(\sigma\sigma') \subseteq \mathcal{T}(\Sigma \backslash \overline{\Sigma}, \mathcal{X})$ ◀

Without any equational theory, the constrained anti-unification of two terms is quite simple.

ALGORITHM 2.5 (Constrained anti-unification algorithm). The following recursive function *ctmp* computes (if it exists) a $\overline{\Sigma}$-template of t and t':

(V) when $(t,\ t') \in \mathcal{T}(\Sigma \backslash \overline{\Sigma}, \mathcal{X}) \times \mathcal{T}(\Sigma \backslash \overline{\Sigma}, \mathcal{X})$, $ctmp(t, t') = x$

otherwise

(R) $ctmp(f(t_1, ..., t_n), f(t'_1, ..., t'_2)) = f(ctmp(t_1, t'_1), ..., ctmp(t_n, t'_n))$

(F) $ctmp(f(...), g(...)) =$ Fail (No $\overline{\Sigma}$-template can be computed)

When we have two terms in $\mathcal{T}(\Sigma\backslash\overline{\Sigma}, \mathcal{X})$, *i.e.*, with no forbidden symbols, the template is a variable. This is because, unlike in the usual anti-unification, we are not interested in computing the most specific template. However the variable has to be fresh in order to avoid conflicts in the substitutions composition when applying the recursive step of rule (R).

PROPOSITION 2.2 (Recursive template construction). *Given $f \in \Sigma^{(n)}$, $t_1, ..., t_n$ and $t'_1, ..., t'_n$ pairwise anti-unifiable terms such that $\overline{t_1}, .., \overline{t_n}$ are the corresponding templates and $\sigma_1, ..., \sigma_n$ and $\sigma'_1, ..., \sigma'_n$ the associated substitutions:*

$$\forall\ j,\ t_j = \overline{t_j}\sigma_j\ \wedge\ t'_j = \overline{t_j}\sigma'_j$$

then, if the substitution have distinct domains, we have a template for $f(t_1, ..., t_1)$ and $f(t'_1, ..., t'_n)$:

$$(\forall\ i, j,\ j \neq i\ \Rightarrow\ \overline{t_j}\sigma_i = \overline{t_j}\ \wedge\ \overline{t_j}\sigma'_i = \overline{t_j}\ \wedge$$
$$\mathcal{D}(\sigma_i) \cap \mathcal{D}(\sigma_j) = \emptyset\ \wedge\ \mathcal{D}(\sigma'_i) \cap \mathcal{D}(\sigma'_j) = \emptyset) \Longrightarrow$$
$$f(t_1, ..., t_n) = f(\overline{t_1}, ..., \overline{t_n})(\sigma_1 \| ... \| \sigma_n) \wedge f(t'_1, ..., t'_n) = f(\overline{t_1}, ..., \overline{t_n})(\sigma'_1 \| ... \| \sigma'_n)$$

Proof. The distinct domains allows us to construct the parallel substitution. The $\overline{t_j}\sigma_k = \overline{t_j}$ condition states that the free variables of $\overline{t_j}$ are not in the domain of σ_k, therefore we have $\overline{t_j}(\sigma_1 \| ... \| \sigma_n) = \overline{t_j}\sigma_j$ and we get the following equality:

$$f(\overline{t_1}, ..., \overline{t_n})(\sigma_1 \| ... \| \sigma_n) = f(\overline{t_1}(\sigma_1 \| ... \| \sigma_n), ..., \overline{t_n}(\sigma_1 \| ... \| \sigma_n)) = f(\overline{t_1}\sigma_1, ..., \overline{t_n}\sigma_n) = f(t_1, ..., t_n)$$

◀

As soon as one of the terms contains a forbidden symbol and the head symbols are different the anti-unification fails. Therefore the set of elements that can be anti-unified is quite small. We can anti-unify more terms if we take into account an equational theory on terms. Hence in the next section, we generalize anti-unification, modulo an equational theory.

2.2 ANTI-UNIFICATION MODULO AN EQUATIONAL THEORY

The anti-unification modulo an equational theory is defined using the equality modulo the theory (using the axioms of the theory).

DEFINITION 2.6 (Anti-unification modulo theory). Given two terms t_1 and t_2 and a theory E, an anti-unifier modulo E is a term t and two substitutions σ_1, σ_2 such that:

$$t_1 =_E t\sigma_1 \qquad \text{and} \qquad t_2 =_E t\sigma_2$$

Where $=_E$ is the equality in the theory E. However we use the simple $=$ notation when the context is clear.

In this section we will see that the use of an equational theories and some properties on the sets Σ and $\overline{\Sigma}$ might allow the constrained anti-unification to be complete.

DEFINITION 2.7 (Completeness). Given $\overline{\Sigma} \subseteq \Sigma$, $\overline{\Sigma}$-constrained anti-unification is said to be complete when every finite set of terms has a $\overline{\Sigma}$-template.

2.2.1 Neutral Elements

A simple case that enables the completeness of the constrained anti-unification is when we only have constants and binary symbols, every binary operation having a constant (*i.e.,* in $\Sigma^{(0)}$) left or right neutral element. We say that $e \in \Sigma^{(0)}$ is a left (respectively right) neutral element of f when for all term t, $f(e,t) =_E t$ (respectively $f(t,e) =_E t$).

PROPOSITION 2.3 (Right neutral restriction). *If $\Sigma = \Sigma^{(0)} \cup \Sigma^{(2)}$, $\overline{\Sigma} \subseteq \Sigma^{(2)}$ and every function f in $\Sigma^{(2)}$ has a left or right neutral element e_f in $\Sigma^{(0)}$ then for every pair of terms t_1 and t_2 there exists a $\overline{\Sigma}$-template.*

Proof. We add to the (V) and (R) rules, the following rules:

(LRN) $ctmp(f(t_1,t_2),s) = f(ctmp(t_1,s), ctmp(t_2,e_f))$

(RRN) $ctmp(s,f(t_1,t_2)) = f(ctmp(s,t_1), ctmp(e_f,t_2))$

when f has a right neutral element e_f. We use the symmetrical rules (LLN) and (RLN) when f admits a left neutral element. This extended set of rules terminates since recursive calls are made on strict sub-terms. Therefore if the head symbols are different, there are two possibilities:

- If one of the term still contains a forbidden symbol, then its head symbol is in $\Sigma^{(2)}$, it has a neutral element so one of the rule can apply
- If there is no forbidden symbol then the (V) rule terminates the anti-unification ◀

By using this neutral element rules, we can always anti-unify simple arithmetic expressions:

EXAMPLE 2.2 (Rational number arithmetic). If $\Sigma = \mathbb{Q} \cup \{+,-,\times,/\}$ and all the binary operators are forbidden ($\overline{\Sigma} = \{+,-,\times,/\}$ (notice that $-$ and $/$ have a right neutral element) we can always anti-unify:
$(x \times y + (t \times u/v)) - (w+z)$ is a template of $a + (b \times c)/d$ and $a' \times b' - (c' + d')$ with the following substitutions:

- $[x \mapsto a, y \mapsto 1, t \mapsto b, u \mapsto c, v \mapsto d, w \mapsto 0, z \mapsto 0]$:
$$a + (b \times c)/d = (a \times 1 + (b \times c/d)) - (0+0)$$
- $[x \mapsto a', y \mapsto b', t \mapsto 0, u \mapsto 1, v \mapsto 1, w \mapsto c', z \mapsto d']$:
$$a' \times b' - (c' + d') = (a' \times b' + (0 \times 1/1)) - (c' + d')$$

2.2.2 The Switch Operator

Another condition that ensures the completeness of the constrained anti-unification is the existence of a switch operator:

DEFINITION 2.8 (Simple switch operator). A switch operator sw in $\mathcal{T}(\Sigma,\mathcal{X})$ is a term that has the following property:

$$\exists\, e_1, e_2 \in \mathcal{T}(\Sigma,\mathcal{X}),\ x,y,z \in \mathcal{X},$$
$$\forall\, s,t \in \mathcal{T}(\Sigma,\mathcal{X}),\ sw[x \mapsto e_1, y \mapsto s, z \mapsto t] = s \ \wedge\ sw[x \mapsto e_2, y \mapsto s, z \mapsto t] = t$$

The terms e_1 and e_2 are called the switch elements and x, y and z the switch variables.

EXAMPLE 2.3 (Switch operators). We can define switch operators in different theories:
In arithmetic, $sw = x \times y + (1 - x) \times z$ with $e_1 = 1$ and $e_2 = 0$.
On Booleans, $sw = (x \wedge y) \vee (\neg x \wedge z)$ with $e_1 = \top$ and $e_2 = \bot$.
In a programming language, $sw = if\ x\ then\ y\ else\ z$ with $e_1 = true$ and $e_2 = false$.

PROPOSITION 2.4 (Switch completeness). *If $\mathcal{T}(\Sigma, \mathcal{X})$ admits a switch operator and if the switch elements are $\overline{\Sigma}$-anti-unifiable, then any finite set of terms has a $\overline{\Sigma}$-template.*

Proof. Given e, a $\overline{\Sigma}$-template of e_1 and e_2, and the corresponding substitutions σ_1 and σ_2, then for all s and t in $\mathcal{T}(\Sigma, \mathcal{X})$, $sw[x \mapsto e, y \mapsto s, z \mapsto t]$ is a $\overline{\Sigma}$-template of s and t with the same substitutions σ_1 and σ_2. ◀

When we are looking for small templates, the use of the switch operator has to be avoided as much as possible since in that case the anti-unification algorithm produces a template whose size is bigger than the sum of the sizes of the input terms. Nevertheless, in many usual theories, this operator can be constructed and allows the completeness of the constrained anti-unification. When such element exists we define a switch rule that can replace the one that fails, *i.e.*, (F):

DEFINITION 2.9 (Switch rule). If sw is a switch term and e a template of the switch elements, the following rule produces a constrained template of two terms:

(SW) $ctmp(t_1, t_2) = sw[x \mapsto e, y \mapsto t_1, z \mapsto t_2]$

We say that a is a left (respectively right) absorbing element of f when for all t, $f(a, t) = a$ (respectively $f(t, a) = a$). When the neutral element of one function is the absorbing element of another one, we can define n-ary switches.

DEFINITION 2.10 (N-ary switch). In every theory that has the following elements:

- a binary function f, with a neutral element e_f
- a binary function g, with e_f as left absorbing element and a left neutral element e_g

A n-ary switch is the following term:

$$sw_n = f(g(x_1, y_1), f(g(x_2, y_2), ..., f(g(x_{n-1}, y_{n-1}), g(x_n, y_n))))$$

That is:

- $sw_2 = f(g(x_1, y_1), g(x_2, y_2))$
- $sw_{n+1} = f(g(x, y), sw_n)$ where x and y do not appear in sw_n

This switch directly defines the template of any set of terms:

PROPOSITION 2.5 (N-ary template). *Given $t_1, ..., t_n$ a set of terms and the hypothesis of definition 2.10, then*

$$sw_n[y_1 \mapsto t_1, ..., y_n \mapsto t_n]$$

is a template of $t_1, ..., t_n$

Proof. $\forall\, i,\ sw_n[x_1 \mapsto e_f, ..., x_{i-1} \mapsto e_f, x_i \mapsto e_g, x_{i+1} \mapsto e_f, ..., x_n \mapsto e_f] = y_i$ ◀

EXAMPLE 2.4. On Booleans:

$sw_n = \bigvee_{i=1}^{n} (x_i \wedge y_i)$ with $e_f = \bot$ and $e_g = \top$

We have seen that, using neutral elements and switch properties, we are able to find constrained templates in many equational theories. However, since we are looking for particular templates, we might want to extend the set of possible templates. This can be done using other axioms of the equational theory.

2.2.3 Function Commutation and Normal Forms

We call a *function commutation axiom*, every axiom of the form $g(...) = f(...)$ that changes the head symbol of a term, *e.g.*, distributivity $r \times (s + t) = r \times s + r \times t$; $r/s + t = (r + t \times s)/s$; $\neg(A \wedge B) = (\neg A \vee \neg B)$ etc. As explained in section 2.2.2 we want to avoid using the switch rule since it produces really big templates. Therefore we prefer using these axioms to change the head symbols of terms to apply the recursive rule (R) then using the switch rule directly.

EXAMPLE 2.5 (Distributivity).

$x \times (1/y + 1/z)$ is a template of $a.(1/b + 1/c)$ and $a'/b' + a'/c'$

The use of these axioms extends the set of templates and also allows the computations of normal forms that simplifies the anti-unification algorithm:

EXAMPLE 2.6 (Division anti-unification). When $\Sigma = \mathbb{Q} \cup \{+, -, \times, /\}$ and $\overline{\Sigma} = \{/\}$ (division is the only forbidden function), then since every expression can be written t/u, t and u not containing any division, x/y is the smallest most general template of any set of expressions.

The use of normal forms really increases the efficiency of the algorithm when we consider computing a template of a whole set of expressions at the same time instead of recursively computing the templates of pairs of elements. Indeed it is easier to anti-unify terms that already have the same shape:

EXAMPLE 2.7 (Boolean DNF anti-unification). On the Booleans $\{\vee, \wedge, \neg, \mathbb{B}\}$, if $\overline{\Sigma} = \{\vee\}$, by transforming every term in its disjunctive normal form we can find a template whose number of $\vee$ is the maximal number of $\vee$ in the normal forms:

$$\begin{array}{lcl} (a_1 \wedge a_2) \vee (a_3 \wedge a_4) \vee a_5 & & \\ (b_1 \wedge b_2 \wedge b_3) \vee b_4 & \longrightarrow & x \vee y \vee z \\ c_1 \vee c_2 & & \end{array}$$

In order to minimize the size of the template we use the neutral elements and function commutation axioms as a priority and only use the switch operator when we have no other choice. We have seen how the use of an equational theory really enlarges the set of possible templates. However, it seems too complex to try to compute all the possible templates modulo a large equational theory, such as the arithmetic. However, for our application, one of the criteria to chose the best template is the number of forbidden function occurrences that we want to minimize. We now present how the use of directed acyclic graphs as representations of terms helps us minimize the number of forbidden functions symbols in the template.

2.3 Anti-Unification on Dag-like Terms

As mentioned previously, we want to compute a "small" template, regarding the number of forbidden function occurrences. More precisely, we want to minimize the number of function calls on distinct elements since, as introduced in section 1.5, the complexity of the elimination in Boolean expressions depends on this number.

Definition 2.11 (Number of forbidden occurrences). Given Σ and $\overline{\Sigma}$, the number of forbidden occurrences of t, denoted $\#_{\overline{\Sigma}}(t)$ is the cardinal of the following set:

$$\{\ f(t_1, ..., t_n) \mid f(t_1, ..., t_n) \leqslant t \ \wedge \ f \ \in \ \overline{\Sigma}\ \}$$

For example the number of f occurrences in $g(f(b, f(a, b)), f(a, b))$ is 2. In order to compute a template which minimizes that number, we adopt a directed acyclic graph (*a.k.a.* dag) based representation.

2.3.1 Dag Representation

The dag representation proposed in this document uses pointers to represent sharing. Therefore the dag nodes are represented by terms extended with pointers in $\mathbb{N}^*$.

Definition 2.12 (Directed Acyclic Graphs). Let $\mathcal{T}(\Sigma, \mathcal{X}, \mathbb{N}^*)$ (or $\dot{\mathcal{T}}$ for short) denote the set of dag terms (or nodes) corresponding to the following grammar:

- $dn \ ::= \ x \mid f(dn_1, ..., dn_m) \mid \dot{k}$ where $x \in \mathcal{X}$, $f \in \Sigma^{(m)}$ and $k \in \mathbb{N}^*$

The dags are lists of such nodes, *i.e.*, $d := [dn_0; ...; dn_n]$, where $\dot{k}$ represents a pointer to the k-th element of the list, $\mathcal{D}(\Sigma, \mathcal{X})$ is the set of these dags. We call the length of the dag the length of its list. We denote the `cons` infix constructor by :: and $[dn_i]_0^n$ the list $[dn_0; ...; dn_n]$. $pt(dn)$ is the set of pointers appearing in dn, *i.e.*, $\{k \mid \dot{k} \leqslant dn\}$. We also extend the substitutions to dag nodes such that $\dot{k}\sigma = \dot{k}$ and to dags $[dn_i]_0^n\sigma = [dn_i\sigma]_0^n$.

We use dags to represent sharing in the terms, in practice we only use sharing for the arguments of the forbidden function calls:

Example 2.8 (Dag representation). The term $g(f(a), f(h(b, f(a))))$ where the f calls arguments are shared is represented by:

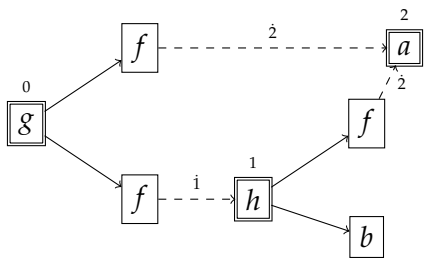


Figure 2.1: Dag representation

and we also use an array representation for dags:

$$\overset{0}{\underline{g(f(\dot{2}), f(\dot{1}))}} \;\|\; \overset{1}{h(b, f(\dot{2}))} \;|\; \overset{2}{a}$$

We use this array representation for dags and separate the first element (the root) for clarity reasons.

In order to assure acyclic behaviors we only authorize in node i pointers to indexes bigger than i, this hypothesis is called *right dependency hypothesis*.

DEFINITION 2.13 (Right dependency hypothesis). Given $d = [d_i]_0^n$ a dag, we say that d is a *right dependency dag* when

$$\forall i \in \{0; ...; n\},\ pt(dn_i) \subseteq \{i+1; ...; n\}$$

This hypothesis allows us to define the following order on dags.

DEFINITION 2.14 (Order on dags). Given $[dn_i]_0^n$ and $[dg_i]_0^n$ we say that $[dn_i]_0^n \dot{\gg} [dg_i]_0^n$ when:

$$\forall i \geq 1, dn_i = dg_i \wedge (min(pt(dn_0)) > min(pt(dg_0)) \vee dn_0 \gg dg_0)$$

where $\gg$ is the sub-term relation defined in Section 2.1.

To compare 2 dags, the nodes bigger than 1 have to be equal. The relation $\dot{\gg}$ is well founded since the pointers in the sub-terms are included in the pointers of the term (*i.e.*, $dn_0 \gg dg_0 \implies (min(pt(dn_0)) \geq min(pt(dg_0)))$. Therefore it provides us a termination criterion for a set of rules and an induction scheme for right dependency dags. Using that order, we can define the semantics $d^\top$ of a dag d in $\mathcal{D}(\Sigma, \mathcal{X})$ as a term in $\mathcal{T}(\Sigma, \mathcal{X})$:

DEFINITION 2.15 (Dag to term). The semantics of a dag is defined by:

- $(x :: [dn_i]_1^n)^\top = x$
- $(f(dn_1, ..., dn_m) :: [dn_i]_1^n)^\top = f((d_1 :: [dn_i]_1^n)^\top, ..., (dn_m :: [dn_i]_1^n)^\top)$
- $(\dot{k} :: [dn_i]_1^n)^\top = (dn_k :: [dn_i]_1^n)^\top$

The semantics of a dag without pointers is equal to the root node:

PROPOSITION 2.6 (Dags without pointers). *When the first element of the dag is a term in $\mathcal{T}(\Sigma, \mathcal{X})$ then the semantics of the dag is this term itself:*

$$\forall\ dn_0, ..., dn_n \in \dot{\mathcal{T}},\ dn_0 \in \mathcal{T} \implies (dn_0 :: [dn_i]_1^n)^\top = dn_0$$

Proof. By induction using the order from Definition 2.14 ($\dot{\gg}$-induction). ◀

The semantics equality means that the pointers identifiers are irrelevant:

EXAMPLE 2.9 (Dag equivalence).

$$[f(\dot{1}, \dot{2}, \dot{3}); a; b; c]^\top = [f(\dot{1}, \dot{3}, \dot{2}); a; c; b]^\top = [f(\dot{3}, \dot{2}, \dot{1}); c; b; a]^\top$$

This renaming is the application of a permutation to the dag with the following definition:

DEFINITION 2.16. Given τ a permutation of $\{1, ..., n\}$, the application of τ to a dag node is defined by:

- $\tau(x) = x$
- $\tau(f(d_1, ..., d_m) = f(\tau(d_1), ..., \tau(d_m))$
- $\tau(\dot{k}) = (\tau(\dot{k}))$

and $\tau([d_0; d_1; ...; d_n]) = [\tau(d_0); \tau(d_{\tau^{-1}(1)}); ...; \tau(d_{\tau^{-1}(n)})])$.

Applying such a permutation to a dag does not change the term it represents

PROPOSITION 2.7 (Permutation preserves the semantics). *Given a dag d and a permutation τ of $\{1, ..., n\}$, we have: $(d)^\top = (\tau(d))^\top$*

Proof. By $\ll$-induction, we only give the pointer case:

$$\begin{aligned}(\tau(\dot{k} :: [d_i]_1^n))^\top &= (\tau(\dot{k}) :: [\tau(d_{\tau^{-1}(i)})]_1^n)^\top \\ &= (\tau(d_{\tau^{-1}(\tau(k))}) :: [\tau(d_{\tau^{-1}(i)})]_1^n)^\top \\ &= (\tau(d_k :: [d_i]_1^n))^\top \\ &= (d_k :: [d_i]_1^n)^\top\end{aligned}$$

◀

Of course, for every dag we can find at least one permutation such that the representation satisfies the right dependency hypothesis:

PROPOSITION 2.8 (Right Dependency Representation). *For all dags d there exists a permutation τ such that $\tau(d)$ is a right dependency dag*

Proof. Renaming nodes using breadth-first numbering (see [Oka00]) allows us to define a right dependency dag. ◀

In practice we avoid pointers in the substitutions and only consider substitutions with usual tree terms in their image. When substitutions do not contain pointers then the substitution commutes with the semantics:

PROPOSITION 2.9 (Substitution and semantics commutation).

$$\forall\, d \in \mathcal{D}(\Sigma, \mathcal{X},\ \sigma\ \in \mathfrak{S},\ \mathcal{I}(\sigma) \subseteq \mathcal{T} \Longrightarrow (d\sigma)^\top = d^\top\sigma$$

Proof. By $\gg$-induction on the dag:
– $(\dot{k}\sigma :: [dn_i\sigma]_1^n)^\top = (dn_k\sigma :: [dn_i\sigma]_1^n)^\top = (dn_k :: [dn_i\sigma]_1^n)^\top\sigma$ since $min(pt(dn_k)) < k$
– $(\sigma(x) :: [dn_i\sigma]_1^n)^\top = \sigma(x)$ since $\sigma(x) \in \mathcal{T}$ and using Proposition 2.6 ◀

Given this definition of dags and their semantics, we introduce the anti-unification of dags and use it to anti-unify the terms they represent.

2.3.2 Dag Constrained Anti-Unification

In this section we assume that all dags respect the right dependency hypothesis. Since we are not eventually interested in the dags but only in the terms they represent, we consider the anti-unification of dags modulo their semantics.

DEFINITION 2.17 (Dag Anti-Unification). Given d a dag, we say that d_t is a template of d when:

$$\exists\, \sigma,\ d^\top = (d_t\sigma)^\top \ \wedge\ \mathcal{I}(\sigma) \subseteq \mathcal{T}(\Sigma\backslash\overline{\Sigma}, \mathcal{X}, \mathbb{N}^*)$$

An algorithm for dag anti-unification requires to find for each pointer another unique pointer it will be anti-unified with, the corresponding dag terms being then anti-unified *e.g.*,

$$\frac{g(f(\dot{1}), f(\dot{2}))}{g(f(\dot{2}), f(\dot{1}))} \left\| \begin{array}{c|c} a & b \\ c & d \end{array} \right. \longrightarrow \underline{g(f(\dot{1}), f(\dot{2})) \parallel x \mid y} \quad \text{with} \quad \begin{array}{l} \sigma_1 = [x \mapsto a; y \mapsto b] \\ \sigma_2 = [x \mapsto d; y \mapsto c] \end{array}$$

matching term $\dot{1}$ with $\dot{2}$ and $\dot{2}$ with $\dot{1}$. This is equivalent to rename node identifiers in the second dag in a first step and then only anti-unifying pointers and terms with the same identifiers in a second, for example by permuting $\dot{1}$ and $\dot{2}$ we have the following semantics equality

$$\underline{g(f(\dot{2}), f(\dot{1})) \parallel c \mid d} \quad = \quad \underline{g(f(\dot{1}), f(\dot{2})) \parallel d \mid c}$$

Since applying permutations to dags does not change the terms they represent, as stated in Proposition 2.7, we can try different permutations on the input terms before anti-unifying node by node in order to find the more suitable dag representation for the template computation.

DEFINITION 2.18 (Pointer anti-unification). The only rule for pointers anti-unification is the equality rule:

$(EP)\ ctmp(\dot{a},\dot{a}) = \dot{a}$

and we have to compute a common template node by node:

DEFINITION 2.19 (Common template of dags). The common template of dags is computed node by node:

$$ctmp([dn_i]_0^n, [dg_i]_0^n) \ = \ [ctmp(dn_i, dg_i)]_0^n$$

***Remark* 2.2.** As for the tree case, the variable chosen for the (V) rules (see Algorithm 2.5) have to be fresh in order to avoid conflicts when joining the substitutions corresponding to the different dag nodes.

***Remark* 2.3.** The (V) rule does not change at all, this means that in order to anti-unify with a variable the dag nodes have to be in $\mathcal{T}(\Sigma\backslash\overline{\Sigma}, \mathcal{X})$ and thus the substitutions do not contain any pointers.

This definition only allows the anti-unification of dags of the same length, however we can extend any dag with any list of nodes without changing its semantics:

PROPOSITION 2.10 (Dag extension). *Given $[dn_i]_0^n$ a dag:*

$$\forall\, dn_{n+1}, ..., dn_m \in \dot{\mathcal{T}},\ ([dn_i]_0^n)^\top \ = \ ([dn_i]_0^m)^\top$$

Proof. $dn_{n+1}, ..., dn_m$ are never reached in the term computation. ◀

Therefore we can extend the smaller input dags to the length of the longest one using any list of nodes. We will see in Section 2.4.4 how we choose terms to extend the dags. We also have to ensure that the common template is an acyclic graph. Indeed, given two acyclic graph, by only using the (V), (R), (F) and (EP) rules, the pointers in the template can only come from the (EQ) rules, therefore they appear in every dag of the inputs. Since they are acyclic, the template is acyclic. This is no longer the case when we use neutral or switch rules *e.g.*,:

$$\begin{array}{c|c|c} g(f(\dot{1}), f(\dot{2})) & f(\dot{2}) & a \\ \hline g(f(\dot{1}), f(\dot{2})) & b & f(\dot{1}) \end{array} \longrightarrow \begin{array}{c|c|c} g(f(\dot{1}), f(\dot{2})) & sw(e_1, f(\dot{2}), b) & sw(e_2, a, f(\dot{1})) \end{array}$$

This is why we enforced the acyclicity using the right dependency hypothesis. Indeed, if the inputs are right dependency dags, then computing the anti-unifier node by node produces an acyclic graph.

PROPOSITION 2.11 (Template right dependency). *Using any set of rules in (V), (R), (F), (EP), (SW) and all neutral rules on right dependency dags produces a right dependency dag, for all dag nodes dn_i and dg_i we have:*

$$\forall i \in \{0; ...; n\},\ (pt(dn_i) \cup pt(dg_i)) \subseteq \{i+1; ...; n\} \Longrightarrow pt(ctmp(dn_i, dg_i)) \subseteq \{i+1; ...; n\}$$

Proof. By induction, the cases of rules (V), (R), (F) are trivial,

(EP): $\dot{a}$ appears in both dn_i and dg_i, thus $a \in \{i+1; ...; n\}$

(LRN): the neutral element is a term in $\mathcal{T}(\Sigma, \mathcal{X})$ with no pointer, $pt(f(t_2, e_f)) = pt(t_2)$

(SW): sw and e are terms in $\mathcal{T}(\Sigma, \mathcal{X})$, $pt(sw[x \mapsto e, y \mapsto s, z \mapsto t]) = (pt(s) \cup pt(t))$ ◀

Therefore we can define the generic dag constrained anti-unification algorithm, this algorithm has many parameters, such as the dag nodes we chose to extend the short dags, that can be used to refine the anti-unification:

ALGORITHM 2.20 (Generic Dag Constrained Anti-Unification). Given $d_1, ..., d_m$ a set of dags, we define l as $max_i(length(d_i))$ the maximum length, then we use the following non-deterministic algorithm to compute a template of this set of dags:

i) Extend all the dags to the length l with dag nodes (without creating cycles)

ii) Choose a permutation for each extended dag and apply it

iii) Check the right dependency hypothesis, if it is false, apply a new permutation

iv) Anti-unify node by node with rules (V), (R), (EP), (F) and the ones allowed by the theory, *e.g.*, neutral, switches,...

***Remark* 2.4.** The length of a template computed in Algorithm 2.20 is the maximum of the lengths of the input dags. Therefore when the pointers designate the forbidden function calls (see Definition 2.11 and Example 2.8) the number of forbidden function calls in the template is the maximum of the number of forbidden calls in the input. This is why we want to use the dag representation to minimize this number.

We have introduced the general problem of constrained anti-unification, some of its properties, *e.g.*, incompleteness, and the particular case of anti-unification of dags. In Section 2.4, we use the principles of anti-unification with dags to define a constrained anti-unification algorithm for tuple terms in arithmetic with $\sqrt{}$ and / as forbidden functions. This algorithm uses dag representation and all the different axioms of the arithmetic to minimize the number of $\sqrt{}$ calls in the template.

2.4 $\sqrt{}$ AND / ANTI-UNIFICATION

This section introduces a constrained anti-unification of arithmetic expressions defined with $+$, $-$, $\times$, $/$, $\sqrt{}$ where square root and division are the forbidden function symbols. In this algorithm, we consider equality modulo theory of arithmetic and use a dag representation of terms.

2.4.1 Theory of Arithmetic

The terms of arithmetic corresponds to the set $\mathcal{A}$ introduced in definition 1.1. We also denote $\sum_1^n a_i$ the term $a_1 + ... + a_n$ and $\prod_1^n a_i$ the term $a_1 \times ... \times a_n$.

We assume that the terms we are dealing with do not fail, this means that they do not contain division by zero or square roots of negative numbers. We assume that the arithmetic theory contains at least the following axioms that can be used to anti-unify terms. For example binary operators have neutral and absorbing elements:

DEFINITION 2.21 (Neutral and absorbing elements). The following axioms define the neutral and absorbing elements behavior of the constants 0 and 1:

$$\begin{aligned} x + 0 &= x & -0 &= 0 \\ x \times 1 &= x & x \times 0 &= 0 \\ x \ / \ 1 &= x & 0 \ / \ x &= 0 \\ \sqrt{0} &= 0 & \sqrt{1} &= 1 \end{aligned}$$

Since 0 is neutral for $+$ and absorbing for $\times$ and 1 is neutral for $\times$ we can define the n-ary switch introduced in Proposition 2.5:

$$sw_n = \sum_1^n x_i \times y_i$$

with the x_i as switch variables and 0 and 1 as switch elements. Thus the constrained anti-unification modulo such a theory of arithmetic is complete. This theory also contains axioms that allow us to transform the terms, we use the following ones:

DEFINITION 2.22 (Arithmetic axioms). The theory of arithmetic contains the following usual axioms:

$$\begin{aligned} x + y &= y + x & x \times y &= y \times x \\ x + (y + z) &= x + y + z & x \times (y \times z) &= x \times y \times z \end{aligned}$$

$$\frac{x/y}{z} = \frac{x}{y \times z} \qquad \frac{x}{y/z} = \frac{x \times z}{y}$$
$$--x = x \qquad \sqrt{x}\sqrt{x} = x$$
$$x \times (y+z) = xy + xz \qquad x \times \frac{y}{z} = \frac{xy}{z}$$
$$\frac{x}{y} + z = \frac{x+yz}{y} \qquad -\frac{x}{y} = \frac{-x}{y} = \frac{x}{-y}$$
$$-(x \times y) = -x \times y \qquad -(x+y) = -x + -y$$

Given this theory we present in the following section an equivalent representation for terms and the associated dag representation that is used to anti-unify arithmetic terms with the constraint that square roots and divisions are forbidden functions.

2.4.2 Dag Representation

In Section 2.2, we explained how the use of a normal form might make easier the research of a common template of expressions. The theory of arithmetic embeds enough axioms (see Definition 2.22) to be able to transform any expression into an equivalent expression that have the following form:

$$\frac{\sum_{i=1}^{n} a_i \cdot \prod_{j_i=1}^{m_i} \sqrt{b_{j_i}}}{\sum_{i=1}^{n} c_i \cdot \prod_{j_i=1}^{m_i} \sqrt{d_{j_i}}}$$

where none of the a_is or c_is contain any square root or division and where the b_{j_i} and d_{j_i} are also in that form. We will see in Chapter 6 an implementation of this reduction in OCamL. The idea of this representation comes from the vector space representation of algebraic numbers fields that are extensions of the rational numbers $\mathbb{Q}$ (see, for example [Lan94]). We introduce a new representation for tuples of arithmetic expressions that corresponds to this normal form, we directly extends this representation with pointers as introduced in section 2.3:

DEFINITION 2.23 (Dag definition).

$$\begin{aligned} dn ::= \ & \mathsf{PairD}(dn_1, dn_2) \\ & \mid \mathsf{DivD}(dn_1, dn_2) \\ & \mid \mathsf{VectD}([(e_1, [dn_{1,1}, ..., dn_{1,j_1}]), ..., (e_m, [dn_{m,1}, ..., dn_{m,j_m}])]) \\ & \mid \mathsf{ExprD}(e) \\ & \mid \dot{n} \\ d ::= \ & [dn_1, ..., dn_k] \end{aligned}$$

Where e, $e_1, ..., e_m$ are square root and division free arithmetic terms. We define the associated arithmetic term $[\![d]\!]$ of the dag d with the following rules:

DEFINITION 2.24 (Dag associated arithmetic term).

$$\begin{aligned} & [\![\mathsf{PairD}(d_{01}, d_{02}) :: [d_i]_1^n]\!] = ([\![d_{01} :: [d_i]_1^n]\!], [\![d_{02} :: [d_i]_1^n]\!]) \\ & [\![\mathsf{DivD}(d_{01}, d_{02}) :: [d_i]_1^n]\!] = [\![d_{01} :: [d_i]_1^n]\!] \ / \ [\![d_{02} :: [d_i]_1^n]\!] \\ & [\![\mathsf{VectD}([(e_j, [sq_{jk}]_1^{p_j})]_1^m) :: [d_i]_1^n]\!] = \sum_{j=1}^{m} e_j \cdot \prod_{k=1}^{p_j} \sqrt{[\![sq_{jk} :: [d_i]_1^n]\!]} \\ & [\![\mathsf{ExprD}(e) :: [d_i]_1^n]\!] = e \\ & [\![\dot{k} :: [d_i]_1^n]\!] = [\![d_k :: [d_i]_1^n]\!] \end{aligned}$$

Therefore any tuple of expressions can be represented by such a dag, we still ensure that the dag is acyclic by using the right dependencies hypothesis.

EXAMPLE 2.10 (Dag representation).
$(\frac{a_1}{a_2+a_3\cdot\sqrt{d}}, b\cdot\sqrt{d}\cdot\sqrt{c_1+c_2\cdot\sqrt{d}})$ is represented by $\begin{array}{c|c|c} (\frac{a_1}{a_2+a_3\cdot\sqrt{2}}, b\cdot\sqrt{2}\cdot\sqrt{\dot{1}}) & c_1+c_2\cdot\sqrt{2} & d \end{array}$

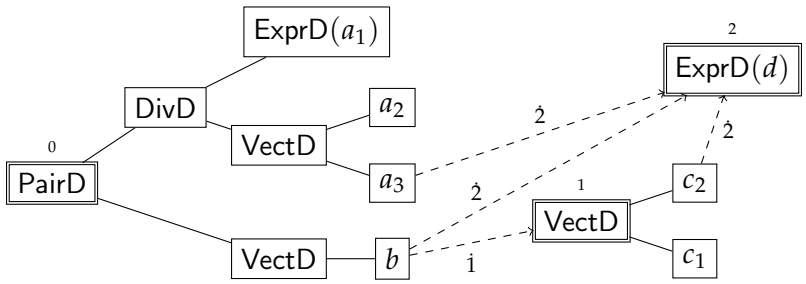


Figure 2.2: Dag representation

In section 2.3 we already introduced the semantics of a dag as the corresponding tree term. The associated arithmetic term is a new layer of interpretation for these terms.

EXAMPLE 2.11 (Dag interpretations).
$[\mathsf{PairD}(\mathsf{VectD}(a,[\dot{1}]),\mathsf{ExpD}(b));\mathsf{ExpD}(c)]^{\top} = \mathsf{PairD}(\mathsf{VectD}(a,[\mathsf{ExpD}(c)]),\mathsf{ExpD}(b))$
$[\![[\mathsf{VectD}(a,[\dot{1}]);\mathsf{ExpD}(c)]]\!] = a\cdot\sqrt{c} = [\![[\mathsf{VectD}(a,[\mathsf{ExpD}(c)])]]\!]$

The equality of the tree term semantics implies the equality of the arithmetic interpretation:

PROPOSITION 2.12 (Tree term semantics and arithmetic term semantics). *For every dag d_1 and d_2, the following implication holds:*

$$d_1^{\top} = d_2^{\top} \implies [\![d_1]\!] = [\![d_2]\!]$$

Proof. We prove by induction that $[\![d_1]\!] = [\![[d_1^{\top}]]\!]$. ◀

In this section, we are interested in anti-unifying modulo this interpretation, indeed our goal is to use the dag representation to compute a constrained anti-unifier of arithmetic terms. Therefore we consider the anti-unification modulo this interpretation. The substitutions are only allowed to replace the part of the dag that contains the arithmetic terms, this means that all the dag symbols are forbidden:

DEFINITION 2.25 (Arithmetic dag constrained anti-unification). Given two dags d_1 and d_2, we aim at computing a dag d, such that it exists two substitution σ_1 and σ_2 from $\mathcal{X}$ to $\mathcal{A}^{\surd}$ such that:

$$[\![d]\!]\sigma_1 =_{\mathcal{A}} [\![d_1]\!] \quad \text{and} \quad [\![d]\!]\sigma_2 =_{\mathcal{A}} [\![d_2]\!]$$

Where $\mathcal{A}^{\surd}$ is the set of arithmetic terms that are square root and division free and $=_{\mathcal{A}}$ the equality modulo the arithmetic theory.

In fact, we only deal with a subset of such dags. In order to focus on minimizing the number of square roots on distinct expressions, only the VectD constructors contain pointers to other nodes and they contains only pointers. Thus the other nodes represent the different square roots of the expression and only the root node can embed PairD constructors.

DEFINITION 2.26 (Well formed arithmetic dags). A dag $[d_i]_0^n$ is *well formed* if

- it is a right dependency dag, $\forall i, pt(d_i) \subseteq \{i+1, ..., n\}$
- $\mathsf{PairD}(a_1, a_2) \lll d_i \implies i = 0$
- $a \ll \mathsf{VectD}(l) \implies \exists k,\ a = \dot{k}$
- $\dot{k} \lll d_i \implies \exists l,\ \dot{k} \ll \mathsf{VectD}(l) \lll d_i$

where $\ll$ is strict and $\lll$ the large sub-term relation on dag nodes.

For clarity and concision, in the examples, we prefer the array representation of dags using the semantics even if the algorithm is described with dag constructors. The expressions that are leaves of these dags are already square root or division free, therefore, the generalization of two leaves is a variable. Thus the $\{\sqrt{}, /\}$-anti-unification consist of computing a common template of the dag structure.

2.4.3 $\{\sqrt{}, /\}$-Anti-Unification

The VectD constructor is interpreted as a sum of products and both addition and multiplication are associative and commutative:

PROPOSITION 2.13 (VectD permutations). *Given* $\mathsf{VectD}([(e_j, [sq_{jk}]_1^{p_j})]_0^m)$ *a dag node, for any permutations* $\tau, \tau_1, ...\tau_m$*:*

$$\forall dn_1, ..., dn_n,$$
$$(\mathsf{VectD}([(e_j, [sq_{jk}]_1^{p_j})]_1^m) :: [dn_i]_1^n)^\top = (\mathsf{VectD}([e_{\tau(j)}, [sq_{\tau(j)\tau_j(k)}]_1^{p_{\tau(j)}})]_1^m) :: [dn_i]_1^n)^\top$$

This means that both level of lists do not need to be ordered, thus we define the range of a VectD node:

DEFINITION 2.27 (VectD Range). Given $\mathsf{VectD}([(e_j, [sq_{jk}]_1^{p_j})]_0^m)$ a dag node, we define its *Range* as the following set of set:

$$Range([(e_j, [sq_{jk}]_1^{p_j})]_0^m) = \{\{sq_{11}, ..., sq_{1p_1}\}, ..., \{sq_{m1}, ..., sq_{mp_m}\}\}$$

Since all the e_j are square roots and division free we can construct templates for any VectD node by only using the *Range*:

PROPOSITION 2.14 (VectD Range). *Given* $\mathsf{VectD}([e_i, l_i]_1^m)$ *and* $\mathsf{VectD}([x_i, l'_i]_1^n)$ *two dag nodes (where for all i,* x_i *is a variable) then if* $Range([e_i, l_i]_1^m) \subseteq Range([x_i, l'_i]_1^n)$ *then* $\mathsf{VectD}([x_i, l'_i]_1^n)$ *is a constrained template of* $\mathsf{VectD}([e_i, l_i]_1^m)$

Proof. We use the substitution σ:

$$\sigma(x_i) = \begin{cases} e_j & \text{if } \{sq \mid sq \in l'_i\} = \{sq \mid sq \in l_j\} \\ 0 & \text{if not} \end{cases}$$

◀

Following the principles of dag anti-unification we introduced in Section 2.3, we can define the arithmetic constrained dag anti-unification.

DEFINITION 2.28 (Arithmetic dag anti-unification). The anti-unification node by node is almost straightforward, we describe the different rules:

- PairD: the expressions to anti-unify are supposed to have the same type, therefore they have the same PairD structure, the only rule is:

(P) $ctmp(\mathsf{PairD}(d_{11}, d_{12}), \mathsf{PairD}(d_{21}, d_{22})) = \mathsf{PairD}(ctmp(d_{11}, d_{21}), ctmp(d_{12}, d_{22}))$

- DivD: two rules are introduced, depending on the head symbols of both nodes:

(D) $ctmp(\mathsf{DivD}(d_{11}, d_{12}), \mathsf{DivD}(d_{21}, d_{22})) = \mathsf{DivD}(ctmp(d_{11}, d_{21}), ctmp(d_{12}, d_{22}))$
when t head symbol is not DivD:
(DI) $ctmp(\mathsf{DivD}(d_{11}, d_{12}), t) = \mathsf{DivD}(ctmp(d_{11}, t), ctmp(d_{12}, ExprD(1)))$

- ExprD: the generalization of square root and division free expressions is a variable:

(E) $ctmp(\mathsf{ExprD}(e_1), \mathsf{ExprD}(e_2)) = \mathsf{ExprD}(x)$
However, as in usual computation of least general template [AEMO08, KLV11], we record the already anti-unified terms and reuse the variable if possible.

- VectD: we use the template construction using the range from Proposition 2.14:

(VR) $ctmp(\mathsf{VectD}(lv_1), \mathsf{VectD}(lv_2)) = \mathsf{VectD}([(x_i, l_i)])$
with $Range([(x_i, l_i)]) = Range(lv_1) \cup Range(lv_2)$
$(V0)$ $ctmp(\mathsf{VectD}(l), \mathsf{ExprD}(e)) = \mathsf{VectD}([(x_i, l_i)])$
with $Range([(x_i, l_i)]) = Range(l) \cup \{\emptyset\}$

- $\dot{n}$: pointers only appear as sub-term of VectD nodes therefore we do not need any rule for pointer since the (V0) and (VR) do not make recursive calls.

PROPOSITION 2.15 (Anti-unification correctness). *The rules introduced in Definition 2.28 compute a $\sqrt{}$, /-constrained anti-unifier modulo arithmetic theory.*

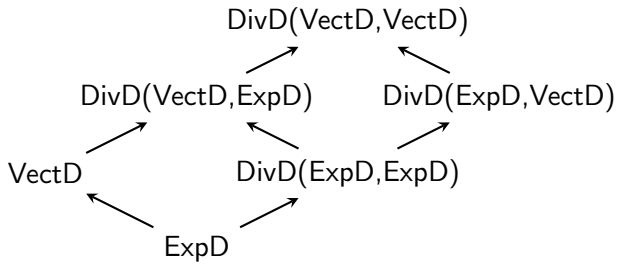
Proof. By induction, correctness of rules (P), (D) and (E) is trivial.

(DI) 1 is a right neutral element for division: $\forall e, e = e/1$

(VR) using Proposition 2.14

$(V0)$ $(\mathsf{VectD}([(0, l_1), ..., (e, []), ..., (0, l_n)]) :: l)^{\top} = e \times \prod_{x \in \emptyset} \sqrt{x} = e$ ◀

This template computation can be represented by the following lattice on dag nodes, the anti-unifier being the least upper bound where every leaf is abstracted with a variable.



with:
- $\mathsf{ExpD}(a) < \mathsf{VectD}(l)$ iff $\emptyset \in \mathit{Range}(l)$
- $\mathsf{VectD}(l) < \mathsf{VectD}(q)$ iff $\mathit{Range}(l) \subseteq \mathit{Range}(q)$

Figure 2.3: Order on dag nodes (DNO)

In Section 2.3 we introduced the principle of extending the smallest dags in order to anti-unify dags of equal length. Although it is easy to extend the smallest dag with constant non-pointed elements (*e.g.*, 0 or 1) before the anti-unification, we now present how extending dags with tailored elements allow us to improve the computed template.

2.4.4 Dag Extension

In order to anti-unify dags node by node we first need to extend the smallest dags so that all the dags we want to anti-unify have the same length. A simple solution would be to extend these dags with non-pointed elements equals to 0 or 1 or any other constant previously chosen. However this is not very efficient and by using more nodes we can change the dag representation of the expression in order to upgrade the anti-unification. Therefore, in a first step we extend the dags with *undefined* elements (#), these elements are not referenced by pointers and are replaced later in the anti-unification process. Replacing undefined elements with appropriate expressions in the dags extension process have two different objectives:

i) Compute more compact templates by breaking unnecessary sharing introduced by the dag representation and avoid the use of the switch operation:

EXAMPLE 2.12 (Node duplication). We have the following semantics equalities

$$\frac{\sqrt{1},\sqrt{2} \;\|\; b \;|\; a}{\sqrt{1},\sqrt{1} \;\|\; c \;|\;} = \frac{\sqrt{1},\sqrt{2} \;\|\; b \;|\; a}{\sqrt{1},\sqrt{1} \;\|\; c \;|\; \#} = \frac{\sqrt{1},\sqrt{2} \;\|\; b \;|\; a}{\sqrt{1},\sqrt{1} \;\|\; c \;|\; c} = \frac{\sqrt{1},\sqrt{2} \;\|\; b \;|\; a}{\sqrt{1},\sqrt{2} \;\|\; c \;|\; c}$$

thus we can use the following constrained template: $\underline{\sqrt{1},\sqrt{2} \;\|\; x \;|\; y}$

ii) Avoid the use of new fresh variables when expressions are already identical variables:

EXAMPLE 2.13 (Avoiding renaming).

$$\frac{\sqrt{1} \;\|\; x+\sqrt{2} \;|\; y}{0 \;\|\; \# \;|\; \#} = \frac{\sqrt{1} \;\|\; x+\sqrt{2} \;|\; y}{0 \;\|\; x+\sqrt{2} \;|\; y}$$

and we have the following constrained template: $\underline{z_1 \cdot \sqrt{1} \;\|\; x+\sqrt{2} \;|\; y}$

The node duplication introduced in i) relies on the following transformations:

PROPOSITION 2.16 (Node and pointer duplication transformations). *The following transformations preserve the associated term:*

(ND) *when* $k < l$: $[d_0, ..., d_{k-1}, \#, d_{k+1}, ..., d_n] \longrightarrow [d_0, ..., d_{k-1}, d_l, d_{k+1}, ..., d_n]$

(PD) *when* $k < l$ *and* $d_k = d_l$: $[d_i]_0^n \longrightarrow [[\dot{l} \mapsto \dot{k}]d_0, ..., [\dot{l} \mapsto \dot{k}]d_{k-1}, d_k, ..., d_n]$

where $[\dot{k}/\dot{l}]d$ *is the node* d *where some occurrences of* $\dot{l}$ *are replaced by* $\dot{k}$*, the equality in* (PD) *condition being syntactic.*

Proof.

(ND) since the k-th element of the list is undefined no nodes has a pointer to $\dot{k}$.

(PD) by induction, using the order defined in Definition 2.14 the pointer case being:

$$\begin{aligned}(\dot{l} :: [[\dot{k}/\dot{l}]d_1, ..., [\dot{k}/\dot{l}]d_{k-1}, d_k, ..., d_n])^\top &= (d_l :: [[\dot{k}/\dot{l}]d_1, ..., [\dot{k}/\dot{l}]d_{k-1}, d_k, ..., d_n])^\top \\ &= (d_l :: [d_i]_1^n)^\top && \text{by induction hypothesis} \\ &= (d_k :: [d_i]_1^n)^\top && \text{since } d_k = d_l\end{aligned}$$

◀

PROPOSITION 2.17 (Right dependency). (ND) *and* (PD) *preserve the right dependency condition:*

Proof.

(ND) since $k < l$, we have $pt(d_l) \subseteq \{l+1, ..., n\} \subseteq \{k+1, ..., n\}$

(PD) $\forall i < k, pt([\dot{l} \mapsto \dot{k}]d_i) \subseteq pt(d_i) \cup \{k\} \subseteq \{i+1, ..., n\}$ ◀

However avoiding renaming of square roots is more complicated. Indeed, we assume that all the expressions we want to anti-unify are correct in the context they will be evaluated in, *i.e.*, arguments of square roots are positive and there is no division by 0. But we do not assume that this context is the same for all the expressions, $\sqrt{x}$ appearing in one expression does not mean that x is positive in every expression. Moreover, we do not consider that 0 is absorbing for failure, $0.\sqrt{-1}$ fails. Therefore we can not replace undefined elements by any term but only by terms that are known to be positive. For example, in order to use $z_1 \cdot \sqrt{x + \sqrt{y}}$ as a template of 0 as in Example 2.13, we have to be sure that y and $x + \sqrt{y}$ are positive

Therefore, when we replace the undefined elements, we have to do it with expressions that are positive in the context the expression is supposed to be evaluated. This problem does not appear in node duplication since replacement of undefined elements is done by another node of the same dag which is already argument of a square root of the same expression and therefore positive, *e.g.*, in Example 2.12, we already know that c is positive in the context corresponding to the second dag.

Given this constraint, we introduce, as a parameter of the algorithm, a set of expressions that are known to be positive in all these contexts: $\mathcal{P}os$. Given such a set, we can use the following rule:

PROPOSITION 2.18 (Positive element replacement). *The rule replacing an undefined element by a positive element preserves the associated term and :*

(PR) *when* $[\![p :: [d_1, ..., d_n]]\!] \in \mathcal{P}os$ *and* $pt(p) \in \{i+1, ..., n\}$ *and* $d_i = \#$:

$$[d_0, ..., d_{i-1}, \#, d_{i+1}, ..., d_n] \longrightarrow [d_0, ..., d_{i-1}, p, d_{i+1}, ..., d_n]$$

In practice, we only use this rule when it allows to avoid renaming. When given a set of dags all the i-th nodes are either equal to the same node, p, or undefined, this ensures that $pt(p) \in \{i+1, ..., n\}$ since p is the i-th node of at least one dag. Therefore the anti-unifier of the i-th node is p itself and we avoid to create a new square root.

If none of these rules can be applied, we can always replace undefined elements with a positive numerical constant, *e.g.,* 0 or 1. Therefore we have 3 different choices in order to replace undefined elements:

- another square root of the same dag, using the rules defined in Proposition 2.16,
- an expression from $\mathcal{P}os$ if all i-th nodes are either undefined or equal to this expression, using rule PR defined in Proposition 2.18,
- a positive constant:
  when $c \geq 0$, $[d_0, ..., d_{i-1}, \#, d_{i+1}, ..., , d_n] \longrightarrow [d_0, ..., d_{i-1}, \mathsf{Expr}(c), d_{i+1}, ..., d_n]$

and allow the use of the new identity rule for anti-unification:

DEFINITION 2.29 (Identity Rule).

(EI) $ctmp(e, e) = e$

We will see in Section 3.7.2 how $\mathcal{P}os$ can be constructed in the context of the program transformation. Given this undefined element replacement, we can now define how, by trying different permutations as introduced in Section 2.3, we can compute a set of $\{\sqrt{}, /\}$-constrained templates for any set of arithmetic expressions:

ALGORITHM 2.30 (Arithmetic expression $\{\sqrt{}, /\}$-constrained anti-unification). The following algorithm computes a set templates of a set of expressions:

i) Transform every expression into its dag representation.

ii) Extend all dags to the same length with undefined elements.

iii) Apply a permutation on the dag nodes identifiers with respect to right dependency as introduced in Definition 2.16.

iv) Replace undefined elements using the different possibilities previously described.

v) Compute the $\{\sqrt{}, /\}$-template node by node using rules (P), (D), (DI), (E), (EI), (VR) and (V0).

By trying different permutations and different undefined elements replacements, we can compute a set of $\{\sqrt{}, /\}$-templates of the input expressions.

THEOREM 2.19 (Anti-unification correctness). *Any template computed with the Algorithm 2.30 computes a valid constrained template modulo the arithmetic theory*

Proof. The fourth first step of the algorithm preserve the semantics of the dags and the anti-unification node by node is proven to be correct in Proposition 2.15. ◀

Conclusion This section has defined the problem of constrained anti-unification and proposed an algorithm to solve it. This constrained anti-unification will be used in the program transformation removing square roots and divisions from programs. It allows us to extract the operations that we do not want to see anymore in a term in order to eliminate them later in some Boolean expressions. This elimination on Boolean expressions has a huge complexity depending on the number of square roots and divisions that appear in the terms. This is the reason why we focused on minimizing that number in the template we compute. We really dealt with this main goal since, using the axioms of the arithmetic theory and the dag representation, we managed to produce templates whose number of square roots is the maximum of the number of square roots that can be found in one of the input (the length of the template as dag is the maximum of the input dags lengths).

3 TRANSFORMATION OF PROGRAMS

IN THIS CHAPTER, A COMPLETE ALGORITHM for transformation of programs is presented. It uses the algorithm introduced in Chapter 1 for the Boolean expressions and the one defined in Chapter 2 for variable definition transformation. In order to discuss the transformation of programs, we need to introduce the language these programs are expressed in. In this chapter we present this language, defining its syntax, type system and semantics. We also present the different subsets of this language that this transformation targets. We introduce some general properties of this language and some partial transformations and use them to define the global program transformation algorithm.

3.1 LANGUAGE

This program transformation aims at transforming programs used in embedded systems such as the ones presented in [NMD12, MBMD09]. These programs do not need all the features that a Turing complete language provides. Therefore we can restrict the source language of our transformation to *straight line programs*. The language this transformation deals with is a typed functional language that contains numerical and Boolean constants $\mathcal{C}$, variables $\mathcal{X}$ and variable definitions (let in instructions), tests (if then else), pairs and the usual arithmetic operators $+, -, \times$ (we also use . instead of $\times$),$/$, $\sqrt{}$, the comparisons $=, =, >, \geq, <, \leq$ and Boolean operators ($\wedge, \vee, \neg$).

DEFINITION 3.1 (Syntax of the language).

$$
\begin{array}{llll}
\mathsf{Prog} := & \mathcal{C} & \mid \mathsf{fst\ Prog} & \mid \mathsf{Prog\ op\ Prog} \\
 & \mid \mathcal{X} & \mid \mathsf{snd\ Prog} & \mid \mathsf{if\ Prog\ then\ Prog\ else\ Prog} \\
 & \mid \mathsf{uop\ Prog} & \mid \mathsf{(Prog,\ Prog)} & \mid \mathsf{let}\ \mathcal{V} = \mathsf{Prog\ in\ Prog}
\end{array}
$$

where:

$$
\begin{array}{rcl}
\mathcal{V} & = & \mathcal{X} \mid (\mathcal{V}, \mathcal{V}) \\
\mathcal{C} & \subseteq & \mathbb{R} \cup \{\mathit{True},\ \mathit{False}\} \\
\mathsf{op} & \in & \{+, \times, /, =, \neq, >, \geq, <, \leq, \wedge, \vee\} \\
\mathsf{uop} & \in & \{\sqrt{}, -, \neg\}
\end{array}
$$

In a variable definition, *e.g.*, $\mathsf{let\ x = b\ in\ sc}$, we call b the *body* and sc the *scope* of this definition. The $\mathcal{V}$ set is used to make multiple variable definitions, we assume that in a multi-variable v, all the variable in it are different. In order to complete the description of the language we introduce the type system, as usual we use a typing environment Γ that associates to every free variable its type.

DEFINITION 3.2 (Type system). The types of the programs are represented by the following set:

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

The rules are the usual ones for such a language:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash c : \mathbb{R}}\ c \in \mathbb{R}$$

$$\frac{}{\Gamma \vdash False : \mathbb{B}} \qquad \frac{}{\Gamma \vdash True : \mathbb{B}}$$

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

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

$$\frac{\Gamma \vdash e_1 : \mathbb{R} \quad \Gamma \vdash e_2 : \mathbb{R}}{\Gamma \vdash e_1\ op\ e_2 : \mathbb{B}}\ op \in \{=, \neq, >, <, \geq, \leq\} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2}$$

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

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \oplus (\mathsf{v}, T_1) \vdash e_2 : T_2}{\Gamma \vdash \mathsf{let\ v} = e_1\ \mathsf{in}\ e_2 : T_2} \qquad \frac{\Gamma \vdash f : \mathbb{B} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{if}\ f\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T}$$

where:

$$\Gamma \oplus ((\mathsf{v}_1, \mathsf{v}_2),\ T_1 \times T_2) = (\Gamma \oplus (\mathsf{v}_1, T_1)) \oplus (\mathsf{v}_2, T_2)$$
$$\Gamma \oplus (\mathsf{x}, T) = \Gamma, (\mathsf{x} : T) \text{ when } \mathsf{x} \in \mathcal{X}$$

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, $Ty_\Gamma(\mathsf{p})$, that returns either a type or an undefined value ($\mathbb{U}$) if the program has no valid type in the given environment. We also define the predicate $wt(p)$ stating that a program is well typed:

DEFINITION 3.3 (Well typed program).

$$wt(\mathsf{p}) \iff \exists\, \Gamma,\ Ty_\Gamma(\mathsf{p}) \neq \mathbb{U}$$

Then we define the denotational semantics of a program in the language, using an environment *Env* that associates a value to every variable. It is the usual semantics of a functional language using real numbers computation. The *Fail* value corresponds to the semantics of square roots of negative numbers, divisions by zero and type errors.

DEFINITION 3.4 (Denotational semantics of the language). The semantics is defined by the following induction rules:

$$\frac{}{Env \vdash [\![\mathsf{c}]\!]^{\mathbb{R}}{=}\mathsf{c}}$$

$$\frac{}{Env,(\mathsf{x},v) \vdash [\![\mathsf{x}]\!]^{\mathbb{R}}{=}v}$$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{R}}{=}Fail}{Env \vdash [\![\mathsf{uop}\ \mathsf{E}]\!]^{\mathbb{R}}{=}Fail}$$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{R}}{=}v}{Env \vdash [\![\mathsf{uop}\ \mathsf{E}]\!]^{\mathbb{R}}{=}uop(v)} \qquad op \in \{-,\neg\} \wedge v \neq Fail$$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{R}}{=}v}{Env \vdash [\![\sqrt{\mathsf{E}}]\!]^{\mathbb{R}}{=}Fail} \qquad v < 0 \vee v = Fail$$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{R}}{=}v}{Env \vdash [\![\sqrt{\mathsf{E}}]\!]^{\mathbb{R}}{=}\sqrt{v}} \qquad v \geq 0$$

$$\frac{Env \vdash [\![\mathsf{E}_1]\!]^{\mathbb{R}}{=}v_1 \quad Env \vdash [\![\mathsf{E}_2]\!]^{\mathbb{R}}{=}v_2}{Env \vdash [\![\mathsf{E}_1\ \mathsf{op}\ \mathsf{E}_2]\!]^{\mathbb{R}}{=}v_1\ op\ v_2} \qquad v_1 \neq Fail \wedge v_2 \neq Fail \wedge (op = / \Rightarrow v_2 \neq 0)$$

$$\frac{Env \vdash [\![\mathsf{E}_1]\!]^{\mathbb{R}}{=}v_1 \quad Env \vdash [\![\mathsf{E}_2]\!]^{\mathbb{R}}{=}v_2}{Env \vdash [\![\mathsf{E}_1\ \mathsf{op}\ \mathsf{E}_2]\!]^{\mathbb{R}}{=}Fail} \qquad v_1 = Fail \vee v_2 = Fail \vee (op = / \wedge v_2 = 0)$$

$$\frac{Env \vdash [\![\mathsf{E}_1]\!]^{\mathbb{R}}{=}v_1 \quad Env \vdash [\![\mathsf{E}_2]\!]^{\mathbb{R}}{=}v_2}{Env \vdash [\![(\mathsf{E}_1,\mathsf{E}_2)]\!]^{\mathbb{R}}{=}Fail} \qquad v_1 = Fail \vee v_2 = Fail$$

$$\frac{Env \vdash [\![\mathsf{E}_1]\!]^{\mathbb{R}}{=}v_1 \quad Env \vdash [\![\mathsf{E}_2]\!]^{\mathbb{R}}{=}v_2}{Env \vdash [\![(\mathsf{E}_1,\mathsf{E}_2)]\!]^{\mathbb{R}}{=}(v_1,v_2)} \qquad v_1 \neq Fail \wedge v_2 \neq Fail$$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{R}}{=}(v_1,v_2)}{Env \vdash [\![\mathsf{fst}(\mathsf{E})]\!]^{\mathbb{R}}{=}v_1} \qquad v_1 \neq Fail \wedge v_2 \neq Fail$$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{R}}{=}(v_1,v_2)}{Env \vdash [\![\mathsf{fst}(\mathsf{E})]\!]^{\mathbb{R}}{=}Fail} \qquad v_1 = Fail \vee v_2 = Fail$$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{R}}{=}(v_1,v_2)}{Env \vdash [\![\mathsf{snd}(\mathsf{E})]\!]^{\mathbb{R}}{=}v_2} \qquad v_1 \neq Fail \wedge v_2 \neq Fail$$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{R}}{=}(v_1,v_2)}{Env \vdash [\![\mathsf{snd}(\mathsf{E})]\!]^{\mathbb{R}}{=}Fail} \qquad v_1 = Fail \vee v_2 = Fail$$

$$\frac{Env \vdash [\![\mathsf{E}_1]\!]^{\mathbb{R}}{=}v_1 \quad Env{\oplus}(\mathsf{v},v_1) \vdash [\![\mathsf{E}_2]\!]^{\mathbb{R}}{=}v_2}{Env \vdash [\![\mathsf{let}\ \mathsf{v} = \mathsf{E}_1\ \mathsf{in}\ \mathsf{E}_2]\!]^{\mathbb{R}}{=}v_2} \qquad v_1 \neq Fail$$

$$\frac{Env \vdash [\![\mathsf{E}_1]\!]^{\mathbb{R}}{=}v_1}{Env \vdash [\![\mathsf{let}\ \mathsf{v} = \mathsf{E}_1\ \mathsf{in}\ \mathsf{E}_2]\!]^{\mathbb{R}}{=}Fail} \qquad v_1 = Fail$$

$$\frac{Env \vdash [\![\mathsf{F}]\!]^{\mathbb{R}}{=}True \quad Env \vdash [\![\mathsf{E}_1]\!]^{\mathbb{R}}{=}v_1}{Env \vdash [\![\mathsf{if}\ \mathsf{F}\ \mathsf{then}\ \mathsf{E}_1\ \mathsf{else}\ \mathsf{E}_2]\!]^{\mathbb{R}}{=}v_1}$$

$$\frac{Env \vdash [\![\mathsf{F}]\!]^{\mathbb{R}}{=}False \quad Env \vdash [\![\mathsf{E}_2]\!]^{\mathbb{R}}{=}v_2}{Env \vdash [\![\mathsf{if}\ \mathsf{F}\ \mathsf{then}\ \mathsf{E}_1\ \mathsf{else}\ \mathsf{E}_2]\!]^{\mathbb{R}}{=}v_2}$$

$$\frac{Env \vdash [\![\mathsf{F}]\!]^{\mathbb{R}}{=}Fail}{Env \vdash [\![\mathsf{If}\ \mathsf{F}\ \mathsf{then}\ \mathsf{E}_1\ \mathsf{else}\ \mathsf{E}_2]\!]^{\mathbb{R}}{=}Fail}$$

where:

$$\Gamma \oplus ((\mathsf{x}_1,\mathsf{x}_2),\ (v_1,v_2)) = (\Gamma \oplus (\mathsf{x}_1,v_1)) \oplus (\mathsf{x}_2,v_2)$$
$$\Gamma \oplus (\mathsf{x},v) = \Gamma,(\mathsf{x},v) \text{ when } \mathsf{x} \in \mathcal{X}$$

The semantics is also *Fail* if $\oplus$ can not be computed, *e.g.*, $Env \oplus ((x,y),3)$.

We now denote $[\![\mathsf{P}]\!]_{Env}$ the abstract semantics of P in an environment Env (*i.e.*, the v such that $Env \vdash [\![\mathsf{P}]\!]^{\mathbb{R}} = v$). In order to define the transformation we need the notion of substitution. The substitution of x by e in p is denoted $p[x \mapsto e]$, it avoids variable capture and thus respects the following property:

PROPOSITION 3.1 (Substitution specification).

$$\forall Env,\ x \in \mathcal{X},\ \mathsf{e}, \mathsf{p} \in \mathsf{Prog}^2,\ [\![\mathsf{p}[\mathsf{x} \mapsto \mathsf{e}]]\!]_{Env} = [\![\mathsf{p}]\!]_{Env, x:[\![\mathsf{e}]\!]_{Env}}$$

We also define the substitution of multi-variable which is required to verify the following property:

PROPOSITION 3.2 (Multi-variable substitution).

$$\forall Env,\ \mathsf{v} \in \mathcal{V},\ \mathsf{e}, \mathsf{p} \in \mathsf{Prog}^2,\ [\![\mathsf{p}[\mathsf{v} \mapsto \mathsf{e}]]\!]_{Env} = [\![\mathsf{let\ v = e\ in\ p}]\!]_{Env}$$

EXAMPLE 3.1 (Substitutions).

$$\begin{array}{rlrl} (\mathsf{x} + \mathsf{y})\ [\mathsf{x} \mapsto \mathsf{z}] & = \mathsf{z} + \mathsf{y} & (\mathsf{x} + \mathsf{y})\ [(\mathsf{x},\mathsf{y}) \mapsto (\mathsf{z},\mathsf{t})] & = \mathsf{z} + \mathsf{t} \\ (\mathsf{x} + \mathsf{y})\ [(\mathsf{x},\mathsf{y}) \mapsto (\mathsf{y},\mathsf{t})] & = \mathsf{y} + \mathsf{t} & (\mathsf{x} + \mathsf{y})\ [(\mathsf{x},\mathsf{y}) \mapsto \mathsf{e}] & = \mathsf{fst}(\mathsf{e}) + \mathsf{snd}(\mathsf{e}) \end{array}$$

We also denote $FV(p)$ the set of the free variables in p. The language being defined, we now precise some subtypes of Prog, that represent restricted syntactic forms.

3.2 PROGRAM SUBTYPES

The first subtype we define is the subtype our transformation applies on.

3.2.1 Normalized language

The normalized language is a subtype of the Prog type where arithmetic and Boolean expressions do not contain tests or variable definitions. For example:
$(\mathsf{let\ v = 3\ in\ v + 4}) + 8$ or $(\mathsf{if\ x > 0\ then\ 3\ else\ 5}) < 4$ are not allowed in P.

This type has the following definition, a transformation of any program into this form will be introduced in Section 3.5.2.

DEFINITION 3.5 (Expressions and programs normal form).
The unary expressions in E_u are built with operators and the set E extends E_u with pairs of unary expressions.

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

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

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

In such a program, every time we meet an arithmetic or Boolean operator, we know there is neither definitions nor tests inside its arguments. Moreover, fst and snd constructs can only be applied to variables (in well typed programs) and thus not contain any square roots or divisions in their arguments. Given such a normalized program, we call *final numerical expressions* the expressions that the program may return:

DEFINITION 3.6 (Final numerical expressions). The final numerical expressions are defined by the following algorithm fne:

$fne(\mathsf{let\ x = b\ in\ sc}) = fne(\mathsf{sc})$
$fne(\mathsf{if\ F\ then\ p1\ else\ p2}) = fne(\mathsf{p1}) \cup fne(\mathsf{p2})$
$fne((\mathsf{e1,e2})) = fne(\mathsf{e1}) \cup fne(\mathsf{e2})$
when e is in E_u then,
if it exists Γ such that $Ty_\Gamma(\mathsf{e}) = \mathbb{R}$ then $fne(\mathsf{e}) = \{\mathsf{e}\}$ else $fne(\mathsf{e}) = \emptyset$

These final numerical expressions are the one where we are not able to eliminate the square roots.

3.2.2 Target language

Now we present 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.*, in the program $\sqrt{2}$ we will have to return a rounded value of $\sqrt{2}$) but we are able to remove them from all the Boolean computations, in particular the Boolean expressions of the tests but also the variables they depend on.

EXAMPLE 3.2 (Targeted programs).

$\mathsf{let\ x} = a - b\ \mathsf{in\ if\ x} + c \times d > 0\ \mathsf{then}\ \sqrt{2}\ \mathsf{else}\ 0$

is a valid output whereas:

$\mathsf{let\ x} = a - b\ \mathsf{in\ if\ x} + \sqrt{c \times d} > 0\ \mathsf{then}\ \sqrt{2}\ \mathsf{else}\ 0$

and

$\mathsf{let\ x} = \sqrt{a - b}\ \mathsf{in\ if\ x} + c \times d > 0\ \mathsf{then}\ \sqrt{2}\ \mathsf{else}\ 0$

are not.

We define new subtypes of expressions and programs to specify this targeted language:

DEFINITION 3.7 (Targeted program subtypes). Our goal is to define programs where square roots and divisions are only allowed in the final numerical expressions. To formally define this subtype we define other subtypes corresponding to different expressions and programs, depending on which operators are allowed in different parts of the program:

- the different sets of operators:
  - Numerical unary and binary operators sets with or without square root and division operators:
    - $Nuop_\surd = \{-, \surd\}$ — $Nuop = \{-\}$
    - $Nbop_/ = \{+, \times, /\}$ — $Nbop = \{+, \times\}$
  - The Boolean unary and binary and the comparison operators:
    - $Buop = \{\neg\}$ — $Cbop = \{=, \neq, <, >, \leq, \neq\}$
    - $Bbop = \{\wedge, \vee\}$
- the different sets of numerical expressions regarding if square roots and divisions are allowed or not:

- $\mathsf{N} := \mathit{Nuop}\ \mathsf{N} \mid \mathsf{N}\ \mathit{Nbop}\ \mathsf{N} \mid \mathsf{fst}\ \mathsf{N} \mid \mathsf{snd}\ \mathsf{N} \mid \mathcal{X} \mid \mathcal{C}$
- $\mathsf{N}_{\surd,/} := \mathit{Nuop}_{\surd}\ \mathsf{N}_{\surd,/} \mid \mathsf{N}_{\surd,/}\ \mathit{Nbop}_{/}\ \mathsf{N}_{\surd,/} \mid \mathsf{fst}\ \mathsf{N}_{\surd,/} \mid \mathsf{snd}\ \mathsf{N}_{\surd,/} \mid \mathcal{X} \mid \mathcal{C}$

- a subset of Boolean programs (without divisions, square roots or tests) that does not contain any square root or division:

  - $\mathsf{B}_{let} := \mathit{Buop}\ \mathsf{B}_{let} \mid \mathsf{B}_{let}\ \mathit{Bbop}\ \mathsf{B}_{let} \mid \mathsf{N}\ \mathit{Cbop}\ \mathsf{N} \mid (\mathsf{B}_{let}, \mathsf{B}_{let}) \mid \mathsf{fst}\ \mathsf{B}_{let} \mid \mathsf{snd}\ \mathsf{B}_{let} \mid \mathsf{let}\ \mathcal{V} = \mathsf{B}_{let}\ \mathsf{in}\ \mathsf{B}_{let} \mid \mathcal{X} \mid \mathcal{C}$

    in this Boolean subtype, we allow variable definition of square root and division free Boolean expressions in order to reduce the size of the output program, as we will see in Section 3.6.

- the different sets of expressions where Boolean expressions do not contain any square root or division but can contain local variable definitions and where numerical ones:

  - can not contain square roots or divisions: $\mathsf{E}_{\mathsf{N}} := \mathsf{N} \mid \mathsf{B}_{let} \mid (\mathsf{E}_{\mathsf{N}}, \mathsf{E}_{\mathsf{N}})$
  - can contain square roots or divisions: $\mathsf{E}_{\mathsf{N}_{\surd,/}} := \mathsf{N}_{\surd,/} \mid \mathsf{B}_{let} \mid (\mathsf{E}_{\mathsf{N}_{\surd,/}}, \mathsf{E}_{\mathsf{N}_{\surd,/}})$

- The programs that do not contain any square root or division
  $\mathsf{P}_{\mathsf{N}} := \mathsf{let}\ \mathcal{V} = \mathsf{P}_{\mathsf{N}}\ \mathsf{in}\ \mathsf{P}_{\mathsf{N}} \mid \mathsf{if}\ \mathsf{P}_{\mathsf{N}}\ \mathsf{then}\ \mathsf{P}_{\mathsf{N}}\ \mathsf{else}\ \mathsf{P}_{\mathsf{N}} \mid \mathsf{E}_{\mathsf{N}}$

Given these definitions, the subtype 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) corresponds to the following definition:

DEFINITION 3.8 (Targeted language). The language the transformation targets is:

$$\mathsf{P}_{\mathsf{N}_{\surd,/}} := \mathsf{let}\ \mathcal{V} = \mathsf{P}_{\mathsf{N}}\ \mathsf{in}\ \mathsf{P}_{\mathsf{N}_{\surd,/}} \mid \mathsf{if}\ \mathsf{P}_{\mathsf{N}}\ \mathsf{then}\ \mathsf{P}_{\mathsf{N}_{\surd,/}}\ \mathsf{else}\ \mathsf{P}_{\mathsf{N}_{\surd,/}} \mid \mathsf{E}_{\mathsf{N}_{\surd,/}}$$

For example $(\sqrt{x}, a > b)$ is in $\mathsf{E}_{\mathsf{N}_{\surd,/}}$ but not in E_{N} and $\sqrt{a} > b$ belongs to none of these sets. Notice that if a program returns a Boolean value and is in $\mathsf{P}_{\mathsf{N}_{\surd,/}}$ then it does not contain any division or square root:

LEMMA 3.3 (Boolean $\mathsf{P}_{\mathsf{N}_{\surd,/}}$).

$$\forall\, \mathsf{p} \in \mathsf{P}_{\mathsf{N}_{\surd,/}},\ (\exists\, \Gamma,\ \mathit{Ty}_{\Gamma}(\mathsf{p}) = \mathbb{B}^{n}) \Longrightarrow \mathsf{p} \in \mathsf{P}_{\mathsf{N}}$$

Proof. By induction on p using the definition of $\mathsf{P}_{\mathsf{N}_{\surd,/}}$. ◀

These definitions allow us to characterize the set of programs transformed by each step of our transformation and what kind of programs it produces, $\mathsf{P}_{\mathsf{N}_{\surd,/}}$ being the output language of our transformation. The language and the target language being defined, we now define the specification of our transformation.

3.3 Specification of the Transformation

The transformation we want to define targets critical embedded systems. Programs used in these systems are most of the time proved to be type safe and one can also prove that failure due to divisions by zero or square roots of a negative number do not occur. Therefore we assume that the programs we want to transform are well typed (see Definition 3.3) and do not fail in the environment where they are evaluated. Hence we do not have to enforce the failure cases that disappear when removing divisions and square roots, *e.g.*, we can transform $1/x > 0$ into $x > 0$ instead of $\mathsf{if}\ x = 0\ \mathsf{then\ Fail\ else}\ x > 0$. Thus we only ensure the preservation of the type and the semantics in every environment where the initial program is well typed and does not fail. This corresponds to the following predicate:

Definition 3.9 (Program equivalence). We say that a program p_2 is equivalent to a program p_1 modulo failure, denoted $sem_ty_eq(\mathsf{p}_1, \mathsf{p}_2)$ when the following predicate holds:

$$\forall \Gamma,\ Ty_\Gamma(\mathsf{p}_1) \neq \mathbb{U} \implies Ty_\Gamma(\mathsf{p}_2) = Ty_\Gamma(\mathsf{p}_1) \wedge \\ \forall Env, [\![\ \mathsf{p}_1\ ]\!]_{Env} \neq Fail \implies [\![\ \mathsf{p}_2\ ]\!]_{Env} = [\![\ \mathsf{p}_1\ ]\!]_{Env}$$

As already mentioned in Section 3.2.2, our transformation aims at transforming any program in Prog into a program in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$, therefore given the no failure hypothesis, we can now formally define our main goal as the defintion of a function called *Elim* that has the following specification:

Definition 3.10 (Transformation specification).
We aim at defining a function transforming program, called *Elim*, such that:

$$\forall p \in \mathsf{Prog},\ Elim(\mathsf{p}) \in \mathsf{P}_{\mathsf{N}_{\sqrt{,/}}} \wedge sem_ty_eq(\mathsf{p}, Elim(\mathsf{p}))$$

The transformation being specified, the following section presents why we need such a transformation and what kind of guaranties having a program in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ instead of Prog can provide.

3.4 Reliable Computations over Real Numbers

As said in the introduction, the main purpose of removing square roots and divisions from these programs is to produce an equivalent program only using operations that we can use safely because they perform exact computation. However, performing exact computation even with addition or multiplication is troublesome, in this section we discuss some way to ensure the exactness of such computations and some properties verified by the transformed program.

3.4.1 Exact Computation

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. Let us now introduce the subset $\mathbb{D}$ of $\mathbb{R}$. $\mathbb{D}$ is the set of dyadic rational numbers, the rational numbers whose 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 (*e.g.*, $1/5$ has no finite binary representation) and will force us to use round offs as in the floating point number representation [IEE85].

Computing in $\mathbb{D}$ with addition, multiplication and subtraction can be done exactly by using a dynamic representation of real numbers which allows us to use all the necessary bits to avoid losing accuracy during computation (*e.g.*, the product of two numbers of size n can be stored in a number of size $2n$). Certainly, these 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 provide the number of bits required by every computation depending on the number of bits of the inputs. The following algorithm computes a large over-approximation of the size required by a program to compute exactly with addition multiplication and subtraction. There may be more efficient way but the purpose here is only to show that the memory use can be predicted.

ALGORITHM 3.11 (Static analysis of required memory). Given m_{FV}, the number of bits that are used to represent free variables (*e.g.*, inputs), the following rules compute the pair of the memory required for the exact computation of the program and the memory required to store the result. We use an environment $\mathcal{M}$ to record the memory used by each variable. The memory required by the constant c is the number of bits used for its exact representation, denoted by mem_c:

$$\frac{}{\mathcal{M},(\mathsf{x},mx) \vdash_m \mathsf{x} : (0,mx)} \qquad \frac{}{\mathcal{M} \vdash_m \mathsf{x} : (m_{FV},m_{FV})} \quad \text{(if x is a free variable)}$$

$$\frac{}{\mathcal{M} \vdash_m c : (mem_c,mem_c)} \qquad \frac{\mathcal{M} \vdash_m \mathsf{e} : (m,mr)}{\mathcal{M} \vdash_m - \mathsf{e} : (m,mr)}$$

$$\frac{\mathcal{M} \vdash_m \mathsf{e}_1 : (m_1,mr_1) \quad \mathcal{M} \vdash_m \mathsf{e}_2 : (m_2,mr_2)}{\mathcal{M} \vdash_m (\mathsf{e}_1 + \mathsf{e}_2) : (max(m_1,mr_1+m_2)+max(mr_1,mr_2)+1,max(mr_1,mr_2)+1)}$$

$$\frac{\mathcal{M} \vdash_m \mathsf{e}_1 : (m_1,mr_1) \quad \mathcal{M} \vdash_m \mathsf{e}_2 : (m_2,mr_2)}{\mathcal{M} \vdash_m (\mathsf{e}_1 \times \mathsf{e}_2) : (max(m_1,mr_1+m_2)+mr_1+mr_2,mr_1+mr_2)}$$

$$\frac{\mathcal{M} \vdash_m \mathsf{e}_1 : (m_1,mr_1) \quad \mathcal{M},(\mathsf{x},mr_1) \vdash_m \mathsf{e}_2 : (m_2,mr_2)}{\mathcal{M} \vdash_m (\mathsf{let}\ \mathsf{x} : \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2) : (max(m_1,mr_1+m_2),mr_2)}$$

$$\frac{\mathcal{M} \vdash_m \mathsf{e}_1 : (m_1,mr_1) \quad \mathcal{M} \vdash_m \mathsf{e}_2 : (m_2,mr_2) \quad \mathcal{M} \vdash_m \mathsf{f} : (m_f,mr_f)}{\mathcal{M} \vdash_m (\mathsf{if}\ \mathsf{f}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2) : (max(m_f,m_1,m_2),max(mr_1,mr_2))}$$

We define similar rules for the Booleans constructors, they are omitted here.

Being able to compute exactly with addition, multiplication and subtraction enables a protection of the control flow of the program from rounding and therefore to have program that are continuous regarding the precision of the implementation of the division and square root operations.

3.4.2 Program $\sqrt{}$ and / Continuity

Given a program in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$, neither the variables defined in this program nor the Boolean values of the tests can contain divisions or square roots. We assume we have a concrete

semantics $[\![p]\!]^{\mathbb{K}}$ that represents the effective behavior of a program on a real machine. We have seen in Section 3.4.1 that we can compute exactly with $+$, $-$ and $\times$. We assume that this semantics implements this exact computation, therefore for any program p in $\mathsf{P_N}$ and for any environment Env, the following equality holds: $[\![p]\!]^{\mathbb{K}}_{Env} = [\![p]\!]_{Env}$. This concrete semantics has the same derivation rules as the semantics on real numbers, except for divisions and square roots that depends on the chosen implementations:

DEFINITION 3.12 (Concrete Semantics). $/^{\mathbb{K}}$ and $\sqrt{}^{\mathbb{K}}$ are the functions corresponding to the chosen implementation of $/$ and $\sqrt{}$

$$\frac{Env \vdash [\![\mathsf{E}]\!]^{\mathbb{K}}{=}e}{Env \vdash [\![\sqrt{\mathsf{E}}]\!]^{\mathbb{K}}{=}\sqrt{e}^{\mathbb{K}}} \qquad \frac{Env \vdash [\![\mathsf{E}_1]\!]^{\mathbb{K}}{=}e_1 \quad Env \vdash [\![\mathsf{E}_2]\!]^{\mathbb{K}}{=}e_2}{Env \vdash [\![\mathsf{E}_1/\mathsf{E}_2]\!]^{\mathbb{K}}{=}e_1 \ /^{\mathbb{K}}\ e_2}$$

Thus having a program in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ allows us to protect its control flow from $\sqrt{}$ and $/$, this means that even if the $/^{\mathbb{K}}$ and $\sqrt{}^{\mathbb{K}}$ operations are not exact, the result is close to the real number semantics. This property is related to the notion of continuity and robustness of programs discussed in [CGLN11, CGL12]. To define our specific program continuity we first define the notion of corresponding environment for a sub term of a program.

DEFINITION 3.13 (Sub Term Environment). For any program p and any environment Env, the Sub-Term Environment $STE(\mathsf{p},Env)$ is recursively defined by:

$$STE(\mathsf{let\ x = b\ in\ SC}, Env) = \{(\mathsf{let\ x = b\ in\ SC}, Env)\} \cup STE(\mathsf{b}, Env) \cup STE(\mathsf{SC}, Env \oplus (\mathsf{x}, [\![\mathsf{b}]\!]_{Env})$$

And for any other constructor C:

$$STE(\mathsf{C}(\mathsf{p}_1,\ldots,\mathsf{p}_n), Env) = \{(\mathsf{C}(\mathsf{p}_1,\ldots,\mathsf{p}_n), Env)\} \cup \bigcup_1^n STE(\mathsf{p}_i, Env)$$

PROPOSITION 3.4 (Program Continuity). *For $x, \epsilon \in \mathbb{R} \times \mathbb{R}^+$, we denote by $V(x,\epsilon)$ the neighborhood of x, i.e., $\{y \in \mathbb{R} \mid |x-y| < \epsilon\}$. We define the neighborhood of the semantics of the sub-terms: $VSEM(\mathsf{p}, Env, \epsilon) = \cup_{\{y \mid \exists (e,E) \in STE(\mathsf{p},Env),\ [\![e]\!]_E = y\}} V(y,\epsilon)$. Then we can state that the program in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ that do not fail are continuous:*

$$\forall \mathsf{p} \in \mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}, \forall Env, \forall \epsilon \in \mathbb{R}^+, \exists \eta \in \mathbb{R}^+,\ [\![\mathsf{p}]\!]_{Env} \neq Fail \Longrightarrow$$
$$\forall \mathsf{e}_1,\ \mathsf{e}_2 \in VSEM(\mathsf{p}, Env, \epsilon)^2,$$
$$\mathsf{e}_2 \neq 0 \Rightarrow |\mathsf{e}_1 \ /^{\mathbb{K}}\ \mathsf{e}_2 - \mathsf{e}_1 \ /\ \mathsf{e}_2| < \eta \ \wedge\ \mathsf{e}_1 \geq 0 \Rightarrow |\sqrt{\mathsf{e}_1}^{\mathbb{K}} - \sqrt{\mathsf{e}_1}| < \eta \Longrightarrow$$
$$|[\![\mathsf{p}]\!]^{\mathbb{K}}_{Env} - [\![\mathsf{p}]\!]_{Env}| < \epsilon$$

Where distance on semantics is defined by

$|True - False| = 1$

$|(v1, v2) - (w1, w2)| = max(|v1 - w1|, |v2 - w2|)$

and the usual distance on $\mathbb{R}$.

Proof. By induction on p:

- p = if F then $\mathsf{p1}$ else $\mathsf{p2}$:
  F is in $\mathsf{P_N}$ thus $[\![\mathsf{F}]\!]^{\mathbb{K}}_{Env} = [\![\mathsf{F}]\!]_{Env}$ then the induction hypothesis on $\mathsf{p1}$ and $\mathsf{p2}$ with the same ϵ provides η_1 and η_2. Therefore $\eta = min(\eta_1, \eta_2)$ verifies the property.

- $\mathsf{p} = \mathsf{let\ x = b\ in\ sc}$:
  b is in $\mathsf{P_N}$ thus $[\![\mathsf{b}]\!]^{\mathbb{K}}_{Env} = [\![\mathsf{b}]\!]_{Env}$ and the induction hypothesis on sc applies with the new environment $Env \oplus (\mathsf{x}, [\![\mathsf{p1}]\!]_{Env})$.

- $\mathsf{p} = (\mathsf{p1},\mathsf{p2})$ can be proved with the induction hypothesis as for the $\mathsf{if\ then\ else}$ case.

- p in B_{let} then p is in $\mathsf{P_N}$ thus $[\![\mathsf{p}]\!]^{\mathbb{K}}_{Env} = [\![\mathsf{p}]\!]_{Env}$

- p in $\mathsf{N}_{\surd,/}$, we only give the case $\mathsf{p} = \mathsf{p1}\ /\ \mathsf{p2}$, other cases are similar:
  We denote $s_1 = [\![\mathsf{p_1}]\!]_{Env}$, $s_2 = [\![\mathsf{p_2}]\!]_{Env}$, $s_1^{\mathbb{K}} = [\![\mathsf{p_1}]\!]^{\mathbb{K}}_{Env}$ and $s_2^{\mathbb{K}} = [\![\mathsf{p_2}]\!]^{\mathbb{K}}_{Env}$
  We define $f(x,y) = |\frac{(s_1+x)}{(s_2+y)} - \frac{s_1}{s_2}|$
  Since $s_2 \neq 0$ (if not $[\![\mathsf{p}]\!]_{Env} = Fail$) and f is continuous around $(0,0)$, then
  $$\exists \alpha,\ max(|x|,|y|) < \alpha \Rightarrow |f(x,y)| < \epsilon/2, \text{ we denote } \alpha' = min(\alpha,\epsilon)$$
  By applying induction hypothesis on $\mathsf{p_1}$, Env and α' we get η_1 such that:
  $$(HI1) \quad \forall \mathsf{e_1},\ \mathsf{e_2}, \in VSEM(\mathsf{p_1}, Env, \alpha')^2, \\ |\mathsf{e_1}\ /^{\mathbb{K}}\ \mathsf{e_2} - \mathsf{e_1}\ /\ \mathsf{e_2}| < \eta_1 \wedge |\sqrt{\mathsf{e_1}}^{\mathbb{K}} - \sqrt{\mathsf{e_1}}| < \eta_1 \Longrightarrow |s_1^{\mathbb{K}} - s_1| < \alpha'$$
  and a similar η_2 and hypothesis $(HI2)$ by applying induction hypothesis on $\mathsf{p_2}$.
  Lets take $\eta = min(\eta_1, \eta_2, \epsilon/2)$
  Assuming (H):
  $$\forall\ \mathsf{e_1},\ \mathsf{e_2} \in VSEM(\mathsf{p}, Env, \epsilon)^2,\ |\mathsf{e_1}\ /^{\mathbb{K}}\ \mathsf{e_2} - \mathsf{e_1}\ /\ \mathsf{e_2}| < \eta \wedge |\sqrt{\mathsf{e_1}}^{\mathbb{K}} - \sqrt{\mathsf{e_1}}| < \eta$$
  We want to prove that $|s_1^{\mathbb{K}} /^{\mathbb{K}} s_2^{\mathbb{K}} - s_1/s_2| < \epsilon$
  Let us first prove the premise of $(HI1)$:
  Since $\mathsf{p_1} \ll \mathsf{p}$ and $\alpha' \leq \epsilon$ we have $VSEM(\mathsf{p_1}, Env, \alpha') \subseteq VSEM(\mathsf{p}, Env, \epsilon)$
  Since $\eta_1 \geq \eta$, using (H) we can prove this premise
  Thus we have $|s_1^{\mathbb{K}} - s_1| < \alpha'$, with the same reasoning we have $|s_2^{\mathbb{K}} - s_2| < \alpha'$
  Thus $\exists x_1, |x_1| < \alpha' \wedge s_1^{\mathbb{K}} = s_1 + x_1$ and $\exists x_2, |x_2| < \alpha' \wedge s_2^{\mathbb{K}} = s_2 + x_2$
  Therefore:
  $$\begin{aligned} |s_1^{\mathbb{K}} /^{\mathbb{K}} s_2^{\mathbb{K}} - \frac{s_1}{s_2}| &= |(s_1+x_1)/^{\mathbb{K}}(s_2+x_2) - \frac{s_1+x_1}{s_2+x_2} + \frac{s_1+x_1}{s_2+x_2} - \frac{s_1}{s_2}| \\ &\leq |(s_1+x_1)/^{\mathbb{K}}(s_2+x_2) - \frac{s_1+x_1}{s_2+x_2}| + |\frac{s_1+x_1}{s_2+x_2} - \frac{s_1}{s_2}| \end{aligned}$$
  Since $|x_1| < \alpha' \leq \epsilon$ and $|x_1| < \alpha' \leq \epsilon$ then, using (H), we have:
  $$|(s_1+x_1)/^{\mathbb{K}}(s_2+x_2) - \tfrac{s_1+x_1}{s_2+x_2}| < \eta \leq \epsilon/2$$
  Since $|x_1| < \alpha' \leq \alpha$ and $|x_1| < \alpha' \leq \alpha$ then, given the definition of α:
  $$|\tfrac{s_1+x_1}{s_2+x_2} - \tfrac{s_1}{s_2}| < \epsilon/2$$
  Then the sum is smaller than ϵ and we have the theorem we want.

The other arithmetic operator can be handled in the same way, therefore, according to this definition the program we build are continuous. ◀

This notion of continuity can be stated in natural language as:

If the concrete divisions and square roots are precise enough around the real semantics of the sub-term then the concrete result of the program is close to its real semantics.

This avoids huge mistakes due to the discontinuity of the test operator:

EXAMPLE 3.3. On floating point numbers, the concrete semantics of this program is 1000 whereas its semantics on real numbers is 0:

$\mathsf{if}\ \sqrt{2} \times \sqrt{2} \neq 2\ \mathsf{then}\ 1000\ \mathsf{else}\ 0$ Any rounding error in a test can introduce large differences between the abstract and the concrete semantics.

The control flow of the program is protected from the rounding introduced by these operations and therefore the result is, even in the worst case, close to the result we would expect with real number arithmetic.

3.5 TRANSFORMING PROGRAMS

In this section we first give some principles about the termination of some program transformations then we present the first step of our transformation.

3.5.1 Orders on Programs

In order to prove the termination of many of our transformations, we need to introduce different well founded orders on programs.

DEFINITION 3.14 (Well founded order). An order $\prec$ on a set $\mathcal{S}$ is well founded, denoted $wf(\prec)$, if every non empty set has a minimal element:

$$\forall\, S,\ S \neq \emptyset \Longrightarrow \exists\, x \in S,\ \forall\, z \in S,\ \neg\, z \prec x$$

The main order that is used is the usual sub-term order, denoted $\ll$. This order states that $\mathsf{p}_1 \ll \mathsf{p}_2$ when p_1 is a sub-term of p_2. However this is not always sufficient so we sometimes need to extend this order by composing it with another order. We compose two orders as the corresponding lexicographic order:

DEFINITION 3.15 (Order composition). Given two orders, $\prec_1$ on a set $\mathcal{S}_1$ and $\prec_2$ on $\mathcal{S}_2$, we define the composition order on $\mathcal{S}_1 \times \mathcal{S}_2$ as:

$$lex(\prec_1, \prec_2)((x_1, x_2), (y_1, y_2)) = x_1 \prec_1 y_1 \vee x_1 = y_1 \wedge x_2 \prec_2 y_2$$

As proved in [BN98] (*Theorem 2.4.2*) the composition of two well-founded orders is well founded. Using this composition of orders, we can compose an order with a measure function.

DEFINITION 3.16 (Order with measure). Given an order $\prec$ on $\mathcal{S}$ and a measure function m from $\mathcal{S}$ to $\mathbb{N}$, we denote $\prec_m$ the following order:

$$x \prec_m y = lex(<_{\mathbb{N}}, \prec)((m(x), x), (m(y), y))$$

Where $<_{\mathbb{N}}$ is the usual strict order on natural numbers.

Below we illustrate this notation with an order on programs based on the number of variable definition it contains and the sub-term order:

EXAMPLE 3.4 (Letin based order). Given p a program, if we denote $letins(\mathsf{p})$ the number of variable definitions it contains, then $\ll_{letins}$ is a well founded order.

In the next section we detail the first step of the transformation, that is reducing any program into its normal form.

3.5.2 Program Normal Form P

In Section 3.2.2, we introduced the P subtype as the source language of the core of our transformation. Since the global transformation aims at transforming every program in Prog, the first step of the transformation is to reduce any program in Prog into an equivalent one in P. The normalization can be done using the following set of rules:

DEFINITION 3.17 (Prog Normalization). Normalization is done by inverting tests and variable definitions on one side and the binary and unary operators on the other side

- 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$\mapsto$x'] op e3) $\quad$ x' $\notin FV$((e2,e3))
  - e1 op (let x = e2 in e3) $\longrightarrow$
    let x' = e2 in (e1 op e3[x$\mapsto$x']) $\quad$ x' $\notin FV$((e1,e3))
  - ((let x = e1 in e2),e3) $\longrightarrow$
    let x' = e1 in (e2[x$\mapsto$x'],e3) $\quad$ x' $\notin FV$((e2,e3))
  - (e1,let x = e2 in e3) $\longrightarrow$
    let x' = e2 in (e1,e3[x$\mapsto$x']) $\quad$ 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:
  in the binary operator case we define a variable corresponding to the test instead of duplicating the other argument to avoid an explosion of the size of the code:
  - uop (if f then e1 else e2) $\longrightarrow$
    if f then (uop e1) else (uop e2)
  - (if f then e1 else e2) op e3 $\longrightarrow$
    let xi = if f then e1 else e2 in (xi op e3) $\quad$ xi $\notin FV$(e3)
  - e1 op (if f then e2 else e3) $\longrightarrow$
    let xi = if f then e2 else e3 in (e1 op xi) $\quad$ xi $\notin FV$(e1)
  - ((if f then e1 else e2),e3) $\longrightarrow$
    let xi = if f then e1 else e2 in (xi,e3) $\quad$ xi $\notin FV$(e3)

- (e1,if f then e2 else e3)) $\longrightarrow$
  let xi = if f then e2 else e3 in (e1,xi) $\quad$ xi $\notin FV$(e1)
- fst (if f then e1 else e2) $\longrightarrow$ if f then fst (e1) else fst (e2)
- snd (if f then e1 else e2) $\longrightarrow$ if f then snd (e1) else snd (e2)

- projections reductions:
  - fst (e1,e2) $\longrightarrow$ e1
  - snd (e1,e2) $\longrightarrow$ e2

Using this set of rules we can transform any program in Prog into a program in P that has the same semantics:

PROPOSITION 3.5 (Prog and P equivalence).

$$\forall\, p \in \mathsf{Prog},\ \exists\, p^{\top} \in \mathsf{P},\ \forall Env,\ [\![p]\!]_{Env} = [\![p^{\top}]\!]_{Env}$$

Proof. By using the transformation rules defined in Definition 3.17. ◀

We will see in Chapter 5 how we can implement a strategy for this set of reduction rules and prove the termination. Now that we have a program in P, we present how we eliminate square roots and divisions from it. The first step of this transformation is to define the elimination of square roots and divisions in Boolean expressions. The algorithm we use is a variant of the algorithm introduced in Chapter 1.

3.6 BOOLEAN EXPRESSION TRANSFORMATION

In this section we present how to eliminate square roots and divisions in *Boolean expressions*. A Boolean expression is a term in E_u of type $\mathbb{B}$ in a valid environment, that is:

DEFINITION 3.18 (Boolean Expression). A term e in Prog is said to be a Boolean expression when:

$$\mathsf{e} \in \mathsf{E}_u \wedge \exists\, \Gamma,\ Ty_{\Gamma}(e) = \mathbb{B}$$

There is a trivial mapping between the *Boolean expressions* and the $\mathcal{B}$ set introduced in Chapter 1 extended with Boolean variable and projections. However, in E_u, pairs are not allowed, therefore projections can only be applied to variables *e.g.*,fst(snd(x)). Therefore the projections can not embed any square root or division and they can be treated the same way as constants or variables in the normalization and elimination rules.

The language the global transformation targets is still $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ and its subtype corresponding to Boolean expressions is B_{let}. Therefore the goal of the elimination of square roots and divisions in Boolean expression is to define a function, $elim_{\mathbb{B}}$ that has the following specification:

DEFINITION 3.19 ($elim_{\mathbb{B}}$ specification). We want to define a function $elim_{\mathbb{B}}$ such that for every Boolean expression e:

$$elim_{\mathbb{B}}(\mathsf{e}) \in \mathsf{B}_{let} \wedge sem_ty_eq(\mathsf{e}, elim_{\mathbb{B}}(\mathsf{e}))$$

The Algorithm 1.13 presented in the previous chapter verifies this specification, indeed it transforms any Boolean expression into an equivalent square root and division free expression. However we allowed the Boolean variable definitions in B_{let} for the following reason. Each application of the square root elimination rules defined in Definition 1.12 can produce up to 6 atoms, but some of these atoms are equals or opposites. Therefore we replace that square root elimination rule with a new one that name the atoms in order to avoid an explosion of the size of the formula:

DEFINITION 3.20 (Square root elimination). We define new rules for the operators $>$, $\geq$, $<$ and $\leq$. We only give the rule for $>$ the other ones being almost identical:

$$(\mathsf{P}.\sqrt{\mathsf{Q}}+\mathsf{R}) > 0 \longrightarrow$$
$$\mathsf{let}\ ((\mathsf{c}_1,\mathsf{c}_2),(\mathsf{c}_3,\mathsf{c}_4)) = ((\mathsf{P} > 0,\mathsf{R} > 0),(\mathsf{P}^2.\mathsf{Q}-\mathsf{R}^2 > 0,\mathsf{P}^2.\mathsf{Q}-\mathsf{R}^2 \neq 0))\ \mathsf{in}$$
$$(\mathsf{c}_1 \wedge \mathsf{c}_2) \vee (\ \mathsf{c}_1 \wedge \mathsf{c}_3\ ) \vee (\ \mathsf{c}_2 \wedge \neg\ \mathsf{c}_3 \wedge \mathsf{c}_4\ )$$

For the $=$ and $\neq$ cases, we still use the elimination rules introduced in Definition 1.12 that do not use variables.

***Remark* 3.1.** We use $\neg\ \mathsf{c}_3 \wedge \mathsf{c}_4$ that corresponds to $\neg\ \mathsf{P}^2.\mathsf{Q}-\mathsf{R}^2 > 0 \wedge \mathsf{P}^2.\mathsf{Q}-\mathsf{R}^2 \neq 0$ instead of defining c_4 directly as $\mathsf{P}^2.\mathsf{Q}-\mathsf{R}^2 < 0$. This allows us to define c_4 as an equality where the elimination of square roots only produces two atoms instead of four. This way, we minimize the number of atoms that will be produced in the following steps.

By replacing the rules of Definition 1.12 by the ones from Definition 3.20 in Algorithm 1.13, we get the $elim_{\mathbb{B}}$ function we wanted, that respects the specification from Definition 3.19. This function transforms every formula into an equivalent one in B_{let}:

THEOREM 3.6 (Square root and division elimination in Boolean expression). *Every Boolean expression has a square root and division free equivalent* B_{let} *program:*

$$\forall\ \mathsf{e}\ \in \mathsf{E}_u,\ (\exists\ \Gamma,\ Ty_\Gamma(e) = \mathbb{B}) \Longrightarrow$$
$$Elim_{\mathbb{B}}(\mathsf{e})\ \in \mathsf{B}_{let}\ \wedge\ \forall\ Env,\ [\![e]\!]_{Env} \neq Fail \Rightarrow [\![\mathsf{e}]\!]_{Env} = [\![Elim_{\mathbb{B}}(\mathsf{e})]\!]_{Env}$$

We are now able to eliminate the square roots and divisions from any Boolean expression, however we still can not ensure that their computation is independent from the rounding that these operations might introduce. Indeed the free variables that appear in these expressions may have been computed using square roots or divisions *e.g.*,

$$\mathsf{let}\ \mathsf{x} = \sqrt{a}\ \mathsf{in}\ \mathsf{x} > e \times r$$

In the next section, we introduce how we can handle such variable definitions in order to avoid having divisions or square roots in their definitions.

3.7 VARIABLE DEFINITION TRANSFORMATION

Rounding due to square roots or divisions might not be explicit in all the Boolean expressions. This means that, even if a Boolean expression does not contain any square roots or divisions, the variables that are used in this expression may have been defined using these operations. Therefore the value of this variable is not exact and we can not guaranty anymore the result of the Boolean computation as in the following example:

EXAMPLE 3.5 (Variable definitions with square roots).

$$\mathsf{let\ x} = (3 \times a)/(b+5)\ \mathsf{in\ let\ y} = \mathsf{x}{+}\sqrt{c+d}\ \mathsf{in\ x + y > e}$$

Therefore we need to transform these variable definitions such that the variables used in Boolean expressions do not embed any square root or division computations but we also have to take care of the size of the output code. Indeed, a simple solution would be to inline all the variable definitions as in the following example:

EXAMPLE 3.6 (Variable definitions with square roots).

$$\begin{array}{ll} \mathsf{let\ x} = (3 \times a)/(b+5)\ \mathsf{in\ let\ y} = \mathsf{x}{+}\sqrt{c+d}\ \mathsf{in\ x + y > e} & \\ \longrightarrow & (3 \times a)/(b+5) + (3 \times a)/(b+5) + \sqrt{c+d} > \mathsf{e} \end{array}$$

Given a variable definition, let x = b in sc, we call b the *body* and sc the *scope* of the definition, the inlining consists in replacing every occurrence of the variable in the scope by the body of the definition. However using such transformations, especially when the variables are defined using conditional expressions, makes the size of the output code explode:

EXAMPLE 3.7 (Variable definitions with tests).

$$\mathsf{let\ x} = \mathsf{if}\ F\ \mathsf{then}\ a/b\ \mathsf{else}\ \sqrt{c}\ \mathsf{in\ x > d} \quad \longrightarrow \quad (F\ \wedge\ a/b > d)\ \vee\ (\neg F\ \wedge\ \sqrt{c} > d)$$

Therefore we had to find another way to transform these variable definitions. The transformed program still uses variable definitions but the expressions contained in the bodies of these definitions are square root and division free.

3.7.1 Specification of Variable Definition Transformation

As mentioned previously we want to eliminate square roots and divisions from the variable definition bodies, without significantly increasing the size of the output code. This means that we can not inline the definitions but we have to replace any definition by a new definition that is square root and division free. The global transformation being a recursive algorithm, we can assume that the body of the definition has already been transformed and is therefore in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$. Therefore we aim at defining a function that have the following specification:

DEFINITION 3.21 ($Elim_{let}$ specification). Given $\mathsf{x} \in \mathcal{X}$, $\mathsf{b} \in \mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ and $\mathsf{sc} \in \mathsf{P}$, we define the $Elim_{let}$ function that computes x', b' and sc' such that:

$$\forall\ Env,\ [\![\mathsf{let\ x = b\ in\ sc}]\!]_{Env} \neq Fail \Longrightarrow$$
$$[\![\mathsf{let\ x = b\ in\ sc}]\!]_{Env} = [\![\mathsf{let\ x' = b'\ in\ sc'}]\!]_{Env}\ \wedge \mathsf{b'} \in \mathsf{P_N} \wedge \mathsf{sc'} \in \mathsf{P}$$

The main idea consists in replacing a variable definition by the definition of multiple variables corresponding to the square roots and division free sub-expressions of the input body. The transformation of Example 3.5 is:

EXAMPLE 3.8 (Naming of square root and division free sub expressions).

$$\begin{array}{l} \mathsf{let\ x} = (3 \times a)/(b+5)\ \mathsf{in\ let\ y} = \mathsf{x}{+}\sqrt{c+d}\ \mathsf{in\ x + y > e} \quad \longrightarrow \\ \quad \mathsf{let}\ (\mathsf{x}_n,\mathsf{x}_d) = ((3 \times a),(b+5))\ \mathsf{in\ let\ y} = \mathsf{x}_n/\mathsf{x}_d + \sqrt{c+d}\ \mathsf{in}\ \mathsf{x}_n/\mathsf{x}_d + \mathsf{y > e} \quad \longrightarrow \\ \qquad \mathsf{let}\ (\mathsf{x}_n,\mathsf{x}_d) = ((3 \times a),(b+5))\ \mathsf{in} \\ \qquad \mathsf{let}\ (\mathsf{y}_{n1},\mathsf{y}_{d1},\mathsf{y}_{sq}) = (\mathsf{x}_n,\mathsf{x}_d,c+d)\ \mathsf{in}\ \mathsf{x}_n/\mathsf{x}_d + \mathsf{y}_{n1}/\mathsf{y}_{d1} + \sqrt{\mathsf{y}_{sq}} > \mathsf{e} \end{array}$$

It is easy to define such an algorithm for a variable definition whose body is an expression in E, the only thing to do is to define a tuple corresponding to the different sub expressions that are square root and division free and to inline the former definition using these new defined variables (*e.g.*, $\mathsf{x}_n/\mathsf{x}_d$ and $\mathsf{y}_{n1}/\mathsf{y}_{d1} + \sqrt{\mathsf{y}_{sq}}$ in the previous example). The principle of this transformation can be extended to variable definitions that contain test:

EXAMPLE 3.9 (Variable definition with test).

$$\begin{array}{l} \mathsf{let\ x} = \\ \quad \mathsf{if\ F\ then}\ a_1 + \sqrt{a_2} \\ \qquad \mathsf{else}\ \frac{b_1}{b_2} \\ \mathsf{in\ P} \end{array} \quad \longrightarrow \quad \begin{array}{l} \mathsf{let}\ (\mathsf{x}_1,\mathsf{x}_2,\mathsf{x}_3) = \\ \quad \mathsf{if\ F\ then}\ (a_1, a_2, 1) \\ \qquad \mathsf{else}\ (b_1, 0, b_2) \\ \mathsf{in\ P}[\mathsf{x} \mapsto \frac{\mathsf{x}_1+\sqrt{\mathsf{x}_2}}{\mathsf{x}_3}] \end{array}$$

The expression $\frac{\mathsf{x}_1+\sqrt{\mathsf{x}_2}}{\mathsf{x}_3}$ comes from the constrained anti-unification of $a_1 + \sqrt{a_2}$ and $\frac{b_1}{b_2}$. It allows us to only define sub-expressions that are square root and division free and to only inline a small expression. We have defined this constrained anti-unification problem in a very generic way and presented an algorithm that computes a constrained template of arithmetic expression in Chapter 2. We now use this anti-unification algorithm on arithmetic expressions to transform the variable definitions.

3.7.2 Variable Definition Transformation

The goal in this section is to define a function $Elim_{let}$ whose specification has been defined in Definition 3.21. The elimination of square roots and divisions from a variable definition (let x = p1 in p2 where $\mathsf{x} \in \mathcal{X}$, $\mathsf{p1} \in \mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ and $\mathsf{p2} \in \mathsf{P}$) relies on a decomposition of the body p1 into three distinct elements:

DEFINITION 3.22. A program in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ can be decomposed into

- a *program part* Pp of type $\mathsf{E}^n_{\mathsf{N}_{\sqrt{,/}}} \longrightarrow \mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ and $\mathsf{E}^n_{\mathsf{N}} \longrightarrow \mathsf{P}_{\mathsf{N}}$ that represents the test and local variable definition structure of the body
- a list of expressions $(\mathsf{e}_1, ..., \mathsf{e}_n)$ of $\mathsf{E}^n_{\mathsf{N}_{\sqrt{,/}}}$ such that $Pp(\mathsf{e}_1, ..., \mathsf{e}_n) = \mathsf{p1}$. This list of expression is then decomposed again by anti-unification with $\sqrt{}$ and / as forbidden symbols. This produces:
  - the *template* T in $\mathsf{E}_{\mathsf{N}_{\sqrt{,/}}}$ that contains the variables $\mathsf{x}_1, ..., \mathsf{x}_k$
  - n substitutions, $\sigma_1, ..., \sigma_n$ on the same domain $\mathsf{x}_1, ..., \mathsf{x}_k$ such that:
    $$\forall i \in [1...n],\ T\sigma_i = \mathsf{e}_i$$
    according to the usual rules of arithmetic and
    $$\mathcal{I}(\sigma_i) \subseteq \mathsf{E}_{\mathsf{N}}.$$

The template T is computed using the algorithm introduced in Section 2.4. This algorithm was defined for tuple of arithmetic expressions but the $\mathsf{E}^n_{\mathsf{N}_{\sqrt{,/}}}$ subtype also embeds Boolean expressions. However these Boolean expressions are already square root and division free and therefore the template of Boolean expressions is a simple variable since we always can eliminate square roots and divisions from this kind of expressions.

Given σ a substitution, if $\sigma = [x_1 \mapsto e_1; ...; x_n \mapsto e_n]$, we denote $var(\sigma)$ the ordered tuple of the variables appearing in the domain of the substitution (of variables in $\mathcal{D}(\sigma)$), *i.e.*, $(x_1, ..., x_n)$ and $arg(\sigma)$ the tuple of the corresponding images, *i.e.*, $(e_1, ..., e_n)$. Using this decomposition, the following rule describes how the variable definition is transformed.

DEFINITION 3.23 (Variable definition transformation). A variable definition is transformed by commuting elements of its decomposition:

$$\begin{array}{l} \mathsf{let\ x} = Pp(T\sigma_1, ..., T\sigma_n)\ \mathsf{in\ p2} \quad \longrightarrow \\ \qquad\qquad \mathsf{let}\ var(\sigma_1) = Pp(arg(\sigma_1), ..., arg(\sigma_n))\ \mathsf{in\ p2[\ x} \mapsto T\ \mathsf{]} \end{array}$$

Therefore, if all the arguments of the substitutions are square root and division free then the body of this new variable definition is in $\mathsf{P_N}$. However the square root and division operations in the inlined template will then have to be eliminated and this elimination is exponential in the number of square roots, hence we try to compute a template that contains the smallest possible number of square root operations in order to keep the size of the transformed program in an acceptable range. This is done using the anti-unification algorithm we introduced in Section 2.4 with a few extra features. We first study how we can decompose the body in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ into its program part and its expression part and then present how we can tune the anti-unification algorithm on arithmetic to exactly fit this transformation. We define the following recursive algorithm *Decompose*, that computes from a program p in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$, its *program part* and its *expression part*.

DEFINITION 3.24 (Program and expression part decomposition). The program part is a meta-function where the final expression are abstracted:

Decompose(p) =

- if p $\in \mathsf{E}_{\mathsf{N}_{\sqrt{,/}}}$ then return $((fun\ x \to x), \mathsf{p})$
- if p = let y = a in p' then
  - (pp,ep) := *Decompose*(p')
  - return $((fun\ x \to \mathsf{let\ y = a\ in}\ \mathrm{pp}(x)), \mathrm{ep})$
- if p = if B then p_1 else p_2 then
  - (pp_1,ep_1) := *Decompose*(p_1)
  - (pp_2,ep_2) := *Decompose*(p_2)
  - return $((fun\ (x_1, x_2) \to \mathsf{if\ B\ then}\ \mathrm{pp}_1(x_1)\ \mathsf{else}\ \mathrm{pp}_2(x_2)), (\mathrm{ep}_1,\mathrm{ep}_2))$

Given Pp the program part, we denote $BV(Pp)$ the set of bound variables that appear in p (*i.e.*, the variables that are defined in the program part).

EXAMPLE 3.10.

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

The program p being in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$, neither the local variable definition 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. The program part only contains variable definitions and conditional expressions. Therefore we can state the correction of the variable definition transformation

PROPOSITION 3.7. *Given the program and variable of Definition 3.23, if* $FV(T) \cap BV(Pp) = \emptyset$ *and* $\forall\ \mathsf{v} \lll var(\sigma_1),\ \mathsf{v} \lll \mathsf{x} \vee \neg\mathsf{v} \lll \mathsf{p2}$ *then the rule from Definition 3.23 preserves the semantics*

Proof. We introduce a functional notation, given a term T and a substitution σ on $(\mathsf{x}_1, ..., \mathsf{x}_n)$ we denote T_σ the function $(x_1, ..., x_n) \longrightarrow \mathsf{t}[\mathsf{x}_1 \mapsto x_1, ..., \mathsf{x}_n \mapsto x_n]$ and $\overrightarrow{a}$ the tuple $(a_1, ..., a_n)$, this gives $T\sigma = T_\sigma(arg(\sigma))$. Therefore the following semantics equivalences:

- $\mathsf{let\ y = B\ in\ T}_\sigma(\overrightarrow{e}) \overset{sem}{=} \mathsf{T}_\sigma(\mathsf{let\ y = B\ in\ }(\overrightarrow{e}))$ when the variables in y are not free variables in T, we enforce this property in the construction of T
- $\mathsf{if\ F\ then\ T}_\sigma(\overrightarrow{e_1})\ \mathsf{else\ T}_\sigma(\overrightarrow{e_2}) \overset{sem}{=} \mathsf{T}_\sigma(\mathsf{if\ F\ then}\ \overrightarrow{e_1}\ \mathsf{else}\ \overrightarrow{e_2})$

enables the transformation of $Pp(T\sigma_1, ..., T\sigma_n)$ into $T_\sigma(Pp(arg(\sigma_1), ..., arg(\sigma_n)))$ and

- $\mathsf{let\ x} = T_\sigma(a)\ \mathsf{in\ p} \overset{sem}{=} \mathsf{let\ x = (let}\ var(\sigma) = a\ \mathsf{in}\ T\ )\ \mathsf{in\ p}$
- $\mathsf{let\ x = (let\ y}= a\ \mathsf{in}\ T)\ \mathsf{in\ p} \overset{sem}{=} \mathsf{let\ y} = a\ \mathsf{in\ p[x} \mapsto T]$
  when the variables in y are either in x or not free in p, we enforce this property in the construction of T

enables the transformation of:
$\mathsf{let\ x} = T_\sigma(Pp(arg(\sigma_1), ..., arg(\sigma_n)))\ \mathsf{in\ p}$
into
$\mathsf{let}\ var(\sigma_1) = Pp(arg(\sigma_1), ..., arg(\sigma_n))\ \mathsf{in\ p2[\ x} \mapsto T\ ]$ ◀

We now detail the effective template computation that is used to minimize the size of the output program.

3.7.3 Single Expression Decomposition

In a first step, we assume that the body of the variable definition does not contain any test, therefore its program part can only be nested variable definitions (*e.g.*, $fun\ x \rightarrow$ let y = a in let z = b in x) and its expression part is reduced to a single expression in $\mathsf{E}_{\mathsf{N}_{\sqrt{,/}}}$. We call the variable defined in the program part the *local variables*. Computing a most general $\{\sqrt{}, /\}$-template and the corresponding substitution of a singleton is quite trivial, it simply consists in substituting with variables the square root and division free sub expressions, this gives the following transformation:

EXAMPLE 3.11 (Simple variable definition transformation).

$$\mathsf{let\ x = (let\ z} = a + b\ \mathsf{in\ z}{+}d + \sqrt{c.d/e})\ \mathsf{in\ P} \longrightarrow$$
$$\mathsf{let\ (x_1,x_2,x_3) = (let\ z} = a + b\ \mathsf{in\ (z}{+}d, c.d, e))\ \mathsf{in\ P[x} \mapsto \mathsf{x}_1 + \sqrt{\mathsf{x}_2/\mathsf{x}_3}]$$

However, one crucial point is to avoid renaming square roots that have already been named. Since new variables are often defined using already existing variables the number of square roots in such a program might artificially and exponentially explode, *e.g.*,

EXAMPLE 3.12 (Renaming explosion).

let $x = a.\sqrt{b}$ in		let $x_1,x_2 = a,b$ in
let $y = x + c.\sqrt{b}$ in	would lead to	let $y_1,y_2,y_3,y_4 = x_1,x_2,c,b$ in
if $x + y > d$ then...		if $x_1.\sqrt{x_2} + y_1.\sqrt{y_2} + y_3.\sqrt{y_4} > c$ then ...

The formula in which we want to eliminate square roots and division now containing 3 different square roots.

The optimized solution records already named sub-expressions, *e.g.,*

EXAMPLE 3.13 (No Renaming Transformation).

let $x_1,x_2 = a,b$ in
let $y = x_1+c$ in
if $x_1 + y\ .\sqrt{x_2} > c$ then ...

It allows us to preserve the number of square roots that were in the input program. Therefore, we remember the already named sub-expressions in order to reuse these names as much as possible. We now explain how we compute the $\{\sqrt{}, /\}$-template of a set of expressions and also avoid the renaming by using the anti-unification algorithm introduced in Chapter 2.

3.7.4 Multiple Expression Decomposition

The decomposition of the set of expressions coming from the different test cases directly uses the anti-unification algorithm introduced in Section 2.4. As in the single expression case and as introduced in Proposition 2.18, we also want to avoid renaming by using the undefined element replacement but we are not always allowed to replace. Indeed, in the final template these expressions replacing undefined elements will appear as square roots and since, in our semantics, 0 can not absorb failures ($0.\sqrt{-1}$ fails), we have to make sure that the expressions we use are positives. For example, the following transformation is not allowed:

DEFINITION 3.25 (Unsafe Transformation).

let $x =$ if $y \geq 0$ then $\sqrt{y}$ else 0 in P $\longrightarrow$ let $x =$ if $y \geq 0$ then 1 else 0 in $P[x \mapsto x.\sqrt{y}]$

since $P[x \mapsto x.\sqrt{y}]$ would fail when $y < 0$. Therefore we have to define the $\mathcal{P}os$ set of positive expressions introduced in Proposition 2.18. Using the hypothesis that our program does not fail, we will take as positive expressions all the expressions that have been previously used under a square root (*e.g.,* square roots arguments coming from previously defined variable inlining), for example:

EXAMPLE 3.14 (Known Positive Expressions).

let $z = \sqrt{a}$ in
let $x =$ if F then $b.z + \sqrt{a}$ else 0 in P $\longrightarrow$ let $z = a$ in
let $x =$ if F then $b + 1$ else 0 in $P[x \mapsto x.\sqrt{z}]$

We know that z (*i.e.,* a) is positive since in the input program it is used under a square root in the definition of z and we explained at the beginning of section 3.3 that the programs we want to transform do not fail due to square roots of negative numbers. In order

to realize such transformation, we record during the transformation the different expressions that were arguments of square roots in the input programs and their corresponding named expression (*e.g.*, (*a*,z)) and use them as the $\mathcal{P}os$ set.

For this anti-unification process, we also might want to extend even the longest dag in order to produce the most suitable template. For example, assume we have the following variable definition:

$$\mathsf{let}\ (\mathsf{x,y,z}) = \mathsf{if}\ \mathsf{F}\ \mathsf{then}\ (\sqrt{a}, \sqrt{b}, \sqrt{b})\ \mathsf{else}\ (\sqrt{c}, \sqrt{c}, \sqrt{d})\ \mathsf{in}\ \mathsf{P}$$

The anti-unification would give a template with only two distinct square roots:

$$\sqrt{\mathsf{x}_1}, y_1.\sqrt{\mathsf{x}_1} + y_2.\sqrt{\mathsf{z}_1}, \sqrt{\mathsf{z}_1}$$

If all the variables of the tuple are used together *e.g.*, if P is $\mathsf{x} + \mathsf{y} - \mathsf{z} > 0$ then this solution is optimal, the inlined P having only 2 square roots: $(1 + \mathsf{y}_1)\sqrt{\mathsf{x}_1} + (\mathsf{y}_2 - 1)\sqrt{\mathsf{z}_1} > 0$. However if in P these variables are used separately *e.g.*, $\mathsf{x} > e \wedge \mathsf{y} > f \wedge \mathsf{z} > h$, then the second comparison, $\mathsf{y} > f$ would embed 2 square roots. In that case, an inlining with no sharing would be a better solution since each of the 3 comparisons only embeds one square root:

EXAMPLE 3.15 (Using Dag Extension).

$$\mathsf{let}\ (\mathsf{x,y,z}) = \mathsf{if}\ \mathsf{F}\ \mathsf{then}\ (a, b, b)\ \mathsf{else}\ (c, c, d)\ \mathsf{in}\ \sqrt{\mathsf{x}} > \mathsf{e} \wedge \sqrt{\mathsf{y}} > \mathsf{f} \wedge \sqrt{\mathsf{z}} > \mathsf{h}$$

Computation of such templates requires to extend all the input dags with undefined elements and replacing undefined elements by node duplication and pointer changes:

$$\begin{array}{c||c|c|c} \sqrt{\dot{1}}, \sqrt{\dot{2}}, \sqrt{\dot{2}} & a & b & \# \\ \hline \sqrt{\dot{1}}, \sqrt{\dot{1}}, \sqrt{\dot{2}} & c & d & \# \end{array} \quad = \quad \begin{array}{c||c|c|c} \sqrt{\dot{1}}, \sqrt{\dot{3}}, \sqrt{\dot{2}} & a & b & a \\ \hline \sqrt{\dot{1}}, \sqrt{\dot{3}}, \sqrt{\dot{2}} & c & d & d \end{array}$$

Therefore, we will also try to extend the input dags with few undefined elements in order to also compute such templates. We also want to ensure that the free variables of the template do not appear in the program part in order to apply the permutation as introduced in Proposition 3.7. This gives a new version of the anti-unification introduced in Definition 2.30 that is completely tailored to the transformation:

ALGORITHM 3.26 ($\{\sqrt{}, /\}$-constrained anti-unification for transformation). Given a set of expressions in $\mathsf{E}_{\mathsf{N}_{\sqrt{},/}}$, a set of forbidden variables S, the following algorithm computes a template of these expressions:

i) Transform every expression into its dag representation.

ii) Choose the length of the anti-unifier (has to be larger than the maximum)

iii) Extend all dags to the same length with undefined elements.

iv) Apply a permutation on the dag nodes identifiers with respect to right dependency

v) Replace undefined elements using either

- Node duplication
- A node dn from another dag such that $dn^{\top} \in \mathcal{P}os$
- A positive constant

vi) Compute the $\{\sqrt{}, /\}$-template node by node as in Definition 2.30, we restrict the use of the (EI) rule (see Definition 2.29), it is only used for a square root entire node (*i.e.*, whose index is bigger than 0) and not for sub-terms of nodes:

$(EI)\ ctmp(dn_i, dn_i) = dn_i \qquad$ when $FV(dn_i) \cap S = \emptyset$

Given these features, we can now defined the $Elim_{let}$ algorithm that, given a set of positive expressions $\mathcal{P}os$, transforms a variable definition in order to eliminate square roots and divisions from the body.

ALGORITHM 3.27 ($Elim_{let}$ Function). Given a variable definition let x = p1 in p2 where x$\in \mathcal{X}$, p1 $\in \mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ and p2 $\in$ P and a set of known positive expressions $\mathcal{P}os$, $Elim_{let}$ is the following transformation:

- Decompose p1 into its program part Pp and its expression part $e_1, ..., e_n$ using the *Decompose* function from Definition 3.24.
- Compute a $\{\sqrt{}, /\}$-template of $e_1, ..., e_n$, T, and the substitution $\sigma_1, ..., \sigma_n$ using $\{e \in \mathcal{P}os \mid FV(e) \cap BV(e) = \emptyset\}$ as positive expression set and $BV(Pp)$ as forbidden variables
- Add $\{e \mid \sqrt{e} \lll T\}$ to $\mathcal{P}os$
- Return:
  $(var(\sigma_1),\ Pp(arg(\sigma_1), ..., arg(\sigma_n)),\ \mathsf{p2}[\ \mathsf{x} \mapsto T\ ])$

This transformation eliminates square roots and divisions from the body and, given the Proposition 3.7, this transformation preserves the semantics. Therefore it satisfies the specification from Definition 3.21. This transformation is now used to define the main algorithm that eliminates square roots and divisions from programs.

3.8 MAIN TRANSFORMATION

In Section 3.6 we defined the $Elim_{\mathbb{B}}$ function that eliminates square roots and divisions from Boolean expressions. In Section 3.7.2 we defined the $Elim_{let}$ function that removes square roots and divisions from variable definition bodies. We now combine both of these functions to define the transformation of any program p $\in$ P into an equivalent one in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$:

DEFINITION 3.28 (The $Elim_{\mathsf{P}}$ function). We define the $Elim_{\mathsf{P}}$ recursive function on any program p in P:

$Elim_{\mathsf{P}}(\mathsf{p}) =$

- if p $\in \mathsf{E}_u$ is a Boolean expression then return $Elim_{\mathbb{B}}(\mathsf{p})$
- if p $\in \mathsf{E}_u$ is an arithmetic expression then return p
- if p = (p1,p2) then return ($Elim_{\mathsf{P}}$(p1),$Elim_{\mathsf{P}}$(p1))
- if p = if F then p1 else p2 then return if $Elim_{\mathsf{P}}$(F) then $Elim_{\mathsf{P}}$(p1) else $Elim_{\mathsf{P}}$(p1)

- if p = let x = b in sc then
  - nb1 := $Elim_{\mathsf{P}}(\mathsf{b})$
  - (x',nb,nsc) := $Elim_{let}(\mathsf{x},\mathsf{nb1},\mathsf{p})$
  - return let x' = nb in $Elim_{\mathsf{P}}(\mathsf{nsc})$

This algorithm preserves the semantics of programs and eliminates the square roots and divisions from variable definitions and Boolean expressions:

PROPOSITION 3.8 ($Elim_{\mathsf{P}}$ Correctness).
The function $Elim_{\mathsf{P}}$ satisfies the following predicate:

$$\forall p \in \mathsf{P},\ Elim_{\mathsf{P}}(\mathsf{p}) \in \mathsf{P}_{N_{\sqrt{,/}}} \ \wedge\ sem_ty_eq(\mathsf{p}, Elim(\mathsf{p}))$$

Proof. By induction on P, using the specification of $Elim_{\mathbb{B}}$ (see Definition 3.19) and $Elim_{let}$ (see Definition 3.21) ◀

Combining this transformation with the program normalization enables the elimination of square roots and divisions in any program in Prog:

DEFINITION 3.29. If *Pnorm* transforms any program in Prog into a program in P using the rules from Definition 3.17 we define the following function *Elim*:

$$Elim(p) = Elim_{\mathsf{P}}(Pnorm(p))$$

This function satisfies the specification from Definition 3.29:

THEOREM 3.9 (Main *Elim* function).

$$\forall p \in \mathsf{Prog},\ Elim(\mathsf{p}) \in \mathsf{P}_{N_{\sqrt{,/}}} \ \wedge\ sem_ty_eq(\mathsf{p}, Elim(\mathsf{p}))$$

Conclusion We have defined function *Elim* that transforms any program in Prog into an equivalent one in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$. This transformation makes the Boolean computations appearing in the input program independent from any square root and division computation. We present in Chapter 6 an OCamL implementation of this transformation. Chapter 5 provides a PVS specification of this transformation along with the correctness proof, *i.e.*, the proof of Theorem 3.9. But first in Chapter 4 we detail an extension of this transformation that enables the transformation of programs that contain function definitions, still using this anti-unification principle.

4

Transforming Functions and Function Calls

We now want to extend the language the transformation applies to. The language defined in Chapter 3 contains many of the core features used in the embedded systems the transformation targets, yet, these programs also use function definitions. As we already explained we could inline the function definitions and then eliminate the square roots and divisions using the *Elim* algorithm defined in Section 3.8. However, this process would increase the size of the program and remove the intention of the programmer that introduced those functions for modularity or readability reasons.

Indeed our goal is still to have Boolean expressions completely independent from square roots or divisions computations and functions can contain these operations not only in their definitions but also in their calls *e.g.*,

Example 4.1 (Functions with square roots).

$$\begin{array}{l} \mathsf{let\ f(x,y) = (x + y)\ /\ y\ ;} \\ \qquad\qquad \vdots \\ \ldots\ \mathsf{f}(a + \sqrt{b}, c) > d\ \ldots \end{array}$$

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

Example 4.2 (Functions full inlining).

$$(a + \sqrt{b} + c)\ /\ c > d\ \ldots$$

But this is not what we are looking for. In this chapter, we present an algorithm that do not increase too much the size of the produced code but also preserves the structure of the program. The introduction of the functions definitions in our transformation can also be handled using the constrained anti-unification we defined in Chapter 2. Indeed a function call is somehow similar to a succession of variable definitions *e.g.*,:

Example 4.3 (Function inlining with variable definition).

$$\begin{array}{lcl} \mathsf{let\ f(x,y) = (x + y)\ /\ y\ ;} & & \vdots \\ \qquad\qquad \vdots & \longrightarrow & \mathsf{let\ (x,y) =} (a + \sqrt{b}, c)\ \mathsf{in} \\ \ldots\ \mathsf{f}(a + \sqrt{b}, c) > d\ \ldots & & \mathsf{let\ f = (x + y)\ /\ y\ in\ f} > d\ \ldots \end{array}$$

We already know how to transform this second program using the $Elim_{let}$ function introduced in Section 3.7, using tuples and templates to partially inline the definitions. We can therefore imagine the same kind of transformation for a function:

EXAMPLE 4.4 (Function transformation with templates).

let f(x,y) = (x + y) / y ;		let f(x_1, x_2, y) = (x_1 + y, x_2, y) ;
⋮	$\longrightarrow$	⋮
... f($a + \sqrt{b}$,c) > d ...		... let (f_1,f_2,f_3) = f(a,b,c) in (f_1 + $\sqrt{f_2}$) / f_3 > d ...

In order to present the extensions of our transformation with functions we first extend the language and the already existing transformation with function definitions. Then we explain how to transform functions in order to remove square roots and divisions both from their definitions and from their calls arguments. Finally we present a set of conditions making this transformation with anti-unification effective since it is not always possible.

4.1 LANGUAGE EXTENSION

The first step to extend the language the transformation applies on consists in adding function definitions and calls in the syntax of the language. We redefine the type Prog:

DEFINITION 4.1 (Language with functions).

Prog :=	$\mathcal{C}$	| fst Prog	| Prog op Prog
	| $\mathcal{X}$	| snd Prog	| if Prog then Prog else Prog
	| uop Prog	| (Prog, Prog)	| let $\mathcal{V}$ = Prog in Prog
	| $\mathcal{X}$ ($\mathcal{V}$)	| letf $\mathcal{X}$ $\mathcal{V}$: $Type \rightarrow Type$ = Prog ; Prog	

where *Type* corresponds to the Definition 3.2, therefore we only define first order functions. For clarity and concision, we might forget the type of the function definitions when we do not need it.

Consequently, we need to extend the type system and semantics of the language, for the already existing constructors the rules are the same, therefore we only give the rules for the application and function definition. ones:

DEFINITION 4.2 (Type and semantics extension). First we extend the type system with functional types:

$$Type_f \ := \ Type \mid Type \rightarrow Type$$

And give the corresponding rules:

$$\frac{\Gamma \vdash \mathsf{f} : A \rightarrow B \quad \Gamma \vdash e : A}{\Gamma \vdash \mathsf{f}(e) : B} \qquad \frac{\Gamma \oplus (\mathsf{x} : A) \vdash b : B \quad \Gamma, \mathsf{f} : A \rightarrow B \vdash sc : C}{\Gamma \vdash \mathsf{letf\ f\ x} : A \text{ -> } B = b;\ sc : C}$$

Then we define the semantics of functions using closures and call by value:

$$\frac{(\mathsf{f}, <\mathsf{x},\mathsf{b},E>) \in Env \quad Env \vdash [\![\mathsf{e}]\!] = u \quad E \oplus (\mathsf{x},u) \vdash [\![\mathsf{b}]\!] = v}{Env \vdash [\![\mathsf{f(e)}]\!] = v}$$

$$\frac{Env,\ (\mathsf{f},<\mathsf{x},\mathsf{b},E>) \vdash [\![\mathsf{sc}]\!] = v}{Env \vdash [\![\mathsf{letf\ f\ x} : \mathsf{A \text{ -> } B = b;\ sc}]\!] = v}$$

$$\frac{Env \vdash [\![\mathsf{e}]\!] = Fail}{Env \vdash [\![\mathsf{f(e)}]\!] = Fail}$$

As we did in Section 3.1, we will require the program to be in a certain normal form in order to transform it. This form prevents defining functions inside expressions (*e.g.*, in binary operators), just like we did not want variable definitions to appear in such expressions previously. Moreover function definitions will only be allowed at top level. We define this new syntactic normal form, it is mutually recursive since functions calls are allowed to contain test and variable definitions.

DEFINITION 4.3 (Program with function normal form). We introduce a new definition for the E_u, E and P types introduced in Definition 3.5.

$$\begin{array}{rcl} \mathsf{E}_u & := & \mathcal{X} \mid \mathcal{C} \mid \mathsf{uop}\ \mathsf{E}_u \mid \mathsf{E}_u\ \mathsf{op}\ \mathsf{E}_u \mid \mathsf{fst}\ \mathsf{E}_u \mid \mathsf{snd}\ \mathsf{E}_u \mid \mathcal{X}(\mathsf{P}) \\ \mathsf{E} & := & (\mathsf{E}, \mathsf{E}) \mid \mathsf{E}_u \\ \mathsf{P} & := & \mathsf{let}\ \mathcal{V} = \mathsf{P}\ \mathsf{in}\ \mathsf{P} \mid \mathsf{if}\ \mathsf{P}\ \mathsf{then}\ \mathsf{P}\ \mathsf{else}\ \mathsf{P} \mid \mathsf{E} \end{array}$$

and add the $\mathsf{P}^\top$ type for the top level:

$$\mathsf{P}^\top := \mathsf{let}\ \mathcal{V} = \mathsf{P}\ \mathsf{in}\ \mathsf{P}^\top \mid \mathsf{letf}\ \mathcal{X}\ \mathcal{V} : Type \to Type = \mathsf{P}\ ;\ \mathsf{P}^\top \mid \mathsf{P}$$

For this transformation we assume that the functions are already defined at top level and therefore we do not provide a mechanism to take function definitions out of the other structures. However we still use the rules from Definition 3.17 to normalize the other parts of the program.

Given the type of the input programs we now need to extend the type of the output programs. We do not want to inline the function definitions but to transform them in order to prevent their result from depending on square roots and divisions. Therefore, like in the variable definition case where we wanted to eliminate square roots and divisions from the body, we want not only the function bodies but also the function arguments to be square root and division free. This gives the following target language:

DEFINITION 4.4 (Target language with functions). We reuse the sets of operators and languages defined in Section 3.2.2 and we define some new languages that allow square roots and divisions in Boolean expressions. As for the input language the definitions are now mutually recursive:

- the different sets of numerical expressions, they allows function calls on square root and division free programs:
  - $\mathsf{N} := Nuop\ \mathsf{N} \mid \mathsf{N}\ Nbop\ \mathsf{N} \mid \mathsf{fst}\ \mathsf{N} \mid \mathsf{snd}\ \mathsf{N} \mid \mathcal{X} \mid \mathcal{C} \mid \mathcal{X}(\mathsf{P_N})$
  - $\mathsf{N}_{\surd,/} := Nuop_{\surd}\ \mathsf{N}_{\surd,/} \mid \mathsf{N}_{\surd,/}\ Nbop_{/}\ \mathsf{N}_{\surd,/} \mid \mathsf{fst}\ \mathsf{N}_{\surd,/} \mid \mathsf{snd}\ \mathsf{N}_{\surd,/} \mid \mathcal{X} \mid \mathcal{C} \mid \mathcal{X}(\mathsf{P_N})$
- the subsets of Boolean expressions with or without square roots and divisions:
  - $\mathsf{B}_{\surd,/} := \begin{array}{lll} Buop\ \mathsf{B}_{\surd,/} & \mid \mathsf{fst}\ \mathsf{B}_{\surd,/} & \mid \mathsf{B}_{\surd,/}\ Bbop\ \mathsf{B}_{\surd,/} \mid \mathcal{X} \\ \mid \mathcal{X}(\mathsf{P}) & \mid \mathsf{snd}\ \mathsf{B}_{\surd,/} & \mid \mathsf{E}_u\ Cbop\ \mathsf{E}_u \mid \mathcal{C} \end{array}$

- $\mathsf{B}_{let} := \; Buop\ \mathsf{B}_{let} \mid \mathsf{B}_{let}\ Bbop\ \mathsf{B}_{let} \mid \mathsf{N}\ Cbop\ \mathsf{N} \mid \mathcal{X}$
  $\mid (\mathsf{B}_{let}, \mathsf{B}_{let}) \mid \mathsf{fst}\ \mathsf{B}_{let} \mid \mathsf{snd}\ \mathsf{B}_{let} \mid \mathcal{C}$
  $\mid \mathsf{let}\ \mathcal{V} = \mathsf{B}_{let}\ \mathsf{in}\ \mathsf{B}_{let} \mid \mathcal{X}(\mathsf{P})$

- the sets of expressions keep the same definition:
  - $\mathsf{E_N} := \mathsf{N} \mid \mathsf{B}_{let} \mid (\mathsf{E_N}, \mathsf{E_N})$
  - $\mathsf{E}_{\mathsf{N}_{\surd,/}} := \mathsf{N}_{\surd,/} \mid \mathsf{B}_{let} \mid (\mathsf{E}_{\mathsf{N}_{\surd,/}}, \mathsf{E}_{\mathsf{N}_{\surd,/}})$
  - $\mathsf{E}_{\mathsf{B}_{\surd,/}} := \mathsf{N} \mid \mathsf{B}_{\surd,/} \mid (\mathsf{E}_{\mathsf{B}_{\surd,/}}, \mathsf{E}_{\mathsf{B}_{\surd,/}})$

- as previously, the programs that do not contain any square root or division
  $\mathsf{P_N} := \mathsf{let}\ \mathcal{V} = \mathsf{P_N}\ \mathsf{in}\ \mathsf{P_N} \mid \mathsf{if}\ \mathsf{P_N}\ \mathsf{then}\ \mathsf{P_N}\ \mathsf{else}\ \mathsf{P_N} \mid \mathsf{E_N}$

- the programs whose final expressions may contain divisions or square roots only in Boolean parts
  $\mathsf{P}_{\mathsf{B}_{\surd,/}} := \mathsf{let}\ \mathcal{V} = \mathsf{P}\ \mathsf{in}\ \mathsf{P}_{\mathsf{B}_{\surd,/}} \mid \mathsf{if}\ \mathsf{P}\ \mathsf{then}\ \mathsf{P}_{\mathsf{B}_{\surd,/}}\ \mathsf{else}\ \mathsf{P}_{\mathsf{B}_{\surd,/}} \mid \mathsf{E}_{\mathsf{B}_{\surd,/}}$

- the programs containing square roots or divisions only in final numerical expressions
  $\mathsf{P}_{\mathsf{N}_{\surd,/}} := \mathsf{let}\ \mathcal{V} = \mathsf{P_N}\ \mathsf{in}\ \mathsf{P}_{\mathsf{N}_{\surd,/}} \mid \mathsf{if}\ \mathsf{P_N}\ \mathsf{then}\ \mathsf{P}_{\mathsf{N}_{\surd,/}}\ \mathsf{else}\ \mathsf{P}_{\mathsf{N}_{\surd,/}} \mid \mathsf{E}_{\mathsf{N}_{\surd,/}}$

The language $\mathsf{B}_{\surd,/}$ is the set of Boolean expressions with square roots and divisions. The languages using this set will be used as intermediate languages since the global strategy to eliminate square roots and divisions will change. Indeed, square root and divisions in Boolean expressions can be eliminated on place (without modifying other part of the program), thus we will only do this elimination as the final step of our transformation, after all the variable and function definitions have been transformed.

EXAMPLE 4.5 (Program subtype).

$$\mathsf{let}\ \mathsf{x} = a\ \mathsf{in}\ (\sqrt{b} > c, \mathsf{x}) \in \mathsf{P}_{\mathsf{B}_{\surd,/}}$$
$$\mathsf{let}\ \mathsf{x} = a\ \mathsf{in}\ (\sqrt{b} > c, \sqrt{\mathsf{x}}) \notin \mathsf{P}_{\mathsf{B}_{\surd,/}}$$

Given these new definitions, we now define the target language. It only allows square root and division free programs to appear in the body of a function or variable definitions :

DEFINITION 4.5 (Target language). The language the transformation targets is:

$$\mathsf{P}^{\top}_{\mathsf{N}_{\surd,/}} := \mathsf{let}\ \mathcal{V} = \mathsf{P_N}\ \mathsf{in}\ \mathsf{P}^{\top}_{\mathsf{N}_{\surd,/}} \mid \mathsf{letf}\ \mathcal{X}\ \mathcal{V} : Type \rightarrow Type = \mathsf{P_N}\ ;\ \mathsf{P}^{\top}_{\mathsf{N}_{\surd,/}} \mid \mathsf{P}_{\mathsf{N}_{\surd,/}}$$

Of course our goal is still to preserve the semantics between the input and the transformed program, therefore the specification of the transformation with functions is defined as follows:

DEFINITION 4.6 (Transformation specification).
The transformation of program with functions, called *FElim*, has to satisfy the following predicate:

$$\forall \mathsf{p} \in \mathsf{P}^{\top},\ FElim(\mathsf{p}) \in \mathsf{P}^{\top}_{\mathsf{N}_{\surd,/}} \wedge sem_ty_eq(\mathsf{p}, FElim(\mathsf{p}))$$

In order to extend the algorithm introduced in Chapter 3, we first define in Section 4.2 two transformations, that are similar to the $Elim_{let}$ function. These transformations allow us to eliminate square roots and divisions from function definitions and calls.

4.2 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 found a way to do both of these transformations completely independently. The following example illustrates an elimination of square roots and divisions in the function calls:

Example 4.6 (Function input transformation).

$$\begin{array}{l}
\mathsf{letf\ f\ x} : \mathbb{R} \to \mathbb{R} = 3\mathsf{x} + \sqrt{a}; \\
\mathsf{f}(b) + \mathsf{f}(c + d\sqrt{e}) > 0 \\
\qquad\qquad \longrightarrow \\
\mathsf{letf\ f}\ (x_1,\ x_2,\ x_3) : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} = 3.(x_1 + x_2\sqrt{x_3}) + \sqrt{a}; \\
\mathsf{f}(b, 0, 0) + \mathsf{f}(c, d, e) > 0
\end{array}$$

Where $x_1 + x_2\sqrt{x_3}$ is a template of b and $c + d\sqrt{e}$.

This transformation has eliminated all the square roots and divisions that used to appear in the arguments of the calls of function f. The next example illustrates the elimination in the function body:

Example 4.7 (Function body transformation).

$$\begin{array}{l}
\mathsf{letf\ f}\ (x_1,\ x_2,\ x_3) : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} = 3.(x_1 + x_2\sqrt{x_3}) + \sqrt{a}; \\
\mathsf{f}(b, 0, 0) + \mathsf{f}(c, d, e) > 0 \\
\qquad\qquad \longrightarrow \\
\mathsf{letf\ f}\ (x_1,\ x_2,\ x_3) : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} = (3.x_1, 3x_2, x_3, a); \\
\mathsf{let}\ (y_1, y_2, y_3, y_4) = \mathsf{f}(b, 0, 0)\ \mathsf{in} \\
\mathsf{let}\ (z_1, z_2, z_3, z_4) = \mathsf{f}(c, d, e)\ \mathsf{in} \\
y_1 + y_2\sqrt{y_3} + \sqrt{y_4} + z_1 + z_2\sqrt{z_3} + \sqrt{z_4} > 0
\end{array}$$

Where $y_1 + y_2\sqrt{y_3} + \sqrt{y_4}$ and $z_1 + z_2\sqrt{z_3} + \sqrt{z_4}$ are templates of $3.(x_1 + x_2\sqrt{x_3}) + \sqrt{a}$

There are no more divisions in the body of the function, therefore 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.

We now formally define the transformation of the function input, still using a decomposition and the constrained anti-unification introduced in Chapter 2.

4.2.1 Function input transformation

In this section we aim at defining a transformation that removes square roots and divisions from the arguments of a function call, indeed in $\mathsf{P}^{\top}_{\mathsf{N}_{\sqrt{,/}}}$ the function calls have to

be made on programs in $\mathsf{P_N}$, however as mentioned before, the elimination in Boolean expression will be done later. Therefore we want to define a transformation $Elim_{fin}$ that has the following specification:

DEFINITION 4.7 ($Elim_{fin}$ specification). Given $\mathsf{f} \in \mathcal{X}$, $\mathsf{x} \in \mathcal{V}$, $A, B \in Type$, $\mathsf{b} \in \mathsf{P}$ and $\mathsf{sc} \in \mathsf{P}$, we want the $Elim_{fin}$ function to compute x', A', b' and sc' such that:

$$\forall\, Env, [\![\mathsf{letf\ f\ x} : A \to B = \mathsf{b};\ \mathsf{sc}]\!]_{Env} \neq Fail \Longrightarrow$$
$$[\![\mathsf{letf\ f\ x} : A \to B = \mathsf{b};\ \mathsf{sc}]\!]_{Env} = [\![\mathsf{letf\ f\ x'} : A' \to B = \mathsf{b'};\ \mathsf{sc'}]\!]_{Env} \wedge$$
$$\forall\, \mathsf{f(p)} \lll \mathsf{sc'},\ \mathsf{p} \in \mathsf{P}_{\mathsf{B}_{\surd,/}}$$

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:

EXAMPLE 4.8 (Elimination in function calls).

$$\mathsf{letf\ f\ (x,y)} : \mathbb{B} \times \mathbb{R} \to \mathbb{R} = \mathsf{if\ x\ then\ y\ else\ 0};$$
$$\mathsf{f(if}\ \sqrt{s} > t\ \mathsf{then}\ (\sqrt{b} > c, d.\sqrt{e})\ \mathsf{else}\ (False, 1)) > 0$$
$$\longrightarrow$$
$$\mathsf{letf\ f}\ (x,\ y_1,\ y_2) : \mathbb{B} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} = \mathsf{if}\ x\ \mathsf{then}\ y_1.\sqrt{y_2}\ \mathsf{else\ 0};$$
$$\mathsf{f(if}\ \sqrt{s} > t\ \mathsf{then}\ (\sqrt{b} > c, d, e)\ \mathsf{else}\ (False, 1, 0)) > 0$$

As in the variable definition case introduced in Section 3.7, this transformation is done using a decomposition and the constrained anti-unification. However, in this case, we have to compute a decomposition of the scope to extract the different function calls of f. We assume that we have no nested calls of a function, indeed in that case we are not able to transform the program as we will see in Section 4.4.

DEFINITION 4.8 (Scope f-decomposition). Given f a function variable and a program p in $\mathsf{P}^\top$, this program can be decomposed into

- a *scope program part scf* of type $\mathsf{P}^n \longrightarrow \mathsf{P}^\top$ that represent the structure of p where the function calls of f have been abstracted. Since there are no nested calls, this function is analogous to the program part computed by the *Decompose* function

- a list of programs $(\mathsf{c}_1, ..., \mathsf{c}_n)$ in P^n such that

$$scf(\mathsf{f}(\mathsf{c}_1), ..., \mathsf{f}(\mathsf{c}_n)) = \mathsf{p}$$

  Using the *Decompose* function from Definition 3.24, every program c_i of this list is then decomposed with a program part Pp_i and a set of expressions $e_{i,1}, ..., e_{i,m_i}$ such that

$$\forall i, \mathsf{c}_i = Pp_i(e_{i,1}, ..., e_{i,m_i})$$

  Then using constrained anti-unification we compute the common template T of the set $\{e_{i,j} \mid 1 \leq i \leq n \wedge 1 \leq j \leq m_i\}$, the set of the corresponding substitutions $\{\sigma_{1,1}, ..., \sigma_{n,m_n}\}$ that $\forall\, i, j,\ \mathcal{I}(\sigma_{i,j}) \subset \mathsf{E}_{\mathsf{B}_{\surd,/}}$.

Therefore we have: $\mathsf{p} = scf(\mathsf{f}(Pp_1(T\sigma_{1,1}, ..., T\sigma_{1,m_1})), ..., \mathsf{f}(Pp_n(T\sigma_{n,1}, ..., T\sigma_{n,m_n})))$

By using the properties of the *Decompose* function, we already have the commutation of the program parts and the template T when $FV(T) \cap BV(Pp_i) = \emptyset$ which is ensured in the construction of T:

$$\forall\, i, Pp_i(T\sigma_{i,1}, ..., T\sigma_{i,m_i}) \overset{sem}{=} T_\sigma(Pp_i(arg(\sigma_{i,1}), ..., arg(\sigma_{i,m_i})))$$

Where T_σ is the function that corresponds to T as defined in the proof of Proposition 3.7. We now define the commutation rule between the template and the scope program part:

DEFINITION 4.9 (Function input transformation). We inline the template of the calls in the body of the function definition:

$$\begin{array}{l}
\mathsf{letf\ f\ x} : A \to B = \mathsf{b}; \\
scf(\mathsf{f}(Pp_1(T\sigma_{1,1}, ..., T\sigma_{1,m_1})), ..., \mathsf{f}(Pp_n(T\sigma_{n,1}, ..., T\sigma_{n,m_n}))) \\
\qquad \longrightarrow \\
\mathsf{letf\ f}\ var(\sigma_1) : A' \to B = \mathsf{b}[\ \mathsf{x} \mapsto T\ ]; \\
scf(\mathsf{f}(Pp_1(arg(\sigma_{1,1}), ..., arg(\sigma_{1,m_1}))), ..., \mathsf{f}(Pp_n(arg(\sigma_{n,1}), ..., arg(\sigma_{n,m_n}))))
\end{array}$$

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

Given some conditions on the variables of the template this rule preserves the semantics:

PROPOSITION 4.1 (Function input transformation correctness). *If*

(Hp) $\forall\, i, \forall v \in FV(T),\ v \notin BV(Pp_i)$

(HT) $\forall\, v \in FV(T),\ v \leqslant \mathsf{x} \vee v \notin BV(scf)$

$(H\mathsf{x'})$ $\forall\, v \leqslant \mathsf{x'},\ v \leqslant \mathsf{x} \vee v \notin FV(\mathsf{b})$

where $\mathsf{x'}$ *is,* $var(\sigma)$*, then the rule from Definition 4.9 preserves the semantics.*

Proof. The commutation of T and Pp_i preserves the semantics using (Hp). Then, for every call of f in sc and for every environment Env where this call is evaluated in, with $(\mathsf{f},<\mathsf{x},\mathsf{b},E>) \in Env$, $v \notin BV(scf)$ means that v is not redefined in scf thus its value in Env and E is the same. This gives:

$$\begin{array}{rll}
[\![\mathsf{f}(T_\sigma(p))]\!]_{Env} & = [\![\mathsf{b}]\!]_{E\oplus(\mathsf{x},[\![T_\sigma(p)]\!]_{Env})} & \\
& = [\![\mathsf{b}]\!]_{E\oplus(\mathsf{x},[\![T]\!]_{Env\oplus(\mathsf{x'},[\![p]\!]_{Env})})} & \\
& = [\![\mathsf{b}]\!]_{E\oplus(\mathsf{x},[\![T]\!]_{E\oplus(\mathsf{x'},[\![p]\!]_{Env})})} & \text{using } (HT) \text{ hypothesis} \\
& = [\![\mathsf{b}]\!]_{E\oplus(\mathsf{x'},[\![p]\!]_{Env})\oplus(\mathsf{x},[\![T]\!]_{E\oplus(\mathsf{x'},[\![p]\!]_{Env})})} & \text{using } (H\mathsf{x'}) \text{ hypothesis} \\
& = [\![\mathsf{b}[\mathsf{x} \mapsto T]]\!]_{E\oplus(\mathsf{x'},[\![p]\!]_{Env})} & \\
& = [\![\mathsf{f}(p)]\!]_{Env\oplus(\mathsf{f},<\mathsf{x'},\mathsf{b}[\mathsf{x}\mapsto T],E>)} &
\end{array}$$

And $< \mathsf{x'}, \mathsf{b}[\mathsf{x} \mapsto T], E >$ corresponds to the new definition of f. ◀

Application of the rule of Definition 4.9 enables the elimination of all square roots and divisions from the numerical expression corresponding to the arguments of the calls of the function f. This transformation inline the template of these calls in the body of the

definition. This transformation allows us to define the $Elim_{fin}$ function that respects the specification from Definition 4.7:

DEFINITION 4.10 ($Elim_{fin}$). The $Elim_{fin}$ function is the composition of 2 steps:

- Decompose the scope according to the Definition 4.8 with respect of the conditions (Hp), (HT) and $(H\mathsf{x}')$ introduced in Proposition 4.1
- Apply the rule of Definition 4.9

We have defined a strategy that enables the elimination of the square roots and divisions from the arguments of a function call, we now introduce the transformation of the body of the definition to remove square roots and divisions from it.

4.2.2 Function output transformation

In this section we want to remove square roots and divisions from the body of a function definition since in $\mathsf{P}^{\top}_{\mathsf{N}_{\surd,/}}$ the function bodies have to be programs in P_{N}. As in the input transformation the elimination of these operations in Boolean expressions will be done later. Therefore we want to define a transformation $Elim_{fout}$ that has the following specification:

DEFINITION 4.11 ($Elim_{fout}$ specification). Given $\mathsf{f} \in \mathcal{X}$, $\mathsf{x} \in \mathcal{V}$, $A, B \in Type$, $\mathsf{b} \in \mathsf{P}$ and $\mathsf{sc} \in \mathsf{P}$, we want the $Elim_{fout}$ function to compute B', b' and sc' such that:

$$\forall\, Env, [\![\mathsf{letf\ f\ x} : A \to B = \mathsf{b};\ \mathsf{sc}]\!]_{Env} \neq Fail \Longrightarrow$$
$$[\![\mathsf{letf\ f\ x} : A \to B = \mathsf{b};\ \mathsf{sc}]\!]_{Env} = [\![\mathsf{letf\ f\ x} : A \to B' = \mathsf{b}';\ \mathsf{sc}']\!]_{Env} \wedge$$
$$\mathsf{b}' \in \mathsf{P}_{\mathsf{B}_{\surd,/}}$$

This means that the new function can only return numerical values that do not contain square root or division operations, these operations are still allowed in Boolean expressions:

EXAMPLE 4.9 (Elimination in function body).

$$\mathsf{letf\ f\ x} : \mathbb{R} \to \mathbb{B} \times \mathbb{R} = \mathsf{if\ x} > 0 \ \mathsf{then}\ (x/2 > 3, \sqrt{\mathsf{x}})\ \mathsf{else}\ (False, 1/\mathsf{x});$$
$$\ldots\mathsf{f(a)}$$
$$\longrightarrow$$
$$\mathsf{letf\ f\ x} : \mathbb{R} \to \mathbb{B} \times \mathbb{R} \times \mathbb{R} = \mathsf{if\ x} > 0 \ \mathsf{then}\ (x/2 > 3, \mathsf{x}, 1)\ \mathsf{else}\ (False, 1, \mathsf{x});$$
$$\ldots\mathsf{let}\ (x_1, x_2, x_3) = \mathsf{f(a)}\ \mathsf{in}\ (x_1, \sqrt{x_2}/x_3)$$

The transformation of the output is even closer to the variable definition case than the input one, indeed a function call can somehow be turned into a variable definition:

EXAMPLE 4.10. When $\forall v \in FV(b), v \leqslant \mathsf{x}$:

$$\mathsf{letf\ f\ x} : A \to B = \mathsf{b};\ \ldots\ \mathsf{f(a)}$$
$$\longrightarrow$$
$$\ldots\mathsf{let\ x = a\ in\ let\ f = b\ in\ f}$$

Since a does not contain any square root or division (due to the $Elim_{fin}$ transformation)

then, using the variable definition transformation $Elim_{let}$ we get

...let x = a in let f' = b' in $\mathsf{T_b}$

where $\mathsf{T_b}$ is a template of b. Yet b is the same for all the calls of f, thus this program can be re-factorized:

letf f x : $A \to B'$ = b'; ... let f' = f(a) in $\mathsf{T_b}$

Once again, the transformation relies on a decomposition, this time the body of the function is decomposed as in the variable definition case (see Definition 3.22) with a program part Pp, a template T and a list of substitution $\sigma_1, ..., \sigma_n$ such that:

$$\mathsf{b} = Pp(T\sigma_1, ..., T\sigma_n)$$

Given this decomposition we define the following transformation:

DEFINITION 4.12 (Function output transformation). We inline the template of the body in the scope of the function:

letf f x : $A \to B = Pp(T\sigma_1, ..., T\sigma_n)$;
scf(f(a_1),...,f(a_m))

$\longrightarrow$

letf f x : $A \to B' = Pp(arg(\sigma_1), ..., arg(\sigma_n))$;
scf(let $var(\sigma)$ = f(a_1) in T,...,let $var(\sigma)$ = f(a_m) in T)

where B' is the type of the $arg(\sigma_i)$

Once again we have to impose some conditions on the variables that are used in the template in order to allow the elements to commute.

PROPOSITION 4.2 (Function output transformation correctness). *If*

(Hp) $\forall v \in FV(T),\ v \notin BV(Pp)$

(HT_1) $\forall\ v \in FV(T),\ v \lll \mathsf{x'} \vee \neg\ v \lll \mathsf{x}$

(HT_2) $\forall\ v \in FV(T),\ (v \lll \mathsf{x'} \vee v \notin BV(scf))$

$(H\mathsf{f})$ $\forall\ v \in FV(T),\ (v \neq \mathsf{f})$

where x' *is,* $var(\sigma)$*, then the rule from Definition 4.9 preserves the semantics.*

Proof. (Hp) allows the commutation of T and Pp, we denote nb = $Pp(arg(\sigma_1), ..., arg(\sigma_n))$. Then, for every call of f in sc and for every environment Env where this call is evaluated

in, with (f,<x,b,E>) $\in Env$, we have:

$$
\begin{aligned}
[\![\mathsf{f}(a_i)]\!]_{Env} &= [\![\mathsf{b}]\!]_{E\oplus(\mathsf{x},[\![a_i]\!]_{Env})} \\
&= [\![T_\sigma(\mathsf{nb})]\!]_{E\oplus(\mathsf{x},[\![a_i]\!]_{Env})} \\
&= [\![T]\!]_{E\oplus(\mathsf{x},[\![a_i]\!]_{Env})\oplus(\mathsf{x'},[\![\mathsf{nb}]\!]_{E\oplus(\mathsf{x},[\![a_i]\!]_{Env})})} \\
&= [\![T]\!]_{E\oplus(\mathsf{x'},[\![\mathsf{nb}]\!]_{E\oplus(\mathsf{x},[\![a_i]\!]_{Env})})} && \text{using } (HT_1) \text{ hypothesis} \\
&= [\![T]\!]_{Env\oplus(\mathsf{x'},[\![\mathsf{nb}]\!]_{E\oplus(\mathsf{x},[\![a_i]\!]_{Env})})} && \text{using } (HT_2) \text{ hypothesis} \\
&= [\![T]\!]_{Env\oplus(\mathsf{x'},[\![\mathsf{f}(a_i)]\!]_{Env\oplus(\mathsf{f},<\mathsf{x},\mathsf{nb},E>)})} \\
&= [\![\mathsf{let\ x'} = \mathsf{f}(a_i)\ \mathsf{in}\ T]\!]_{Env\oplus(\mathsf{f},<\mathsf{x},\mathsf{nb},E>)} && \text{using } (Hf) \text{ hypothesis}
\end{aligned}
$$

And $< \mathsf{x}, \mathsf{nb}, E >$ corresponds to the new definition of f. ◀

This rule eliminates the square roots and divisions from the body of the function definition. This enables us to define the $Elim_{fout}$ function that respects the specification of Definition 4.11:

DEFINITION 4.13 ($Elim_{fout}$). The $Elim_{fout}$ function is the composition of 2 steps:

- Decompose the body according to the Definition 3.22 with respect of the conditions (Hp), (HT_1), (HT_2) and (Hf)
- Apply the rule of Definition 4.12

Given these two functions, $Elim_{fin}$ from Definition 4.10 and $Elim_{fout}$ from Definition 4.13, and the variable definition transformation defined in Definition 3.23 we define a strategy that removes square roots and divisions from all the definitions of a program.

4.3 DEPENDENCY GRAPH

Application of the rules relatives 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 neither the functions calls and definitions nor the variable definitions contain square roots or divisions. Indeed the transformation $Elim_{fin}$ 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:

EXAMPLE 4.11 (Transformation order). Given the following program:

$$\mathsf{letf\ f\ x} = \mathsf{x}/2;$$
$$\mathsf{let\ y} = \sqrt{a}\ \mathsf{in\ f(y)} > 0$$

By following the program order and first eliminating square roots and divisions in f and then in x we would get the following program:

$$\mathsf{letf\ f\ x} = (\mathsf{x},2);$$
$$\mathsf{let\ y} = a\ \mathsf{in\ let}\ (f_1, f_2) = \mathsf{f}(\sqrt{\mathsf{y}})\ \mathsf{in}\ f_1/f_2 > 0$$

The argument of a call of f still contains a square root.

This means that when we apply a transformation rule, we have to be sure that all the variables and functions it depends on have already been transformed. In order to be sure that we apply the rules in the right order we construct a dependency graph, that associates to every variable x, and to every function input f_i and function output f_o, the variables the corresponding transformation depends on. We call these elements, *i.e.*, the non functional variables and the function inputs and outputs, the *transformation items*.

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 $Elim_{fin}$ 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):

DEFINITION 4.14 (Unique Variable Definition). A program p has unique variable definition if every variable is only defined once. This means that, for every variable x the following set contains at most one element:

$$\begin{aligned}&\{\mathsf{let\ v = b\ in\ sc} \mid \mathsf{let\ v = b\ in\ sc} \lll \mathsf{p} \wedge \mathsf{x} \lll \mathsf{v}\} \cup \\ &\qquad \{\mathsf{letf\ f\ v = b;\ sc} \mid \mathsf{letf\ f\ v = b;\ sc} \lll \mathsf{p} \wedge \mathsf{x} \lll \mathsf{v}\} \cup \\ &\qquad\qquad \{\mathsf{letf\ x\ v = b;\ sc} \mid \mathsf{letf\ x\ v = b;\ sc} \lll \mathsf{p}\}\end{aligned}$$

And this set has to be empty if x is a free variable of p.

Every program can be easily transformed into a program with unique variable definition using variable renaming:

PROPOSITION 4.3. *By choosing* $\mathsf{x'}$ *and* $\mathsf{f'}$ *such that* $\neg\mathsf{x'} \ll (\mathsf{sc},\mathsf{b}) \wedge \neg\mathsf{f'} \ll (\mathsf{sc},\mathsf{b})$*, the following rules preserve the semantics:*

$$\begin{aligned}&\mathsf{let\ x = b\ in\ sc} &\longrightarrow\quad &\mathsf{let\ x' = b\ in\ sc[x \mapsto x']}\\ &\mathsf{letf\ f\ x = b;\ sc} &\longrightarrow\quad &\mathsf{letf\ f\ x' = b[x \mapsto x']; sc}\\ &\mathsf{letf\ f\ x = b;\ sc} &\longrightarrow\quad &\mathsf{let\ f'\ x = b;\ sc[f \mapsto f']}\end{aligned}$$

Given such a program with unique affectation we define the transformation item associated to every variable:

DEFINITION 4.15 (Corresponding transformation item). For every variable $\mathsf{x} \in \mathcal{X}$, we define $\mathsf{x}^\top$ the associated transformation item that corresponds to the only occurrence of its definition:

if $\{\mathsf{let\ v = b\ in\ sc} \mid \mathsf{let\ v = b\ in\ sc} \lll \mathsf{p} \wedge \mathsf{x} \lll \mathsf{v}\} = \{\mathsf{let\ v = b\ in\ sc}\}$ then $\mathsf{x}^\top = \mathsf{v}$
if $\{\mathsf{letf\ f\ v = b;\ sc} \mid \mathsf{letf\ f\ v = b;\ sc} \lll \mathsf{p} \wedge \mathsf{x} \lll \mathsf{v}\} = \{\mathsf{letf\ f\ v = b;\ sc}\}$ then $\mathsf{x}^\top = \mathsf{f}_i$
if $\{\mathsf{letf\ x\ v = b;\ sc} \mid \mathsf{letf\ x\ v = b;\ sc} \lll \mathsf{p}\} \neq \emptyset$ then $\mathsf{x}^\top = \mathsf{x}_o$

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. We compute the graph in a context that represents the transformation item that is depending on the current expression:

DEFINITION 4.16 (Definition Context). A context is a list of

- Variable definition context: $\mathcal{VD}(x)$ with $x \in \mathcal{V}$

- Function definition context: $\mathcal{FD}(f, x)$ with $(f, x) \in \mathcal{X} \times \mathcal{V}$

- Function application context: $\mathcal{FA}(f)$ with $f \in \mathcal{X}$

- Boolean context: $\mathcal{BC}$

In fact we are only interested in the items the corresponding $Elim_{let}$, $Elim_{fin}$ or $Elim_{fout}$ transformation depends on and these are the ones involved in the template computation. However, since the template of a Boolean expression is always a simple fresh variable, no square root or divisions appear in such a template and therefore the dependencies of a Boolean expression are empty. The Boolean context is used when we know that we are computing a Boolean value (*e.g.*, in the condition of an if then else).

We define the set of free variables of the numerical part of one expression in E. This set contains the variables that, if they are substituted, may change the form of the template:

DEFINITION 4.17 (Free variables of the numerical part). Given e an expression in E, in a program p, the set of free transformation items of the numerical part is defined by the following rules:

$$
\begin{aligned}
FV_{\mathbb{N}}(\mathsf{c}) &= \emptyset & FV_{\mathbb{N}}(\neg\mathsf{e1}) &= \emptyset \\
FV_{\mathbb{N}}((\mathsf{e1},\mathsf{e2})) &= FV_{\mathbb{N}}(\mathsf{e1}) \cup FV_{\mathbb{N}}(\mathsf{e2}) & FV_{\mathbb{N}}(\mathsf{uop\ e1}) &= FV_{\mathbb{N}}(\mathsf{e1}) \\
FV_{\mathbb{N}}(\mathsf{e1\ op\ e2}) &= \emptyset \quad \text{when } \mathsf{op} \in Cbop \cup Bbop & FV_{\mathbb{N}}(\mathsf{fst\ e1}) &= FV_{\mathbb{N}}(\mathsf{e1}) \\
FV_{\mathbb{N}}(\mathsf{e1\ op\ e2}) &= FV_{\mathbb{N}}(\mathsf{e1}) \cup FV_{\mathbb{N}}(\mathsf{e2}) & FV_{\mathbb{N}}(\mathsf{snd\ e1}) &= FV_{\mathbb{N}}(\mathsf{e1}) \\
FV_{\mathbb{N}}(\mathsf{f(p1)}) &= \mathsf{f}_o & FV_{\mathbb{N}}(\mathsf{x}) &= x^{\top}
\end{aligned}
$$

We now define the dependency graph. It associates to every transformation item the other items that are depending on it. The graph is represented as a set of edges, (a, b) meaning that b depends on a:

DEFINITION 4.18 (Dependency Graph). We denote the graph associated to the program $\mathsf{p} \in \mathsf{P}^{\top}$ in the context C by $(\mathsf{p}, C)^{\mathcal{G}}$:

$$
\begin{aligned}
(\mathsf{c}, C)^{\mathcal{G}} &= \emptyset & (\neg\mathsf{e1}, C)^{\mathcal{G}} &= \emptyset \\
((\mathsf{e1},\mathsf{e2}), C)^{\mathcal{G}} &= (\mathsf{e1}, C)^{\mathcal{G}} \cup (\mathsf{e2}, C)^{\mathcal{G}} & (\mathsf{uop\ e1}, C)^{\mathcal{G}} &= (\mathsf{e1}, C)^{\mathcal{G}} \\
(\mathsf{e1\ op\ e2}, C)^{\mathcal{G}} &= \emptyset \quad \text{when } \mathsf{op} \in Cbop \cup Bbop & (\mathsf{fst\ e1}, C)^{\mathcal{G}} &= (\mathsf{e1}, C)^{\mathcal{G}} \\
(\mathsf{e1\ op\ e2}, C)^{\mathcal{G}} &= (\mathsf{e1}, C)^{\mathcal{G}} \cup (\mathsf{e2}, C)^{\mathcal{G}} & (\mathsf{snd\ e1}, C)^{\mathcal{G}} &= (\mathsf{e1}, C)^{\mathcal{G}}
\end{aligned}
$$

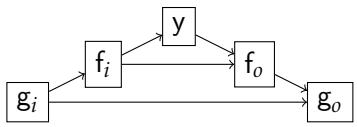
$$
\begin{aligned}
(\mathsf{if\ c\ then\ e1\ else\ e2}, C)^{\mathcal{G}} &= (\mathsf{c}, \mathcal{BC} :: C)^{\mathcal{G}} \cup (\mathsf{e1}, C)^{\mathcal{G}} \cup (\mathsf{e2}, C)^{\mathcal{G}} \\
(\mathsf{let\ v = e1\ in\ e2}, C)^{\mathcal{G}} &= (\mathsf{e1}, \mathcal{VD}(v) :: C)^{\mathcal{G}} \cup (\mathsf{e2}, C)^{\mathcal{G}} \\
(\mathsf{letf\ f\ x = e1\ in\ e2}, C)^{\mathcal{G}} &= \{(\mathsf{f}_i, \mathsf{f}_o)\}(\mathsf{e1}, \mathcal{FD}(f, x) :: C)^{\mathcal{G}} \cup (\mathsf{e2}, C)^{\mathcal{G}} \\
(\mathsf{f(e)}, C)^{\mathcal{G}} &= \emptyset \quad \text{when } C = \mathcal{BC} :: C' \text{ or } C = \emptyset \\
(\mathsf{f(e)}, \mathcal{VD}(x) :: C)^{\mathcal{G}} &= \{(\mathsf{f}_o, x)\} \cup (\mathsf{e}, \mathcal{FA}(f) :: \mathcal{VD}(x) :: C)^{\mathcal{G}} \\
(\mathsf{f(e)}, \mathcal{FD}(g, x) :: C)^{\mathcal{G}} &= \{(\mathsf{f}_o, g_o)\} \cup (\mathsf{e}, \mathcal{FA}(f) :: \mathcal{FD}(g, x) :: C)^{\mathcal{G}} \\
(\mathsf{f(e)}, \mathcal{FA}(g) :: C)^{\mathcal{G}} &= \{(\mathsf{f}_o, g_i)\} \cup (\mathsf{e}, \mathcal{FA}(f) :: \mathcal{FA}(g) :: C)^{\mathcal{G}} \\
(\mathsf{y}, C)^{\mathcal{G}} &= \emptyset \quad \text{when } C = \mathcal{BC} :: C' \text{ or } C = \emptyset \\
(\mathsf{y}, \mathcal{VD}(x) :: C)^{\mathcal{G}} &= \{(\mathsf{y}^{\top}, x)\} \\
(\mathsf{y}, \mathcal{FD}(g, x) :: C)^{\mathcal{G}} &= \{(\mathsf{y}^{\top}, g_o)\} \\
(\mathsf{y}, \mathcal{FA}(g) :: C)^{\mathcal{G}} &= \{(\mathsf{y}^{\top}, g_i)\}
\end{aligned}
$$

If $\mathsf{y}^{\top}$ does not exists (y is a free variable of the program), then there is no dependency.

In the following example we give the corresponding dependency graph

EXAMPLE 4.12 (Dependency graph).

$\mathsf{letf\ f\ x = let\ y} = \sqrt{\mathsf{x}} + a \ \mathsf{in}\ (\mathsf{y} + \mathsf{x})/b;$
$\mathsf{letf\ g\ z = f(z)}/e;$
$(\mathsf{f}(c + \sqrt{b}) + \mathsf{g}(d))$



The dependencies computed by this graph represents the variable that are involved in the template computation in the input program:

PROPOSITION 4.4 (Dependency graph characterization). *Given a program* p *and its dependency graph* $\mathsf{p}^{\mathcal{G}}$ *we have the following properties:*

$$
\begin{aligned}
&\forall\, \mathsf{x,\ b,\ sc,\ let\ x = b\ in\ sc} \ll \mathsf{p} \wedge \mathsf{b} = (Pp(e_1, ..., e_n)) \Longrightarrow \\
&\qquad \forall\, y \notin FV(\mathsf{p}),\ \exists i,\ y \in FV_{\mathbb{N}}(e_i) \Leftrightarrow (y^{\top}, \mathsf{x}) \in \mathsf{p}^{\mathcal{G}} \\
&\forall\, \mathsf{f,\ x,\ b,\ sc,\ letf\ f\ x = b;\ sc} \ll \mathsf{p} \wedge \mathsf{b} = (Pp(e_1, ..., e_n)) \Longrightarrow \\
&\qquad \forall\, y \notin FV(\mathsf{p}),\ \exists i,\ y \in FV_{\mathbb{N}}(e_i) \Leftrightarrow (y^{\top}, \mathsf{f}_o) \in \mathsf{p}^{\mathcal{G}} \\
&\forall\, \mathsf{f,\ x,\ b,\ sc,\ letf\ f\ x = b;\ sc} \ll \mathsf{p} \wedge \mathsf{sc} = scf(Pp_1(e_{11}, ..., e_{1m_1}), ..., Pp_n(e_{n1}, ..., e_{nm_n})) \Longrightarrow \\
&\qquad \forall\, y \notin FV(\mathsf{p}),\ \exists i, j,\ y \in FV_{\mathbb{N}}(e_{ij}) \Leftrightarrow (y^{\top}, \mathsf{f}_i) \in \mathsf{p}^{\mathcal{G}}
\end{aligned}
$$

We define an extension of the $FV_{\mathbb{N}}$ notation. For any transformation item, in a program p, we denote $FV_{\mathbb{N}}(x, p)$ the variables that can appear in the template computation.

DEFINITION 4.19 (Variable template dependencies). We call the template dependencies of a transformation item the transformation items corresponding to the free variable of the numerical sub-expressions that have to be anti-unified:

$$
\begin{aligned}
&FV_{\mathbb{N}}(\mathsf{x}, \mathsf{p}) = \cup_i FV_{\mathbb{N}}(e_i) \\
&\qquad \text{when } \mathsf{let\ x} = (Pp(e_1, ..., e_n)) \ \mathsf{in\ sc} \ll \mathsf{p} \\
&FV_{\mathbb{N}}(\mathsf{f}_o, \mathsf{p}) = \cup_i FV_{\mathbb{N}}(e_i) \\
&\qquad \text{when } \mathsf{letf\ f\ x} = (Pp(e_1, ..., e_n));\ \mathsf{sc} \ll \mathsf{p}
\end{aligned}
$$

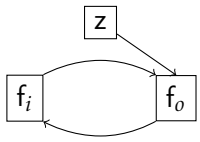
$$FV_{\mathbb{N}}(\mathsf{f}_i, \mathsf{p}) = \cup_{ij} FV_{\mathbb{N}}(e_{ij})$$
$$\text{when letf f x =b;}scf(\mathsf{f}(Pp_1(e_{11}, ..., e_{1m_1}), ..., \mathsf{f}(Pp_n(e_{n1}, ..., e_{nm_n})))) \leqslant \mathsf{p}$$

We dissociate the dependency graph and the $FV_{\mathbb{N}}$ since the dependency graph will be only computed once at the beginning of the transformation. However we will see that this graph contains enough information to track the evolution of the $FV_{\mathbb{N}}$ during the transformation.

Firstly, notice that, in some cases, the dependency graph can contain cycles, one of the most simple case with cycle is when a function is applied to itself:

EXAMPLE 4.13 (Cyclic dependency graph).

letf f x = let z = e in x $+\sqrt{\mathsf{z}}$;
f(f(s))



A cycle means that both templates of the input and of the output depend on each other. We are not able to transform program whose dependency graph contains cycle at this moment. As seen in Example 4.13, since for every function the edge (f_i, f_o) is in the graph and a nested call (*e.g.*, $f(f(x))$) introduces an edge or a path between f_o and f_i, then having acyclic graphs prevents nested calls of functions and thus the $Elim_{fin}$ rules can always be applied.

We will discuss in Section 4.5 prospectives to extend the transformation to handle these cycles. However, there are lots of programs whose dependency graph is acyclic, or programs that can easily be transformed into a program that has an acyclic dependency graph. These programs with acyclic graphs can be transformed with the algorithm we present in the following section. Given an acyclic dependency graph, we now present how it is used to define the order to transform the variable and function definitions.

4.4 ORDER FOR VARIABLE DEFINITION TRANSFORMATION

We want to find a right order for the transformation of the variable and function definitions. The main goal this order has to achieve is that when a variable or a function input or output has been transformed by removing square roots and divisions from the body of the definition or the call, then the transformations that will be applied after will not introduce new square roots or divisions in the definitions or function arguments where square roots and divisions have already been eliminated.

An edge of the dependency graph (x, y) represents the fact that the inlining of the template by applying one of the rules $Elim_{let}$, $Elim_{fin}$ or $Elim_{fout}$ to the definition of x, might introduce square roots or divisions in numerical part of the definition of y. This means that, in the input program, the inlining relative to these rule can only modify the direct dependencies of x. However, the template only contains variables that x was depending on, thus, the inlining of this template might make y to depend on the variables x was depending on. Therefore by doing repeated inlining, we aim at proving that the variables that may introduce square roots or divisions in a variable y are the z such that there is a path from y to z inside the graph of the input program. Since the transformed

definitions have the same dependencies than the original ones, if we transform the variables by following the graph, then once a variable definition has been transformed, the following transformations will not introduce any square roots or divisions in the parts these operations have been eliminated from.

Therefore in a first step we present how a definition transformation might modify the other definitions and then how we can arrange these transformation to avoid the creation of square roots and divisions in already transformed definitions.

4.4.1 Variable inlining consequences

In order to formally state the evolution of the transformed program, we define the relation corresponding to the transformation of a variable or function input or output in a program.

DEFINITION 4.20 (Variable transformation relation). The application of one of the rules $Elim_{let}$, $Elim_{fin}$ or $Elim_{fout}$ in a program is represented by the following relation:

$$\begin{aligned}
&\mathcal{E}lim_{(x,T)}(\mathsf{p1},\mathsf{p2}) = \\
&\qquad \mathsf{p2} = \mathsf{p1}[\mathsf{let}\ x = \mathsf{b}\ \mathsf{in}\ \mathsf{sc} \mapsto Elim_{let}(\mathsf{let}\ x = \mathsf{b}\ \mathsf{in}\ \mathsf{sc})] \vee \\
&\qquad x = f_i \wedge \mathsf{p2} = \mathsf{p1}[\mathsf{letf}\ f\ \mathsf{z} = \mathsf{b};\ \mathsf{sc} \mapsto Elim_{fin}(\mathsf{letf}\ f\ \mathsf{z} = \mathsf{b};\ \mathsf{sc})] \vee \\
&\qquad x = f_o \wedge \mathsf{p2} = \mathsf{p1}[\mathsf{letf}\ f\ \mathsf{z} = \mathsf{b};\ \mathsf{sc} \mapsto Pnorm(Elim_{fout}(\mathsf{letf}\ f\ \mathsf{z} = \mathsf{b};\ \mathsf{sc})))]
\end{aligned}$$

Where we use T as the template for the chosen $Elim_x$ rule and $Pnorm$ is the normalization algorithm from Definition 3.17 (we denote by $Elim_x$ the one of the rules $Elim_{let}$, $Elim_{fin}$ or $Elim_{fout}$ corresponding to the transformation item x). The $Pnorm$ reduction is used because the $Elim_{fout}$ rule introduces variable definitions inside expressions. Thus we need to extract these new variable definitions from these expressions before continuing the transformation in order to still have a program in normal form P.

These transformation rules might modify the dependencies of the variables, indeed these rules inline the template in the body or the scope of the definition. However we can easily track the modified definitions since they are the ones that were depending on x.

PROPOSITION 4.5 ($\mathcal{E}lim$ and transformation dependencies). *We can define the modification in the transformation dependencies sets*

$$\begin{aligned}
&\forall \mathsf{p1},\ \mathsf{p2},\ \mathsf{x},\ T, \mathcal{E}lim_{(\mathsf{x},T)}(\mathsf{p1},\mathsf{p2}) \Longrightarrow \\
&\quad \forall\ \mathsf{v} \leqslant \mathsf{p1}, \mathsf{v} \neq \mathsf{x}, \\
&\qquad \mathsf{x} \notin FV_{\mathbb{N}}(\mathsf{v},\mathsf{p1}) \Longrightarrow FV_{\mathbb{N}}(\mathsf{v},\mathsf{p2}) = FV_{\mathbb{N}}(\mathsf{v},\mathsf{p1}) \wedge \\
&\qquad \mathsf{x} \in FV_{\mathbb{N}}(\mathsf{v},\mathsf{p1}) \Longrightarrow FV_{\mathbb{N}}(\mathsf{v},\mathsf{p2}) \subseteq (FV_{\mathbb{N}}(\mathsf{v},\mathsf{p1}) \backslash \{\mathsf{x}\}) \cup FV_{\mathbb{N}}(T) \cup \mathcal{F}_{\mathsf{x}}
\end{aligned}$$

Where $\mathcal{F}_{\mathsf{x}}$ are new fresh variables that may be introduced by the Pnorm reduction after we use the $Elim_{fout}$ rule. Therefore $\mathcal{F}_{f_i} = \mathcal{F}_x = \emptyset$ and $\forall \mathsf{z} \in \mathcal{F}_{f_o}$, $FV_{\mathbb{N}}(\mathsf{z},\mathsf{p2}) = \{f_o\}$ (z is defined by let z = $f(e_i)$ in sc*).*

Proof. By using rule $Elim_{let}$ then x is replaced by its template, by using $Elim_{fin}$ the bound variables are replaced by the template, thus the variable in the dependency set after inlining are the one before the inlining plus the one of the template.

The rule $Elim_{fout}$ replaces f(e) by let x' = f(e) in T and then after the reduction that normalizes the program, the variables in the corresponding expressions are the original

ones plus the ones in T plus the fresh ones that have replaced the substitution variables in T using the switch rules from Definition 3.17. ◀

We can also characterize the free variables of a template. Indeed in the template computation introduced in Definition 3.26, the variable appearing in the template are either the ones of the substitution or the ones in terms that are anti-unified using equality (using the (EI) rule).

PROPOSITION 4.6 (Template variables). *Given a set of expressions $e_1, ..., e_n$, the free variables of a template T with the associated substitutions $\sigma_1, ..., \sigma_n$ on the same domain computed as introduced in Definition 3.26 are either the ones chosen for the substitution or free variables appearing in at least one of the e_i expressions:*

$$\forall\, x \ll T,\ x \in \cup_i FV_{\mathbb{N}}(e_i) \cup \mathcal{D}(\sigma_1)$$

The variable in the numerical part of the images of the substitution are also in the input expressions:

$$\forall\, j,\ FV_{\mathbb{N}}(\mathcal{I}(\sigma_j)) \subseteq \cup_i FV_{\mathbb{N}}(e_i)$$

Proof. The only terminating cases for the template computation are:

- the (V) rule, that produces a variable in the substitution, and the corresponding expression in the image of the substitution
- the (EI) rule, that is applied when all the corresponding square roots are equal. Therefore their free variables are in the numerical part of at least one the input expressions (it is at least and not all since the replacement with elements of the $\mathcal{P}os$ set can introduce variables from the other expressions).

◀

Thus the variables of the template used in the transformation of x in p1 only depend on the ones in $FV_{\mathbb{N}}(\mathsf{x},\mathsf{p1})$:

PROPOSITION 4.7 (Template variables dependencies).

$\forall \mathsf{p1},\ \mathsf{p2},\ \mathsf{x},\ T, \mathcal{E}lim_{(\mathsf{x},T)}(\mathsf{p1},\mathsf{p2}) \Longrightarrow$
$\quad FV_{\mathbb{N}}(T) \subseteq FV_{\mathbb{N}}(\mathsf{x},\mathsf{p1}) \cup \{\mathsf{x'}\}$

Where x' *is the multi-variable corresponding to the substitution.*
And then we have $FV_{\mathbb{N}}(\mathsf{x'}^{\top},\mathsf{p2}) \subseteq FV_{\mathbb{N}}(\mathsf{x},\mathsf{p1})$ *or* $FV_{\mathbb{N}}(\mathsf{x'}^{\top},\mathsf{p2}) = \{\mathsf{f}_o\}$ *and* $FV_{\mathbb{N}}(\mathsf{f}_o,\mathsf{p2}) \subseteq FV_{\mathbb{N}}(\mathsf{f}_o,\mathsf{p1})$ *when* $\mathsf{x} = \mathsf{f}_o$

Proof. Using Proposition 4.6, the variables of the input expressions are in $FV_{\mathbb{N}}(\mathsf{x},\mathsf{p1})$.
If a variable is in the substitution then the new definition in p2 is either:

- let $\mathsf{x'} = Pp(arg(\sigma_1), ..., arg(\sigma_n))$ in $\mathsf{sc}[\mathsf{x} \mapsto T]]$
  and thus $FV_{\mathbb{N}}(\mathsf{x'},\mathsf{p2}) \subseteq FV_{\mathbb{N}}(\mathsf{x},\mathsf{p1})$
  since $FV_{\mathbb{N}}(arg(\sigma_1), ..., arg(\sigma_n)) \subseteq FV_{\mathbb{N}}(e_1, ..., e_n)$

- letf f x' = b[x$\mapsto T$]; $scf(Pp_1(arg(\sigma_{11}),...,arg(\sigma_{nm_n})))$
  and thus $FV_{\mathbb{N}}$(x'$^{\top}$,p2) $\subseteq FV_{\mathbb{N}}$(f$_i$,p1)
  since $FV_{\mathbb{N}}(arg(\sigma_{11}),...,arg(\sigma_{nm_n})) \subseteq FV_{\mathbb{N}}(e_{11},...,e_{nm_n})$

- letf f x = $Pp(arg(\sigma_1),...,arg(\sigma_n))$;
  scf(let x$_1$' = f(a$_1$) in T[x' $\mapsto$ x$_1$'],...,let x$_m$' = f(a$_m$) in T[x' $\mapsto$ x$_m$'])
  and thus $FV_{\mathbb{N}}$(x$_k$',p2) = {f$_o$} and $FV_{\mathbb{N}}$(f$_o$,p2) $\subseteq FV_{\mathbb{N}}$(f$_o$,p1)

◀

The set of the transformation dependencies ($FV_{\mathbb{N}}$) is the set of variables such that, if they are inlined, they may introduce square roots or divisions in the numerical part of the body:

PROPOSITION 4.8 (Free variable of numerical expression dependency). *Only the elimination rules corresponding to a dependency can change the subtype of a definition:*

$$\forall \mathsf{p1},\mathsf{p2},\ \mathcal{E}lim_{(x,T)}(\mathsf{p1},\mathsf{p2}) \Longrightarrow$$
$$\forall \mathsf{z},\mathsf{b},\mathsf{b'},\mathsf{sc},\mathsf{sc},$$
$$\mathsf{let\ z = b\ in\ sc} \lll \mathsf{p1} \wedge \mathsf{b} \in \mathsf{P}_{B_{\surd,/}} \wedge \mathsf{let\ z = b'\ in\ sc'} \lll \mathsf{p2} \wedge \mathsf{b'} \notin \mathsf{P}_{B_{\surd,/}} \Longrightarrow$$
$$x \in FV_{\mathbb{N}}(\mathsf{z},\mathsf{p1}) \wedge$$
$$\forall f,$$
$$(\forall \mathsf{e}, f(\mathsf{e}) \lll \mathsf{p1} \Rightarrow \mathsf{e} \in \mathsf{P}_{B_{\surd,/}}) \wedge \exists \mathsf{e}, f(\mathsf{e}) \lll \mathsf{p2} \wedge \mathsf{e} \notin \mathsf{P}_{B_{\surd,/}} \Longrightarrow$$
$$x \in FV_{\mathbb{N}}(f_i,\mathsf{p1}) \wedge$$
$$\forall f,z,\mathsf{b},\mathsf{b'},\mathsf{sc},\mathsf{sc},$$
$$\mathsf{letf}\ f\ z = \mathsf{b;\ sc} \lll \mathsf{p1} \wedge \mathsf{b} \in \mathsf{P}_{B_{\surd,/}} \wedge \mathsf{let}\ f\ z' = \mathsf{b';\ sc'} \lll \mathsf{p2} \wedge \mathsf{b'} \notin \mathsf{P}_{B_{\surd,/}} \Longrightarrow$$
$$x \in FV_{\mathbb{N}}(f_o,\mathsf{p1})$$

Proof. The $FV_{\mathbb{N}}$(z,p) is the set of the free variables appearing in the definition of z (or function calls) in p. Thus if we introduce a square root or a division in such definition (or call) by using one of the *Elim* rules, then this square root comes from the template inlining of x and the inlining only modify the parts that contains x as a free variable. Since square roots and divisions introduction in Boolean expressions is not a concern for the $\mathsf{P}_{B_{\surd,/}}$ subtype, the elimination rule can only change the type of definition or call that have x as a free variable in the numerical part of the definition (or call). ◀

We now know what variables might modify the subtype of a definition, thus we present how iteration of such transformations can be arranged in order to avoid such modifications after the transformations.

4.4.2 Definition transformation iteration

We only aim at transforming definitions that are in the input program, hence we are only interested in tracking in the transformation dependencies the transformation items that are in the input program. We first define the set of items corresponding to a program:

DEFINITION 4.21 (Transformation items of a program $IT(\mathsf{p})$). These are the items corresponding to all the definitions of the program

$$\mathsf{x} \in IT(\mathsf{p}) \text{ when } \exists\, \mathsf{b},\ \mathsf{sc},\ \mathsf{let\ x = b\ in\ sc} \lll \mathsf{p}$$
$$\{\mathsf{f}_i,\mathsf{f}_o\} \subseteq IT(\mathsf{p}) \text{ when } \exists\, \mathsf{x},\ \mathsf{b},\ \mathsf{sc},\ \mathsf{letf\ f\ x = b;\ sc} \lll \mathsf{p}$$

Let us now consider the iteration of the $Elim_x$ transformations, we define the transitive closure of the $\mathcal{E}lim$ relation:

DEFINITION 4.22 (Transformation iteration).

$$\mathcal{E}lim_{[\,]}(\mathsf{p1},\mathsf{p2}) \iff \mathsf{p1} = \mathsf{p2}$$
$$\mathcal{E}lim_{(x,T)::l}(\mathsf{p1},\mathsf{p2}) \iff \exists \mathsf{p}, \mathcal{E}lim_l(\mathsf{p1},\mathsf{p}) \wedge \mathcal{E}lim_{(x,T)}(\mathsf{p},\mathsf{p2})$$

We only want to transform definitions and calls that are in the input program, thus we want to see, at any point of the transformation, which $FV_{\mathbb{N}}(z,p)$ contain transformation items that are going to be transformed. This is why we defined the dependency graph, since the transformation items that may end up in $FV_{\mathbb{N}}(z,p)$ are the ones that have a path to z. We define as usual a path in such a graph:

DEFINITION 4.23 (Path). We use the usual definition of a path, we say that there is a path from x to z in G, denoted $(x,z)^* \in G$ if

$$\exists\, y_1,...,y_n,\ (x,y_1) \in G\ \wedge\ \forall\, i,\ (y_i,y_{i+1}) \in G\ \wedge (y_n,z) \in G$$

Then we can state the property for the transformation items in the program transformation process:

PROPOSITION 4.9 (Path and original item dependencies). *Given a program* p *and its dependency graph* $\mathsf{p}^{\mathcal{G}}$*, the dependencies of a transformation item after a sequence of transformation are included its predecessors in the initial graph.*

$$\forall \mathsf{p'},\ l, \mathcal{E}lim_l(\mathsf{p},\mathsf{p'}) \wedge \forall\, x \in l, x \in IT(\mathsf{p}) \implies$$
$$\forall ti \in IT(\mathsf{p}), FV_{\mathbb{N}}(ti,\mathsf{p'}) \cap IT(\mathsf{p}) \subseteq \{it \in IT(\mathsf{p}) \mid (it,ti)^* \in \mathsf{p}^{\mathcal{G}}\}$$

Proof. By induction on l, Proposition 4.4 gives the empty list case. Then the induction case: assume we have $\mathsf{p1}$ such that $\mathcal{E}lim_l(\mathsf{p},\mathsf{p1}) \wedge \mathcal{E}lim_{(x,T)}(\mathsf{p1},\mathsf{p'})$, then using Proposition 4.5, we have 2 cases:

- $x \notin FV_{\mathbb{N}}(ti,\mathsf{p1})$ then the induction steps trivially applies
- $x \in FV_{\mathbb{N}}(ti,\mathsf{p1})$ then $FV_{\mathbb{N}}(ti,\mathsf{p'}) = (FV_{\mathbb{N}}(\mathsf{ti},\mathsf{p1}) \backslash \{x\}) \cup FV_{\mathbb{N}}(T) \cup \mathcal{F}_x$
  We have
  $\mathcal{F}_x \cap IT(\mathsf{p}) = \emptyset$ since they are new fresh variables and
  $(FV_{\mathbb{N}}(ti,\mathsf{p1}) \backslash \{x\}) \subseteq \{it \in IT(\mathsf{p}) \mid (it,ti)^* \in \mathsf{p}^{\mathcal{G}}\}$ by induction.
  Then using Proposition 4.7 we have:
  $FV_{\mathbb{N}}(T) \subseteq FV_{\mathbb{N}}(x,\mathsf{p1}) \cup \{x'\}$:
  Since $x \in IT(\mathsf{p})$, using the induction with x, we have:
  $FV_{\mathbb{N}}(x,\mathsf{p1}) \cap IT(\mathsf{p}) \subseteq \{it \in IT(\mathsf{p}) \mid (it,x)^* \in \mathsf{p}^{\mathcal{G}}\}$
  and since $x \in FV_{\mathbb{N}}(ti,\mathsf{p1})$ using induction with ti then:

$(x, ti)^* \in \mathsf{p}^{\mathcal{G}}$.

Thus $FV_{\mathbb{N}}(x, \mathsf{p1}) \cap IT(\mathsf{p}) \subseteq \{it \in IT(\mathsf{p}) \mid (it, ti)^* \in \mathsf{p}^{\mathcal{G}}\}$

Finally, x' is either equal to x (when the definition does not change) and thus in $FV_{\mathbb{N}}(ti, \mathsf{p1})$ or a fresh variable used for the template computation thus not in $IT(\mathsf{p})$.

◀

This proposition only characterizes the set of variables involved in the template computation of an item of the input program. We now describe the set of variables that may modify an already transformed definition where variables have been renamed, first let us define the first occurrence of this new defined variable:

PROPOSITION 4.10 (Variable transformation introduction). *Given a program* p*, if a definition is in the transformed program and not in* p*, then there is one program that have introduced it.*

$$\forall\, \mathsf{p'},\ l, \mathcal{E}lim_l(\mathsf{p}, \mathsf{p'}) \wedge \forall y \in l, y \in IT(\mathsf{p}) \Longrightarrow$$
$$\forall ti \in IT(\mathsf{p'}) \backslash IT(\mathsf{p}), \exists\, x, T, l', \mathsf{p1}, \mathsf{p2}, \mathcal{E}lim_l(\mathsf{p}, \mathsf{p1}) \wedge \mathcal{E}lim_{(x,T)}(\mathsf{p1}, \mathsf{p2})$$
$$x :: l' \ll l \wedge ti \notin IT(\mathsf{p1}) \wedge ti \in IT(\mathsf{p2})$$

Where the sub-term relation on list $l' \ll l$ means that there exists $y_1, ..., y_n$ such that $y_1 :: ... :: y_n :: l' = l$. We denote the elements verifying this property by $x_{intro}(it)$, $l'_{intro}(it)$, $\mathsf{p1}_{intro}(it)$, $\mathsf{p2}_{intro}(it)$ and $T_{intro}(it)$.

Proof. Trivially, there is one program which is the first to introduce this new definition and no definition is deleted except the ones in p. ◀

We can state that, if the order for transformation is correct, then once introduced, the new variable only depends on variables that have already been transformed.

PROPOSITION 4.11 (Dependencies of transformed definition). *Given a program* p*, its dependency graph* $\mathsf{p}^{\mathcal{G}}$ *the dependency of the new definitions introduced by the transformation are the ones of the input item:*

$$\forall \mathsf{p'},\ \mathsf{p''},\ l,\ x \in IT(\mathsf{p}), \mathcal{E}lim_l(\mathsf{p}, \mathsf{p'}) \wedge \mathcal{E}lim_{(x,T)}(\mathsf{p'}, \mathsf{p''})\ \wedge\ \forall y \in l, y \in IT(\mathsf{p}) \Longrightarrow$$
$$\forall ti \in IT(\mathsf{p''}) \backslash IT(\mathsf{p'}),\ FV_{\mathbb{N}}(ti, \mathsf{p''}) \cap IT(\mathsf{p}) \subseteq \{y \mid (y, x)^* \in \mathsf{p}^{\mathcal{G}}\} \cup \{x\}$$

Proof. The ti definition is introduced by the $Elim_x$ rule, then depending on the rule:

- if x is a non functional variable, then we have the following relation:

  $\mathsf{p''} = \mathsf{p'}[\mathsf{let}\ x = \mathsf{b}\ \mathsf{in}\ \mathsf{sc} \mapsto (\mathsf{let}\ x' = \mathsf{nb}\ \mathsf{in}\ \mathsf{sc}[\mathsf{x} \mapsto T])]$

  where the only new item is x' and its dependencies are the dependencies of nb, that are included in the dependencies of b (the expressions in the substitution are subterms of the input expression for the template computation). The dependencies of b are the dependencies of x which is in $IT(\mathsf{p})$ and thus Proposition 4.9 gives us the property we want.

- if $x = \mathsf{f}_i$ then

  $\mathsf{p2} = \mathsf{p1}[\mathsf{letf}\ f\ \mathsf{z} = \mathsf{b};\ \mathsf{sc} \mapsto \mathsf{letf}\ f\ \mathsf{z'} = \mathsf{b}[\mathsf{z} \mapsto T];\ \mathsf{sc}[\mathsf{f}(\mathsf{e}_1) \mapsto \mathsf{f}(\mathsf{e'}_1),\ldots,\mathsf{f}(\mathsf{e}_n) \mapsto \mathsf{f}(\mathsf{e'}_n)]]$

  There is no new variable definition

- if $x = \mathsf{f}_o$ then

  $\mathsf{p2} = Pnorm(\mathsf{p1}[\mathsf{letf}\ f\ \mathsf{z} = \mathsf{b};\ \mathsf{sc} \mapsto \mathsf{letf}\ f\ \mathsf{z} = \mathsf{nb};$
  $\mathsf{sc}[\mathsf{f}(\mathsf{e}_1) \mapsto \mathsf{let}\ \mathsf{y}_1 = \mathsf{f}(\mathsf{e}_1)\ \mathsf{in}\ T,\ldots,\mathsf{f}(\mathsf{e}_n) \mapsto \mathsf{let}\ \mathsf{y}_n = \mathsf{f}(\mathsf{e}_n)\ \mathsf{in}\ T]])$

  The new definitions, $\mathsf{y}_1,\ldots,\mathsf{y}_n$ are defined by $\mathsf{f}(.)$ thus their only dependency is $\mathsf{f}_o = x$ and the normalization process does not change this definition but only the names of the y_i

◀

Thus by choosing the right order we can ensure that the set of dependencies of a transformed variable does not change later in the transformation process:

PROPOSITION 4.12 (Invariant dependencies of created variables).

$$\forall \mathsf{p'},\ l, \mathcal{E}lim_l(\mathsf{p}, \mathsf{p'}) \wedge \forall\, x \in l, x \in IT(\mathsf{p}) \Longrightarrow$$
$$\forall\ (x :: ls) \lll l, \{y \mid (y, x)^* \in \mathsf{p}^{\mathcal{G}}\} \subseteq ls \wedge x \notin ls \Longrightarrow$$
$$\forall\, ti \in IT(\mathsf{p'}) \backslash IT(\mathsf{p}),$$
$$FV_{\mathbb{N}}(ti, \mathsf{p2}_{intro}) \cap IT(\mathsf{p}) = FV_{\mathbb{N}}(ti, \mathsf{p'}) \cap IT(\mathsf{p})$$

Proof. There exists a list l' that represents the transformation that happened after the one that created ti: $l = l'@(x_{intro}(ti), T_{intro}(ti)) :: l_{intro}(ti)$ where @ is the list concatenation operation. Now depending on $x_{intro}(ti)$:

- if x_{intro} is a non functional variable, then using Proposition 4.11 we have:

  $$FV_{\mathbb{N}}(ti, \mathsf{p2}_{intro}) \cap IT(\mathsf{p}) \subseteq \{it \in IT(\mathsf{p}) \mid (it, x)^* \in \mathsf{p}^{\mathcal{G}}\}$$

  Thus using the hypothesis of the list, we know that any y in $\{it \in IT(\mathsf{p}) \mid (it, x)^* \in \mathsf{p}^{\mathcal{G}}\}$ is in l_{intro} and thus not in l' (x can not be twice in l). By using Proposition 4.5, we get that the $FV_{\mathbb{N}}(ti, \mathsf{p2}_{intro})$ set does not change due to the transformation in l'

- if $x_{intro} = f_i$ there is no new definition

- if $x_{intro} = f_o$, $FV_{\mathbb{N}}(ti, \mathsf{p2}_{intro}) = f_o$ and since $x_{intro} \notin l'$, using Proposition 4.5, the set does not change due to the elimination rules applied in l'.

◀

We can specify the elements in the $FV_{\mathbb{N}}$ for the transformation items. Thus we can state the property that the set of variables that remain to be transformed decreases. We define the set of transformations that remain to be completed in order to remove square roots and divisions from definitions or function calls:

DEFINITION 4.24 (Set of transformation items with square roots or divisions). We define the set of the variable definitions and function calls that still contain square roots or divisions in their numerical parts $\mathcal{D}(\mathsf{p})$:

- $\mathsf{x} \in \mathcal{D}(\mathsf{p})$ when $\mathsf{let\ x = b\ in\ sc} \lll \mathsf{p} \wedge \mathsf{b} \notin \mathsf{P}_{\mathsf{B}_{\surd,/}}$
- $\mathsf{f}_i \in \mathcal{D}(\mathsf{p})$ when $\exists\, \mathsf{e} \notin \mathsf{P}_{\mathsf{B}_{\surd,/}}$, $\mathsf{f(e)} \lll \mathsf{p}$
- $\mathsf{f}_o \in \mathcal{D}(\mathsf{p})$ when $\mathsf{let\ f\ x = b;\ sc} \lll \mathsf{p} \wedge \mathsf{b} \notin \mathsf{P}_{\mathsf{B}_{\surd,/}}$

Thus given a transformation order that respect the dependencies, we can state that we can reduce this set to the empty set, since it is always included in the list of transformations that are left to be completed.

PROPOSITION 4.13 (Transformation order). *Given a program* p *and its dependency graph, the transformations that remain to be done are the ones of the input program that have not been transformed yet:*

$$\forall \mathsf{p}',\ l, \mathcal{E}lim_l(\mathsf{p},\mathsf{p}') \wedge \forall\, x \in l, x \in IT(\mathsf{p}) \Longrightarrow$$
$$\forall\ (x :: l') \lll l, \{y | (y,x)^* \in \mathsf{p}^{\mathcal{G}}\} \subseteq l' \wedge x \notin l' \Longrightarrow \mathcal{D}(\mathsf{p}') \subseteq IT(\mathsf{p}) \backslash l$$

Proof. By induction on l, initiation case is trivial with $l = []$, then the induction case:
Assume the list is $x :: l$ and p'' such that $\mathcal{E}lim_l(\mathsf{p},\mathsf{p}'') \wedge \mathcal{E}lim_x(\mathsf{p}'',\mathsf{p}')$ and $it \in \mathcal{D}(\mathsf{p}')$ then:
If $it \in \mathcal{D}(\mathsf{p}'')$ then the induction hypothesis terminates
If $it \notin \mathcal{D}(\mathsf{p})$ then, using Proposition 4.8, we have $x \in FV_{\mathbb{N}}(it, p'')$. Then depending on it:
If $it \notin IT(\mathsf{p})$ then using propositions 4.12 and 4.11, we have that:
$FV_{\mathbb{N}}(it, p'') \subseteq \{y \mid (y, x_{init}(it))^* \in \mathsf{p}^{\mathcal{G}}\} \cup \{x_{init}(it)\}$
and thus $(x, x_{init}(it))^* \in \mathsf{p}^{\mathcal{G}}$ which contradicts the hypothesis of the list, we have
$x_{init} :: l_{init} \ll l \wedge x \notin l$.
If $it \in IT(\mathsf{p})$ then using propositions 4.9 we have that:
$x \in \{y \mid (y, it)^* \in \mathsf{p}^{\mathcal{G}}\}$
and then if it is in l then it contradicts the hypothesis on the list, therefore it has not been transformed yet, $it \in IT(\mathsf{p}) \backslash l$ ◀

Therefore, once every element in $IT(p)$ has been transformed, there is no more definition or function call that contains any square root or division. The only thing to do is to define this order of transformation that respect the hypothesis on the list, that is we only apply a transformation after all the transformations it depends on have been done. We define the following algorithm that transforms a program into an equivalent one where all the variable definitions and function calls are in $\mathsf{P}_{\mathsf{B}_{\surd,/}}$:

ALGORITHM 4.25 (Variable definition transformation $Elim_{\mathcal{V}ar}$). Given a program p we define the $Elim_{\mathcal{V}ar}$ function that transforms all the variable definitions and function calls:

i) $\mathcal{G} := \mathsf{p}^{\mathcal{G}}$, check that $\mathcal{G}$ is acyclic

ii) while $\mathcal{G}$ is not empty do

- Choose a root x
- Transform the definition of x using the according $Elim_{let}$, $Elim_{fin}$ or $Elim_{fout}$ transformation
- Re-normalize the program with *Pnorm* from Definition 3.17 if the $Elim_{fout}$ rule was used
- $\mathcal{G} := \{(y, z) \in \mathcal{G} \mid y \neq x\}$

iii) Transforms the variables that were not in $\mathsf{p}^{\mathcal{G}}$, *e.g.*, variables that are only used in Boolean expressions $\mathsf{let\ x} = \sqrt{2}\ \mathsf{in\ x} > 1.4$

PROPOSITION 4.14 ($Elim_{\mathcal{V}ar}$ correctness). *The $Elim_{\mathcal{V}ar}$ algorithm preserves the semantics and produces a program where all the variable definition are in* $\mathsf{P}_{\mathsf{B}_{\sqrt{,/}}}$.

Proof. Preservation of the semantics is ensured by the preservation induced by any of the $Elim_{let}$, $Elim_{fin}$ or $Elim_{fout}$ and *Pnorm* rules. Since we only transform roots, the hypothesis of Proposition 4.13 is verified, one node becoming 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, the $\mathcal{D}$ set is empty at the end, and therefore the variable definitions and functions calls are in $\mathsf{P}_{\mathsf{B}_{\sqrt{,/}}}$. ◀

Once all the variable definitions have been transformed, the only step left is to eliminate the square roots and divisions that are left in the Boolean expressions. This is done using the following algorithm that applies the $Elim_{\mathbb{B}}$ transformation to any sub-term that is a Boolean expression (whose head operators is either a Boolean or a comparison operator):

ALGORITHM 4.26 (Global elimination in Boolean expression). Using the $Elim_{\mathbb{B}}$ algorithm introduced in Definition 3.19, we define the global algorithm $Elim_{\mathbb{B}}^{G}$ that eliminates every square root and division in the Boolean sub-expressions of a program p:

- if p in E_u then
  - if $\mathsf{p} = \neg\ \mathsf{p2}$ or $\mathsf{p} = \mathsf{p1\ op\ p2}$ with $\mathsf{op} \in Cbop \cup Bbop$ then $Elim_{\mathbb{B}}(\mathsf{p})$
  - else return p
- else apply recursive calls on the sub terms.

Then by first transforming the variable definition and calls into the $\mathsf{P}_{\mathsf{B}_{\sqrt{,/}}}$ subtype and then eliminating the square roots and divisions in the Boolean expression we can transform any program with function into a new program that is square root and division free (as usual except in the returned numerical expression) that is a program in $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$.

ALGORITHM 4.27 (Main square root and division elimination with functions). The main elimination algorithm is only the composition of these 2 steps:

$$FElim(p) = Elim^{G}_{\mathbb{B}}(Elim_{\mathcal{V}ar}(p))$$

This function satisfies the specification of Definition 4.6 with the new definition of $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$:

THEOREM 4.15 (Main *Elim* function).

$$\forall p \in \mathsf{Prog},\ FElim(\mathsf{p}) \in \mathsf{P}_{\mathsf{N}_{\sqrt{,/}}} \ \wedge\ sem_ty_eq(\mathsf{p}, FElim(\mathsf{p}))$$

This algorithm enables the transformation of any program with functions whose dependency graph is acyclic, however this is not the case of every program we can build with the language we introduced. We will now see how, in a future work, it might be possible to transform programs with cyclic dependency graphs.

4.5 TOWARDS ACYCLIC GRAPHS AND LOOPS

The ideas introduced in this section have not been implemented or tested, even if some programs have been manually transformed using such ideas as a pre-process of the transformation. They are only leads to extend the transformation to larger sets of programs with functions whose dependency graph contains cycles or even programs from a language that contains loops.

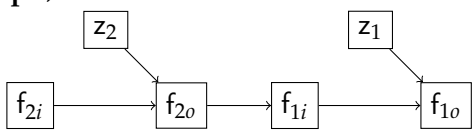
The first idea to transform programs whose dependency graph is not acyclic is to break the cycles with a duplication of the functions that are involved in these cycles.

4.5.1 Function duplication

If every function of the program is only called once then one might prove that the dependency graph is acyclic. Therefore by duplicating function definitions as many times as they are called, we have an acyclic graph. The Example 4.13 can be transformed into the following program whose dependency graph is acyclic.

EXAMPLE 4.14 (Breaking cyclic dependency graph).

$\mathsf{letf}\ \mathsf{f}_1\ \mathsf{x}_1 = \mathsf{let}\ \mathsf{z}_1 = e\ \mathsf{in}\ \mathsf{x}_1 + \sqrt{\mathsf{z}_1};$
$\mathsf{letf}\ \mathsf{f}_2\ \mathsf{x}_2 = \mathsf{let}\ \mathsf{z}_2 = e\ \mathsf{in}\ \mathsf{x}_2 + \sqrt{\mathsf{z}_2};$
$\mathsf{f}_1(\mathsf{f}_2(\mathsf{s}))$



We do not have to duplicate every function for every call but we still can do this transformation on functions involved into cycles until the graph becomes acyclic.

A cycle has to involve at least one function since the variables that a variable definition of x depends on were defined previously to x or in the body of the definitions, and x can not be used in these parts of the program. In fact, for most of the dependencies (x,y), x was defined previously to y or in its definition body where y is not yet defined. The only special case is (x,f_i) (x in a call of f) where f_i depends on x and x can be defined in the scope of f, this is the only *backward* edge and it is responsible of the cyclic dependencies.

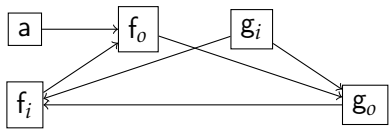
In some cases we also can handle the cyclic dependencies by using a template fixpoint without having to duplicate the functions.

4.5.2 Template fixpoint

Let us consider the following program whose dependency graph contains a cycle:

EXAMPLE 4.15.

```
let a = e in
letf f x = 2x + 3√a;
letf g y = f(y) + y + 5;
f(g(n)) > 0
```



Even if this program contains a cycle ($f_i \to f_o \to g_o \to f_i$) it can be transformed, using $s + t \cdot \sqrt{a}$ as a template for f and g:

EXAMPLE 4.16 (Common template with dependency graph).

```
let a = e in
letf f (x1,x2) = (2x1,2x2 + 3);
letf g (y1,y2) =
    let (u,v) = f(y1,y2) in (y1+u+5, y2+v);
let (f1,f2) = f(g(n)) in f1 + f2 · √a > 0
```

This is because $s + t \cdot \sqrt{a}$ is what we call a fixpoint template for the expressions x + y + 5 and 2x $+3 \cdot \sqrt{a}$, that is $s + t \cdot \sqrt{a}$ is a template of the expression after replacing the free variable by this template itself.

DEFINITION 4.28 (Fixpoint Template). Given two terms t and e and their set of substitution variables $x_1, ..., x_n$ and $y_1, ..., y_m$ then t is a constrained fixpoint template for e if t is a constrained template of

$$e[y_1 \mapsto t[x_1 \mapsto y_{11}; ...; x_n \mapsto y_{1n}]; ...; y_m \mapsto t[x_1 \mapsto y_{m1}; ...; x_n \mapsto y_{mn}]]$$

Where the y_{ij} are new fresh variables

EXAMPLE 4.17.

$$2(s + t \cdot \sqrt{a}) + 3 \cdot \sqrt{a} = 2s + (2t + 3) \cdot \sqrt{a}$$
$$(s_1 + t_1 \cdot \sqrt{a}) + (s_2 + t_2 \cdot \sqrt{a}) + 5 = (s_1 + s_2 + 5) + (t_1 + t_2) \cdot \sqrt{a}$$

We do not know yet how to compute such a template and it seems to us that it does not always exists, for example a fixpoint template for $\sqrt{x}$ might not be possible since the depth of the most nested square root in $\sqrt{t}$ is one more than the deepest one in t. However, when they exist, such fixpoints could be used to handle some loops, as in the following imperative program:

EXAMPLE 4.18 (Fixpoint template for loops).

```
while x < 0 do                          x1 := x;
    x := 2x + √a;          ⟶           x2 := 0;
done                                    while x1× x1 - x2× x2 × a < 0 do
                                            x1 := 2x1;
                                            x2 := 2x2 + 1;
                                        done
```

Once again these are only hints for the extension of the program transformation to richer languages and we have not tackled this problem. However it could be worth to study the cases where such fixpoints exist and therefore enable the transformation of such programs.

4.5.3 The Division Case

There is one particular case that always allows a fixpoint template, it is the absence of square roots. For expressions that only contain divisions and no square roots then s/t is a fixpoint template for these expressions. Indeed, every expression that is square root free can be transformed into an equivalent expression with only one head division using the reduction rules from Definition 1.6. Therefore division can be eliminated from any program by only using 2 variables, representing the numerator and the denominator for every variable of the input program.

PROPOSITION 4.16 (Division elimination). *Given a program with no square root then s/t can be used as the template by any of the $Elim_{let}$, $Elim_{fin}$ and $Elim_{fout}$ rules.*

This template would also allow us to transform any programs, with loops, recursion or any other features, by using 2 variables for numerator and denominator:

EXAMPLE 4.19. Using the syntax of OCaml with the recursive definition let rec:

```
let rec f x =                           let rec f xn xd =
    if x > 1                                if xn× xd > xd× xd
    then f(x/(x+1))                         then f(xn,(xn+xd))
    else x / 3;            ⟶               else (xn,3xd);
let g x = f(x + b);                     let g xn xd = f(xn + bxd,xd);
  f(a/c + g(d)) > e                     let (gn,gd) = g(d,1) in
                                        let (fn,fd) = f(a×gd + c×gn, c×gd) in
                                          fn× fd > e × fd× fd
```

Therefore any program with only divisions can easily be transformed into a division free program. Transformation is more complicated with square root since there is no such normal form that every expression can take.

Conclusion We have now extended our program transformation to programs with function that satisfies a certain hypothesis on the dependencies between the variable and functions that are defined in it. Once again, we used the anti-unification introduced in Chapter 2 as the core of our transformation and introduced an extended notion of anti-unification that might enable us to transform program from an even richer language. This concludes the description of the transformation algorithm, in the following part we present how we have been able to prove in PVS and implement in OCaml such a transformation.

Part II

Implementations and Applications

5 Formal PVS Proof

Programs we are willing to transform require a very high level of safety, therefore we want to use formal proof assistant to achieve such a goal. In particular, we want to ensure that the transformation is correct, that means it preserves the semantics as introduced in its specification in Definition 4.6. Therefore we formalized this transformation in the PVS proof assistant [ORS92] and proved its correctness. The proof is done on the transformation without function definitions corresponding to the program *Elim* introduced in Chapter 3. This chapter follows the description of the algorithms in Chapter 1 and Chapter 3. We focus on the specificity of the formalism of the proof, the proof itself being described in these chapters. In order to present this formal proof, we briefly introduce the PVS proof assistant and then present how we have specified the transformation and proved its correctness in this system.

5.1 The PVS Proof Assistant

The PVS proof assistant is based on classical typed logic with an extended use of powerful sub-typing features. Indeed many correctness properties are expressed in the type of the objects instead of using predicates. The specification of an algorithm and the related proofs in PVS mainly rely on the PVS 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 verify `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,x')}
```

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

i) (*Completeness*) `f` can be applied to every element of type `(P)`
   Indeed there are no partial functions in PVS, therefore we have to prove that for every input, one case of the function deals with it. Using a predicate as `P` allows us to restrict this domain and therefore to define these partial functions.

ii) (*Soundness*) $\forall$ `x : T, f(x) : T'` $\wedge$ `P'(x,f(x))`
This is the correctness lemma of the function, we want to ensure that the element returned by the function verifies its specification.

iii) (*Recursive function*) if `f` is recursive, then for every recursive call on `e`:

a) (*Recursive call*) `P(e)`
Indeed we have to be sure that the recursive calls are made on elements that are in the domain of the function.

b) (*Termination*) `measure x by <`
In order to ensure the termination we have to provide a measure that decreases, according to a well founded order, in the definition of `f`. Then we need to prove that recursive calls are made on terms smaller than the input.

A well founded order is defined in PVS as an order such that every set has a minimal element, therefore there can exist no infinite decreasing sequence:

```
well_founded?(<): bool =
 (FORALL p:
    (EXISTS y: p(y))
       IMPLIES (EXISTS (y:(p)): (FORALL (x:(p)): (NOT x < y))))
```

The type of a function can also be restricted using the `HAS_TYPE` judgment, *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 judgment:

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

and once the according type checking conditions are proven, use either `T'` or `S'` as type of `f(x)` when `x` has type `S`. Therefore most of the time we will not give any correctness lemmas for the functions but we will encode the properties we need in the type. Indeed, properties encoded in the type are much easier to prove since the type checking condition generation already decomposes the function according to the different cases.

EXAMPLE 5.1 (PVS type checking condition generation). Given the following function that adds an element to a list if the element is not yet in the list:

```
add(x : T, l : list[T]) : RECURSIVE {nl : list[T] | member(x,nl)} =
    CASES l OF
       null : cons(x,l),
       cons(a,q) : IF x = a THEN l ELSE cons(a,add(x,q)) ENDIF
    ENDCASES
    MEASURE l BY <<
```

PVS generates the following set of type checking conditions that have to be proved to complete the type-checking of the definition:

```
add_TCC1: OBLIGATION
  FORALL (x: T, l: list[T]): l = null IMPLIES
    member[T](x, cons[T](x, l));

add_TCC2: OBLIGATION
  FORALL (x: T, l: list[T], a: T, q: list[T]):
    x = a AND l = cons(a, q) IMPLIES member[T](x, l);
```

```
add_TCC3: OBLIGATION
  FORALL (x: T, l: list[T], a: T, q: list[T]):
    NOT x = a AND l = cons(a, q) IMPLIES <<[T](q, l);

add_TCC4: OBLIGATION
  FORALL (x: T, l: list[T],
          v:
            [d1: {z: [T, list[T]] | z`2 << l} ->
               {nl: list[T] | member(d1`1, nl)}],
          a: T, q: list[T]):
    NOT x = a AND l = cons(a, q) IMPLIES
         member[T](x, cons[T](a, v(x, q)));
```

where << is the sub-term order on lists.

Therefore if `x` is already in `l` then `add(x,l) = l`, this can be stated as a lemma or a new typing judgement:

```
  add_member : LEMMA
           FORALL (x : T, l : list[T]) :
                  member(x,l) IMPLIES l = add(x,l)

  add_member_st : RECURSIVE JUDGEMENT
           add(x : T, l : list[T] | member(x,l))
                  HAS_TYPE { nl : list[T] | nl = l}
```

Given these features we describe the proof that the program transformation *Elim* as defined in 3.29 satisfies its specification as stated in Theorem 3.9.

5.2 PVS FORMALIZATION

The first step of the formalization is to define the abstract syntax of the program that corresponds to the programs in Prog introduced in Definition 3.1. Thus we define the following datatype in PVS:

DEFINITION 5.1 (PVS abstract syntax).

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

`V` is an abstract set of identifier for variables, however in this PVS specification and proof we do not handle multi-variable definition, therefore we use projections to replace the multi-variable definitions:

EXAMPLE 5.2.

$$\mathsf{let}\ (x,y) = e\ \mathsf{in}\ sc \quad \longrightarrow \quad \mathsf{let}\ x = e\ \mathsf{in}\ sc[x \mapsto \mathsf{fst}(x),\ y \mapsto \mathsf{snd}(x)\ ]$$

A PVS datatype also includes several functions for each constructor that characterize the head constructor and allow direct access to the sub-terms:

```
letin?(p) = EXISTS x,p1,p2 : p = letin(x,p1,p2)
body(letin(x,p1,p2)) = p1
```

Once again, after defining the syntax we need to define in PVS the type of a program and its semantics. The type of the program can be numerical, Boolean, a pair, or a failure, it is represented with the following datatype:

```
    type_value : DATATYPE
         BEGIN
            numt: numt?
            boolt: boolt?
            pairt(tl : type_value, tr : type_value): pairt?
         END type_value

    mb_type : TYPE = Maybe[type_value]
```

where `Maybe[T]` is equivalent to the option type which is `T` $\cup$ `{None}` where `None` represent the failure of the type inference. Therefore we define the type inference of a program as the following function:

```
type_infer(p : program, env : type_environment) : RECURSIVE mb_type
```

which returns the type in the environment that associates a value to every variable:

```
type_environment : TYPE = [ V -> type_value]
```

In the same way we define the semantics of a program:

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

program_environment : TYPE = [ V -> prog_value]

mb_value : TYPE = Maybe[prog_value]

semantics(p : program, env : program_environment) : RECURSIVE mb_value
```

Given the definition of the semantics and types, their preservation, as introduced in Definition 3.9 corresponds to the following predicate:

```
preserves_semantics_and_type(p : program)(pp : program) : bool =
  (FORALL tenv : (some?(type_infer(p,tenv))) IMPLIES
     type_infer(p,tenv) = type_infer(pp,tenv)) &
  (FORALL env : (some?(semantics(p,env))) IMPLIES
     semantics(p,env) = semantics(pp,env))
```

As in Chapter 3 we also define some subtypes of the `program` datatype that allow square roots and divisions to appear in different sub-terms, they correspond to the languages introduced in Definition 3.5, 3.7 and 3.8. Therefore we define some inductive predicate to characterize these languages, *e.g.*,

```
is_expression(p : program) : INDUCTIVE bool =
 is_expression_unit(p) OR
 pair?(p) & is_expression(pl(p)) & is_expression(pr(p))
```

```
is_program_P(p : program) : INDUCTIVE bool =
 is_expression(p) OR
 letin?(p) & is_program_P(body(p)) & is_program_P(scope(p)) OR
 ift?(p) & is_program_P(fm(p)) &
   is_program_P(prt(p)) & is_program_P(prf(p))
```

Our goal is to define an executable function that eliminates square roots and divisions from Boolean parts of a program and that preserves the semantics and type. Given the previously defined predicates, we specify the main function we want to define:

DEFINITION 5.2 (Main Transformation).

```
main_elim(tenv)(p : program | some?(type_infer(p,tenv))) :
            {pp : program_N_sq | preserves_semantics_and_type(p)(pp)}
```

Where `program_N_sq` corresponds to the $\mathsf{P}_{\mathsf{N}_{\sqrt{,/}}}$ set of programs defined in Definition 3.8 and `tenv` is a typing environment provided for the proof of the transformation. Indeed the transformation is complete and terminates only on well typed programs, having a program well typed allows us to restrict the syntactic forms that are contained in the program we transform, for example in a well typed expression, a projection can only be applied to a variable or another projection:

```
proj_subterm_restrict : LEMMA
 FORALL (x : well_typed_program | is_expression(x)), (y : program) :
  (fst?(x) OR snd?(x)) & y << x IMPLIES fst?(y) OR snd?(y) OR value?(y)
```

Therefore in order to ensure such properties, we need to have an environment where the program is well typed. It has to be explicit in order to compose these properties *e.g.*,

```
 (some?(type_infer(p1,tenv))) & (some?(type_infer(p2,tenv))) IMPLIES
                                 (some?(type_infer(pair(p1,p2),tenv)))
```

***Remark* 5.1.** This explicit environment is only used to ensure the termination by enforcing syntactic properties of the program we transform. Therefore it is only used in the type and never in the body of the transformation, the transformed program does not depend on the environment we provided.

We now present how to implement in PVS the transformation algorithm introduced in Chapter 3. The first step of this algorithm is to transform any `program` into an equivalent one in normal form, *i.e.*, that satisfies the `is_program_P` predicate.

5.3 PROGRAM NORMAL FORM

A program is a program in normal form, *i.e.*, in `p_norm = (is_program_P)`, when it corresponds to the definition of P introduced in Chapter 3. Before presenting this normalization we first need to define a substitution that does not capture variables.

5.3.1 Substitution

The substitution that replaces the variable `x` by the program `e` in `p` is defined by the function `replace` that has the following type:

```
replace(x : V, e,p : program) : RECURSIVE { pp : program |
   (FORALL (env) : some?(semantics(e,env)) IMPLIES
      semantics(p,env WITH [(x) := val(semantics(e,env))])
      = semantics(pp,env)) &
   (FORALL (tenv) : some?(type_infer(e,tenv)) IMPLIES
      type_infer(p,tenv WITH [(x) := val(type_infer(e,tenv))])
      = type_infer(pp,tenv)) &
   (let_number(e) = 0 IMPLIES let_number(pp) = let_number(p))} =
```

The substitution preserves the behavior, that is the type and the semantics when the expression used to replace the variable does not fail. The `let_number` property will be used to prove termination of recursive call on programs where some variables have been replaced. It also allows us to prove the termination of the `replace` function itself using the composition of the `let_number` measure and the program sub-term order introduced in Section 3.5.1. Indeed in the variable definition case, to avoid variable capture, we make a recursive call on program where a variable has already been substituted:

```
replace(x : V, e,p : program) ... =
CASES p OF
 ...
 letin(y,e1,e2) :
  IF y = x
  THEN letin(y,replace(x,e,e1),e2)
  ELSIF notin?(get_FV(e))(y)
  THEN letin(y,replace(x,e,e1),replace(x,e,e2))
  ELSE
   LET yp =
    fresh_name_fv(y,add(x,disjunct_union(get_FV(e2),get_FV(e)))) IN
    letin(yp,replace(x,e,e1),replace(x,e,replace(y,value(yp),e2)))
  ENDIF,
```

Therefore we have to prove that for a certain order `replace(y,value(yp),e2) < e2` this is done using this $<_{letins}$ order. The substitution being defined we present the program normalization.

5.3.2 Program Normalization

In Definition 3.17 we introduced a set of rules that enables the transformation of any `program` into an equivalent one in `p_norm`. This transformation has to preserve the semantics and type of the program but we also need to prove that it terminates. In a programs in `p_norm`, the variable definitions and tests never appear in arguments of other operators such as Boolean or arithmetic ones. Therefore we will first define functions that transform the application of operator and projection to a program in normal form into a program where such operator is no longer the head one, for example we present the switch with unary operator:

```
uop_p_norm_switch(op: unop)(p: p_norm | well_typed_program?(uop(op,p))):
  RECURSIVE {pp: p_norm | preserves_semantics_and_type(uop(op,p),pp) } =
    CASES p OF
      letin(x,e1,e2) :
            letin(x,e1,uop_p_norm_switch(op)(e2)),
      ift(f,e1,e2) :
            ift(f,uop_p_norm_switch(op)(e1),uop_p_norm_switch(op)(e2))
    ELSE uop(op,p)
```

```
    ENDCASES
    MEASURE p BY <<
```

We define the same kind of function for the binary operators, the projections and the pair. Given these functions, we can define the main function that apply the switch to all the operators that contain variable definition or test:

```
p_norm_red( p : (prog_not_clear) | well_typed_program?(p) ) :
 RECURSIVE { pp : p_norm | preserves_semantics_and_type(p,pp) } =
  CASES p OF
   uop(op,e1) :
    LET p1 = p_norm_red(e1) IN uop_p_norm_switch(op)(p1),
   ...
   letin(x,e1,e2) :
     letin(x,p_norm_red(e1),p_norm_red(e2)),
   ift(f,e1,e2) :
     ift(p_norm_red(f), p_norm_red(e1), p_norm_red(e2))
  ELSE p
  ENDCASES
  MEASURE p BY <<
```

This implementation of the reduction might not be the most efficient, however, the termination of this strategy is quite easy to prove and given the size of the programs we transform and that this reduction is only used once, the efficiency is not an issue. Thus, this reduction produces a program in `p_norm` that is equivalent if the input program does not fail. Indeed we need this hypothesis since we want to reduce the projections on pairs *e.g.*, `fst(pair(e1,e2))` $\longrightarrow$ `e1` that would change the semantics if `e2` were failing in the input program. The first step of the transformation being introduced, in the following section we present the proof of another of the algorithms used in the global transformation, the $Elim_{\mathbb{B}}$ algorithm introduced in Chapter 1.

5.4 $Elim_{\mathbb{B}}$ PROOF

The $Elim_{\mathbb{B}}$ algorithm is completely formalized and proved in the PVS proof assistant, indeed we have been able to entirely define and prove a function with the following type:

```
elim_bool(tenv)(e : expr | wt_bool_expr(tenv)) :
 RECURSIVE { es : (is_bool_expr) | preserves_semantic_and_type(e)(es)} =
```

Where `wt_bool_expr(tenv)` is the set of expressions that have a type bool in the environment `tenv` and `(is_bool_expr)` is the B_{let} subtype introduced in Definition 3.7. This function relies on the elimination of the square roots and divisions in all the comparisons that are sub-terms of this Boolean expression. A comparison is a Boolean expression whose head constructor is a comparison:

```
comparison(tenv) : TYPE = { x : typed_program(tenv,boolt) |
  bop?(x) & is_compop(bop(x)) &
    is_num_expression_unit(pl(x)) & is_num_expression_unit(pr(x))}
```

where `is_compop(op)` means that `op` is a comparison operator, *i.e.*, in *Cbop* as defined in Definition 3.7 and `(is_num_expression_unit)` is the subtype of the `programs` corresponding to the set of arithmetic expressions as introduced in Definition 1.1. Therefore the core of the elimination of square roots and divisions in Boolean expression will be

done on such comparison expressions. The $Elim_{\mathbb{B}}$ algorithm introduced in Chapter 1 relies on a sequence of four transformations:

i) Factorize the arithmetic expressions with division as head operator

ii) Eliminate this head

iii) Factorize the arithmetic expressions with one top level square root

iv) Eliminate this square root

By iterating this sequence we eliminate all the square root and division operations in the original comparison, this iteration terminates since the number of sub-terms whose square root is the head operators strictly decrease after the sequence. Therefore in order to prove the correction of this transformation we have to prove that:

- each rule preserves the semantics and the type
- each rule preserves the number of square root sub-terms
- at least one rules makes the number of square root strictly decrease

It is the rule iv which eliminates the chosen square root that makes this square root sub-terms number strictly decrease. We now detail how these four rules are implemented so that we can prove these properties.

5.4.1 Head Division Form

We define the inductive predicates stating that a numerical expression is in *PF* (denoted by `pf?` in PVS) or *HDF* (`hdf?`) forms as introduced in Definition 1.5. Therefore the first step of the transformation is to reduce both arguments of the comparison into these forms, this is done with the following function:

```
to_hdf(tenv)( e : wt_num_expr(tenv) ) : RECURSIVE
  { eout : wt_num_expr(tenv) | hdf?(eout) &
     preserves_semantic_and_type(e)(eout) & preserve_sq(tenv)(e,eout) }
```

This function implements the set of reduction rules introduced in Definition 1.6 extended with the rules when both arguments of the head operator have a division as head operation:

DEFINITION 5.3 (Head division reduction).

$$\frac{e_1}{e_2} + \frac{e_3}{e_4} \longrightarrow \frac{e_4 \times e_1 + e_2 \times e_3}{e_2 \times e_4} \qquad \frac{e_1}{e_2} \times \frac{e_3}{e_4} \longrightarrow \frac{e_1 \times e_3}{e_2 \times e_4} \qquad \frac{\frac{e_1}{e_2}}{\frac{e_3}{e_4}} \longrightarrow \frac{e_1 \times e_4}{e_2 \times e_3}$$

Therefore the implementation of the reduction is straightforward, for any expression `bop(op,e1,e2)` we first reduce `e1` and `e2` and then depending if they have a division as head operation or not, we apply the according rule. The termination is straightforward since recursive calls are only made on direct sub-terms. Therefore we can prove that this transformation:

- Preserves the semantics and the type, *i.e.*, `preserves_semantic_and_type(e)(eout)` using the properties of the division in the arithmetic theory

- Returns a program in *HDF* form, *i.e.*, `hdf?`, the only division not under a square root is the head operator

- Preserves the square root sub-terms since we do not reduce under the square roots:

```
preserve_sq(tenv)(cin, cout : wt_num_expr(tenv)) : MACRO bool =
  FORALL (x : wt_num_expr(tenv)) :
    (uop(sqrt,x) << cout OR uop(sqrt,x) = cout) IMPLIES
        (uop(sqrt,x) << cin OR uop(sqrt,x) = cin)
```

  This predicate states that the square roots that are sub-terms of the output were sub-terms of the input.

This first transformation being implemented we can now define the elimination of the head division in a comparison between two expressions in *HDF*.

5.4.2 Division Elimination

In the PVS specification we only use the division elimination that does not use the case distinction, *i.e.*, this transformation simply implements the rules introduced in definitions 1.7 and 1.9. It allows us to define a function that has the following type:

```
elim_div_rule(tenv)(p : comparison(tenv)) :
 { x : comparison(tenv) | pf?(pl(x)) & pr(x) = zero &
    preserves_semantic_and_type(p)(x) & preserve_sq_comp(tenv)(e,x) }
```

Once again this function:

- produces a comparison between an expression in *PF* form and 0

- preserves the semantics and the type, using lemmas such as:

```
 square_div_mult_gt : LEMMA
      n1/d1 > n2/d2 IFF n1 * d1 * d2 * d2 - n2 * d1 * d1 * d2 > 0
```

- preserves the square roots sub-terms

The right argument of the comparison is now zero, therefore the next step is the factorization of the left one using one top level square root.

5.4.3 Square Root Factorization

We want to define a function that, given a square root argument q, factorizes an expression in *PF* in a form $p.\sqrt{q}+r$, this is done by the following function:

```
factorize_sq(tenv)(e : pnf_type(tenv))(sq : wt_num_expr(tenv) ) :
 RECURSIVE  { e1,e2 : wt_num_expr(tenv) |
   pre_sem_type_w_sq(e)(uop(sqrt,sq))
                    (bop(plus,bop(times,e1,(uop(sqrt,sq))),e2))}
```

Semantics and type preservation In order to preserve the semantics and the type, we have to ensure that not only the input expression `e` but also the square root expression do not fail. Indeed when the reference square root $\sqrt{sq}$ is not a sub-term of the expression we want to factorize e then we transform e into $0.\sqrt{sq} + e$. This may happen in the binary operator case since the square root is not always a sub-term of both arguments. Moreover, for efficiency, we do not want to check that the reference square root is a sub-term of each argument but make straightforward calls:

```
bop(op,e1,e2) :
  LET (p1,r1) = (factorize_sq(tenv)(e1)(sq)) IN
  LET (p2,r2) = (factorize_sq(tenv)(e2)(sq)) IN
  IF plus?(op)
  THEN (bop(plus,p1,p2),bop(plus,r1,r2))
  ELSIF ...
```

Therefore, to ensure the preservation of the semantics we have to suppose that the reference square root does not fail in the environments where we want to prove the semantics and type equality. This is why we use a new predicate for semantics and type equivalence:

```
pre_sem_type_w_sq(p: program)(sq: program)(pp: program) : bool =
 (FORALL (tenv)  : (some?(type_infer(p,tenv)) &
    numt?(val(type_infer(p,tenv))) & some?(type_infer(sq,tenv))) IMPLIES
                    type_infer(p,tenv) = type_infer(pp,tenv)) &
 (FORALL (env) : (some?(semantics(p,env)) &
    numv?(val(semantics(p,env))) & some?(semantics(sq,env))) IMPLIES
                    semantics(p,env) = semantics(pp,env))
```

Since the reference square root that will be chosen is always a sub-term of the expression we want to transform, we will be able to prove these hypothesis and therefore get the semantics and type equivalence.

To prove the termination In order to prove the termination of the transformation we need to make the number of square root sub-term decrease, therefore we have to ensure that `uop(sqrt,sq)` is not a sub-term of the returned arguments. In order to have this property we need to chose a top level square root (that is not nested). As mentioned in Proposition 1.4 every expression that contains a square root has at least one top level square root, for example we can take the maximum for the sub-term order

```
max_subterm(tenv)( l : { ls : list[wt_num_expr(tenv)] | cons?(ls)} ) :
 RECURSIVE { p : wt_num_expr(tenv) | member(p,l) &
   FORALL (x : wt_num_expr(tenv)) : member(x,l) IMPLIES NOT p << x } =
```

By applying this function to the list of the square root sub-terms of a program, we have this top level square root. By using such a square root for the factorization we can ensure that the square root sub-terms of the returned expressions `e1` and `e2` were square root of the input and are different from `sq` as stated by the following predicate.

```
 prsrv_sq_ref(tenv)(e : wt_num_expr(tenv))
     (sq : wt_num_expr(tenv))(x : wt_num_expr(tenv)) : MACRO bool =
   FORALL (s: wt_num_expr(tenv)) :
     (uop(sqrt,s) << x OR uop(sqrt,s) = x) IMPLIES
    (s /= sq AND (uop(sqrt,s) << e
      OR uop(sqrt,s) = e OR uop(sqrt,s) << uop(sqrt,sq) ))
```

Thus we can add the following type to the factorization function:

```
factorize_square_root_preserve_sq : RECURSIVE JUDGEMENT
  factorize_square_root(tenv)(e : pnf_type(tenv))
    (sq : wt_num_expr(tenv) | (FORALL (s : wt_num_expr(tenv)) :
      ((uop(sqrt,s) << e OR uop(sqrt,s) = e) IMPLIES
         NOT (uop(sqrt,sq) << (uop(sqrt,s)))))) HAS_TYPE
  {e1,e2 : wt_num_expr(tenv) |
    (prsrv_sq_ref(tenv)(e)(sq)(e1) AND prsrv_sq_ref(tenv)(e)(sq)(e2))}
```

Therefore this function returns numerical expressions e1 and e2 such that e1.$\sqrt{sq}$ + e2 has the same type and semantics and their square root sub-terms were sub-terms of e and are different from sq.

The only step left is to eliminate this square root and then we will have completed that sequence of four transformation.

5.4.4 Square root elimination

We now want to eliminate the square root that has been used to factorize the left member of the comparison (the right one still being zero). This will be done using the rules introduced in Definition 3.20 using names for the produced comparisons. However the previously defined rules can only be applied to comparison expressions and this rule produces a program with variable definition and Boolean operators. Therefore we first apply the recursive call on the new comparisons where the top level square root have been eliminated before combining them to produce the program equivalent to the input comparison. Therefore the elimination rules are specified in the following way:

```
elim_sqrt_rule_gt(tenv)(p,q: wt_num_expr(tenv))(sq: wt_num_expr(tenv)):
 { e1, e2, e3, e4 : comparison(tenv) |
   FORALL (x : V) :
      preserves_semantic_and_type
        (bop(gt,bop(plus,bop(times,e1,(uop(sqrt,sq))),e2),zero))
        (elim_sqrt_name_composition(x,e1,e2,e3,e4)) } =
```

Where `elim_sqrt_name_composition(x,e1,e2,e3,e4)` is the naming and the combination of the atoms according to the rule for $>$,

```
elim_sqrt_name_composition(x : V, e1, e2, e3, e4 : program) : program =
 letin(x,pair(pair(e1,e2),pair(e3,e4)),
  bop(orf,
      bop(andf,fst(fst(value(x))),snd(fst(value(x)))),
  bop(orf,
      bop(andf,fst(fst(value(x))),fst(snd(value(x)))),
      bop(andf,snd(fst(value(x))),
               bop(andf,uop(notf,fst(snd(value(x)))),
                        snd(snd(value(x))))))))
```

By denoting the projection ‘1 and ‘2 the concrete syntax of this expression is:

let x = ((e1,e2),(e3,e4)) in x‘1‘1 ∧ x‘1‘2 ∨ x‘1‘1 ∧ x‘2‘1 ∨ x‘1‘2 ∧¬ x‘2‘1 ∧ x‘2‘2

By returning the 4 atoms we are able to apply the recursive calls before combining the produced programs with this `elim_sqrt_name_composition`.

In order to prove the termination we also need to prove that the targeted square root has indeed disappeared:

```
elim_sqrt_rule_gt_preserves_sq : JUDGEMENT
  elim_sqrt_rule_gt(tenv)(p,q : wt_num_expr(tenv))
    (sq : wt_num_expr(tenv) | FORALL (s : wt_num_expr(tenv)) :
```

```
      (uop(sqrt,s) << p OR uop(sqrt,s) = p OR
       uop(sqrt,s) << q OR uop(sqrt,s) = q) IMPLIES (s /= sq))
    HAS_TYPE
     { e1, e2, e3, e4 : comparison_expression(tenv) |
        prsrv_sq_ref_comp(tenv)(p,q)(sq)(e1) AND
        prsrv_sq_ref_comp(tenv)(p,q)(sq)(e2) AND
        prsrv_sq_ref_comp(tenv)(p,q)(sq)(e3) AND
        prsrv_sq_ref_comp(tenv)(p,q)(sq)(e4)
     }
```

Where `prsrv_sq_ref_comp` is analogous to `prsrv_sq_ref` for comparisons. Therefore this rule states that the square roots in each atoms were already either in `p` or `q` or `sq` and are different from `sq`. This allows us to make recursive call on these atoms since the number of their square root sub-terms is lower than in the input atom.

We have now defined the four successive transformation that are required for the elimination of square roots and divisions in a Boolean expression and typed them in order to ensure both the preservation of the semantics and the type and the termination using the number of square root sub-terms. Therefore the only thing left is to combine these four transformations and to apply the recursive calls in order to eliminate all the square roots and divisions. This is done by defining a recursive function that transforms a comparison into a square root and division free Boolean program:

```
elim_bool(tenv)(xsq)(c : comparison_expression(tenv)) :
 RECURSIVE {e : (is_bool_expr)| preserves_semantic_and_type(c)(e) }=
```

It terminates using the number of square root sub-terms of the comparison:

```
MEASURE (length(comp_expression_sq_list(tenv)(c))) BY <
```

By recursively applying this function to any comparison that can be found in a Boolean expression, we are able to completely eliminate square roots and divisions from these Boolean expressions. This defines the function `elim_bool` introduced at the beginning of Section 5.4. The $Elim_{\mathbb{B}}$ function now being specified and proved in PVS, we now present the proof of the other main transformation, the $Elim_{let}$ function introduced in Section 3.7.

5.5 Variable Definition Transformation

In this section we aim at proving the transformation of a variable definition. This transformation relies on a decomposition of the body and then on the transformation introduced in Definition 3.23. When the body of the transformation contains tests, we use the anti-unification algorithm introduced in Chapter 2 in order to compute the common template of the expressions corresponding to the different test cases. This anti-unification algorithm is not specified in PVS but only stated as an axiom. Therefore we assume that we have a function that can produce a template of a set of expressions.

5.5.1 Template

As introduced in Section 3.7.4, given a set of positive expressions and a set of variables that are not allowed to be used, we want the template to have the following PVS property:

Definition 5.4 (Template property). Given a sequence of expressions *se*, a set of known positive arithmetic expressions *spos* and a set of forbidden variables *locvar*, we want the

template computation to return a term t, a variable x and a sequence of expressions *nse* (that represent the substitutions) respecting the following predicate:

```
template_prop(t : expr, x : V, nse : finseq[program])
  (se : finseq[program], spos : finite_set[(is_num_sq_expr)],
   locvar : finite_set[V]) : MACRO bool  =
```

We detail and comment the body of the predicate, it states that:
— *se* and *nse* have the same length:

```
 nse`length = se`length &
```

— the free variables of the template are not in *locvar*:

```
 disjoint?(get_FV(t),locvar) &
```

— in an environment where the elements of *spos* are positive numerical expressions then the semantics of the n-th input expression is the same as the one of the template where x is replaced by the n-th expression of the new sequence:

```
(FORALL (env : program_environment) :
(FORALL (es : (spos)) :
 value?(semantics(es,env)) & numv?(val(semantics(es,env))) &
 r(val(semantics(es,env))) >= 0 ) IMPLIES
(FORALL (n : below(nse`length)) :
 value?(semantics(se`seq(n),env)) IMPLIES
  (value?(semantics(nse`seq(n),env)) &
  semantics(se`seq(n),env) = semantics(replace(x,nse`seq(n),t),env)))) &
```

— in an environment where the elements of *spos* have numerical type then the type of the n-th input expression is the same as the one of the template where x is replaced by the n-th expression of the new sequence:

```
(FORALL (tenv : type_environment) :
(FORALL (es : (spos)) :
 value?(type_infer(es,tenv)) &
 numt?(val(type_infer(es,tenv)))) IMPLIES
(FORALL (n : below(nse`length)) :
 value?(type_infer(se`seq(n),tenv)) IMPLIES
  (value?(type_infer(nse`seq(n),tenv)) &
  type_infer(se`seq(n),tenv)=type_infer(replace(x,nse`seq(n),t),tenv))))
```

This allows us to state that the term `t` is a template for this set of expressions. In order to ensure that it is a constrained template, we also need to state that the new expressions are square root and division free.

It is easy to define such a function when the sequence only has one element. In this case, we only need to replace the sub-expressions of a term that are square root and division free with projection of a variable. However the template construction when the sequence contains more expressions is not specified, we only assume that we have a function with the right type that can handle this case. Therefore we have a partially executable function that computes a constrained template:

```
template(x : V,
 se : { s : finseq[program] |
   all_well_typed(s) & every_fs(is_expr_N_sq)(s)},
 spos : list[program],
 locvar : { l : list[V] |
   FORALL (es : (in?(spos))) :
     is_num_sq_expr(es) & disjoint_sl?(get_FV(es),l)}) :
```

```
  { t : expr, se_free : finseq[program] |
    every_fs(is_expr_N)(se_free) &
    template_prop(t,x,se_free)(se,spos,locvar) } =
```

Given such a template, we describe the other part of the variable definition transformation that is the *Decompose* function from Definition 3.24.

5.5.2 Decomposition

We define this *Decompose* function and prove that it commutes with a template that corresponds to our specification. Once again the size of the type is quite huge due to the number of parameters of such function, we comment all the different properties of the corresponding to the returned type:

DEFINITION 5.5 (*Decompose* in PVS). This function input is a program and a set of numerical expressions:

```
program_part_extract(tenv)(p : program_N_sq | some?(type_infer(p,tenv)),
  spos : list[program] | FORALL (s : (in?(spos))) :
    is_num_sq_expr(s) & some?(type_infer(s,tenv)) &
    numt?(val(type_infer(s,tenv)))) : RECURSIVE
```

It returns the number `n` of test cases (*i.e.,* the number of expression to anti-unify), a function `f` that is the *program part*, the list of expression to anti-unify, the set of local variable and the new set of positive expressions `spos`:

```
 {n: nonneg_int, f: [{s : finseq[program] | s`length = n} -> program],
   (se : finseq[program] | se`length = n ), locvar : list[V],
   spos_out : list[program] |
```

These returned elements verify the following properties:
i) The elements of the sequence are expressions, so we can apply the template computation:

```
 every_fs(is_expr_N_sq)(se) &
```

ii) The program part applied to square root and division free expressions produces a square root and division free program. Therefore we will able to produce a square root and division free program by applying this program part to the square root and division free expressions returned by the template computation:

```
FORALL (fseq : finseq[program] | fseq`length = n) :
    every_fs(is_expr_N)(fseq) IMPLIES (is_program_N(f(fseq))) &
```

iii) The expressions in the sequence have the same type as the input program, in fact we only need to prove that they have the same type for the template computation:

```
(FORALL (k : below(n)) : EXISTS (tenvk : type_environment) :
  type_infer(se`seq(k),tenvk) = type_infer(p,tenv)) &
```

iv) The new positive expressions are the input positive expressions that do not contain a local variable, therefore their semantics is not changed by the definition of the local variables.

```
FORALL (epos : (in?(spos_out))) : in?(spos)(epos) &
  disjoint_sl?(get_FV(epos),locvar) &
```

v) A template commutes with the program part, *i.e.,* $Pp(t(e1), ..., t(en)) = t(Pp(e1, ..., en))$ for the semantics:

```
FORALL (t : expr, x : V, se_free : finseq[program] |
            template_prop(t,x,se_free)(se,spos_out,locvar)) :
 FORALL (env : program_environment) :
 FORALL (es : (in?(spos))) : some?(semantics(es,env)) &
  numv?(val(semantics(es,env))) & re(val(semantics(es,env)))>=0 IMPLIES
   some?(semantics(p,env)) IMPLIES
    (some?(semantics(f(se_free),env)) &
    semantics(p,env) = semantics(replace(x,f(se_free),t),env)) &
```

And for the type:

```
 FORALL (tenv : type_environment) :
 FORALL (es : (in?(spos))) : some?(type_infer(es,tenv)) &
  numt?(val(type_infer(es,tenv))) IMPLIES
   some?(type_infer(p,tenv)) IMPLIES
    (some?(type_infer(f(se_free),tenv)) &
    type_infer(p,tenv) = type_infer(replace(x,f(se_free),t),tenv))
   } =
```

The body of the definition is straightforward using the Definition 3.24.

Given this decomposition function and the template computation we are now able to define the $Elim_{let}$ function as defined in Section 3.7.2. This function transforms a variable definition into a new one whose body is square root and division free by decomposing the body, computing a template and inlining this template in the scope of the definition. This function returns the new body and scope of the definition and update the set of positive expressions by adding the square root sub-terms of the template, preserving the semantics and the type of the program.

```
elim_let(tenv)(x : V, p1 : program_N_sq,
   p2 : p_norm | some?(type_infer(letin(x,p1,p2),tenv)),
   spos : list[program] |
    FORALL (s : (in?(spos))) : is_num_sq_expr(s) &
     some?(type_infer(s,tenv)) & numt?(val(type_infer(s,tenv)))) :
 {pp1 : program_N, pp2 : p_norm, nspos : list[program] |
   every(is_num_sq_expr)(nspos) &
   preserves_semantic_and_type_with_spos(x,p1,p2,spos)(pp1,pp2,nspos) &
   if_letin_number(pp2) <= if_letin_number(p2)
 } =
 LET nspos =
  filter(spos,
    (LAMBDA (sp : (is_num_sq_expr)) : notin?(get_FV(sp))(x))) IN
 LET (n,f,se,locvar,sposout) = program_part_extract(tenv)(p1,nspos) IN
 LET (t,sef) = template(x,se,sposout,locvar) IN
   (f(sef),
    pair_reduction(replace(x,t,p2)),
    disjunct_union(nspos,get_sqrt_expr(t)))
```

The `if_letin_number` hypothesis will be used to prove the termination of the main algorithm that combine this `elim_let` and the `elim_bool` function previously defined. Therefore the only step left is to combine these functions to transform any normalized program into a square root and division free one.

5.6 Main elimination

By using the `elim_let` and the `elim_bool` functions we are able to define the transformation of any normalized program. Indeed we can define the $Elim_{\mathrm{P}}$ algorithm that was introduced in Definition 3.28. As intended, this function preserves the semantics and the type of the program and removes square roots and divisions from it:

```
elim(tenv)((p : p_norm | some?(type_infer(p,tenv))),
   spos : list[program] | FORALL (s : (in?(spos))) : is_num_sq_expr(s) &
    some?(type_infer(s,tenv)) & numt?(val(type_infer(s,tenv)))) :
 RECURSIVE { pp : program_N_sq |
 (FORALL (tenv : type_environment) :
  ((FORALL (es : (in?(spos))) :
     some?(type_infer(es,tenv)) & numt?(val(type_infer(es,tenv)))) &
   some?(type_infer(p,tenv))) IMPLIES
        (type_infer(p,tenv) = type_infer(pp,tenv))) &
(FORALL (env : program_environment) :
  ((FORALL (es : (in?(spos))) : some?(semantics(es,env)) &
     numv?(val(semantics(es,env))) & re(val(semantics(es,env))) >= 0 ) &
   some?(semantics(p,env))) IMPLIES
        (semantics(p,env) = semantics(pp,env))) } =
```

And the final step consist of combining this function with the program normalization, enabling the transformation of any `program`:

```
main_elim(tenv)(p : (prog_not_clear) | some?(type_infer(p,tenv))) :
         {pp : program_N_sq | preserves_semantic_and_type(p)(pp)} =
     elim(tenv)(p_norm_reduction(p),null)
```

The type of this function is exactly the specification of the transformation that we introduced in Definition 4.6.

Conclusion

Except for the template computation, we have completely defined and proved correct the *Elim* transformation in the PVS proof assistant. The modularity of the proof ensures that if the template computation is correct, as specified by the type of the axiomatized template computation, then the complete transformation is correct. Therefore we only have to prove a correct implementation of the constrained anti-unification to complete the proof. Moreover, since the transformation of expressions does not require the template computation, we will see in Chapter 7 how we can define a strategy in PVS that uses this function to transform PVS expressions during a PVS proof.

6

OCaml Implementation

The transformation of programs using functions introduced in Chapter 4 has been implemented in the OCaml Language. OCaml is a typed functional language [LDF+12] that allows us to completely define our transformation of programs. This chapter only deals with the core of the transformation, that is the transformation of the symbolic abstract syntax of the program. The implementation in OCaml of the part of the transformation that has also been defined in PVS being very similar, we will not discuss it in this chapter. Therefore this chapter mainly describes the anti-unification and the function transformation implementations that are used to transform complete programs with function definitions. Questions related to the interfaces, like parsing and pretty printing will be handled in Chapter 7.

We have tried to keep a purely functional style for programming this transformation in OCaml and to have a code as simple as possible in order to be certain about the transformations it does. Therefore we only use imperative features or references for optimizations and some choices in the transformation, the correctness of the transformation being independent of these choices. Moreover we also want to only use basic features in these programs so that they can be easily translated into a PVS specification and hopefully proven correct. Let us first define the abstract syntax of the programs as the type Prog introduced in Definition 4.1. It corresponds to the following type:

Definition 6.1 (OCaml type for programs). This type defines the abstract syntax of the programs:

```
type program =
  | Value of uvar
  | Const of constants
  | UOp of unop * program
  | BOp of binop * program * program
  | Pair of program * program
  | Fst of program
  | Snd of program
  | Letin of var * program * program
  | If of program * program * program
  | App of uvar * program
  | Dfun of uvar * (var * types) * types * program * program
```

where

```
type unop =
  | Sqrt | Umin | Neg
type binop =
  | Plus | Times | Div | And | Or | Neq | Eq | Gt | Geq | Lt | Leq
```

This definition directly corresponds to the Definition 4.1, the variable identifiers are strings and the types corresponds to the Definition 3.2. Therefore all the transformations are going to be defined on that type, transforming every program into an equivalent one that does not contain the Div and Sqrt operators. As one can notice, this syntax is an extension of the one used in Chapter 5 to prove the *Elim* transformation on program without functions. The OCaml implementation of this part of the transformation is almost identical to its PVS specification. The only change is that this program type enables the multi-variable definition since var is the following type:

```
type var =
  | Var of uvar
  | Pairvar of var * var
```

This barely changes the part of the transformation that has been defined and proven in the PVS proof assistant. Therefore we do not detail the OCaml implementation of the $Elim_{\mathbb{B}}$ or $Elim_{let}$ functions which have already been defined and proved in PVS but we focus on the part that have not.

After presenting some general properties of our OCaml transformation we will detail the implementation of the common template computation used in the variable and function definition transformation and then we present the global transformation of programs with function definitions.

6.1 Simplification

We still want to minimize the size of the output program, and for efficiency reason it is easier to simplify programs as soon as we can in the transformation. It avoids us to handle expression part of the program that are not relevant anymore. In order to simplify the expressions all along the transformation, we define some functions corresponding to the different operators that allows us to use the absorbing and neutral properties of some constants:

```
let one = (Const (Num 1.))
let zero = (Const (Num 0.))
let tr = Const (Bool true)
let fs = Const (Bool false)

let plus a b =
  match a,b with
    | Const (Num 0.),_ -> b
    | _,Const (Num 0.) -> a
    | _,_ -> BOp(Plus,a,b)

let div a b =
  match a,b with
    | Const (Num 0.),_ -> zero
    | _, Const (Num 1.) -> a
    | _,_ -> BOp(Div,a,b)
```

In the same way we define the mult, umin, ands, ors, sqrt and neg function that respectively correspond to the Times, Umin, And, Or, Sqrt and Neg operators. All of these functions trivially preserve the semantics when the program does not fail (*e.g.*, $0/0$ reduces to 0). Using these functions instead of the Boolean and arithmetic constructors allows us not to handle all the particular cases with zero, one, true or false that could be used to reduce the size of the returned expressions and programs. These definitions given, we now explain how the anti-unification algorithm defined in Section 3.7.4 is implemented in OCaml.

6.2 Anti-unification Algorithm

The algorithm for constrained anti-unification has been introduced in Chapter 1 and some extra features have been added in Section 3.7.4. This algorithm aims at computing a constrained template of a set of arithmetic expressions such that the associated substitutions do not use any square root or division. It also uses a dag representation in order to minimize the number of square roots in the computed template.

6.2.1 Dag construction

In Section 2.4 we explained that the anti-unification relies on the following normal form of arithmetic expressions:

$$\frac{\sum_{i=1}^{n} a_i \prod_{j_i=1}^{m_i} \sqrt{b_{j_i}}}{\sum_{i=1}^{n} c_i \prod_{j_i=1}^{m_i} \sqrt{d_{j_i}}}$$

Where the a_i and c_i are square root and division free, and the b_j and d_j are also in this form with the division as head operator. In order to represent arithmetic expressions in this normal form with dags we use the following OCaml type:

```
type dag_elt =
  | ExprD of program
  | VecD of (program * dag_elt list) list
  | DivD of dag_elt * dag_elt
  | PtD of int
  | PairD of dag_elt * dag_elt

type dag = dag_elt list;;
```

The semantics of such dags is exactly the one introduced in Definition 2.24, we also want to have normalized dags with only one division at head except under the square roots. In order to transform arithmetic expressions in program into this dag type, we first transform them into a single tree term (a dag_elt with no pointer) and then use pointers to share the square roots arguments. Reduction of an arithmetic expression in program into a dag_elt relies on the following mutually recursive algorithm that computes with these normal forms.

Definition 6.2. We define mutually recursive functions for the addition, multiplication and division of tree and the monomial multiplication ($a_1 . \prod_{sq \in l_1} \sqrt{sq} \times a_2 . \prod_{sq \in l_2} \sqrt{sq}$):

```
let rec monom_mult a1 l1 a2 l2 =
  let ltree = intersect l1 l2 in
```

```
  let lsq = disj_union l1 l2 in
  let ncoef = (fold_right mult_tree ltree (ExprD one)) in
    match lsq with
      | [] -> constant_multiplication (mult a1 a2) ncoef
      | _ -> mult_tree ncoef (VecD([(mult a1 a2,lsq)]));;
```

The monomial multiplication relies on the following equality:

$$a_1 \times \prod_{sq \in l_1} \sqrt{sq} \times a_2 \times \prod_{sq \in l_2} \sqrt{sq} = a_1 \times a_2 \times \prod_{d_i \in (l_1 \cup l_2) \setminus (l_1 \cap l_2)} \sqrt{sq} \times \prod_{d_j \in (l_1 \cap l_2)} sq$$

The division is straightforward:

```
and div_tree t1 t2 =
  match t1,t2 with
    | ExprD _,ExprD _ | VecD _, ExprD _
    | VecD _, VecD _ | ExprD _, VecD _ -> DivD(t1,t2)
    | DivD(e1,e2),_ -> div_tree e1 (mult_tree e2 t2)
    | _,DivD(e1,e2) -> div_tree (mult_tree t1 e2) e1
```

For the multiplication of trees, we only give the VecD case:

```
and mult_tree t1 t2 =
  match t1,t2 with
    ...
    | VecD((a1,l1)::q1), VecD((a2,l2)::q2) ->
        let elet = monom_mult a1 l1 a2 l2 in
        let rec1 = mult_tree (VecD([(a1,l1)])) (VecD(q2)) in
        let rec2 = mult_tree (VecD(q1)) t2 in
          plus_tree elet (plus_tree rec1 rec2)
```

It relies on: $(a+b) \times (c+d) = (a \times c) + (a \times d) + (b \times (c+d))$
And the addition of trees:

```
and plus_tree t1 t2 =
  match t1,t2 with
  ...
```

That implements the following simplification:

$$a_1 \times \prod_{sq \in l_1} \sqrt{sq} + a_2 \times \prod_{sq \in l_2} \sqrt{sq} = (a_1 + a_2) \times \prod_{sq \in l_2} \sqrt{sq}$$

when $l_1 = l_2$ (as set equality)

This allows us to define the transformation of any arithmetic expression by combining their tree representations using the previously defined functions:

```
let rec expr_num_to_tree e =
  match (to_hdf e) with
    | e1 when is_sqrt_div_free e1 -> ExprN e1
    | BOp(Div,e1,e2) ->
       div_tree (expr_num_to_tree e1) (expr_num_to_tree e2)
    | BOp(Mult,e1,e2) ->
       mult_tree (expr_num_to_tree e1) (expr_num_to_tree e2)
    | BOp(Plus,e1,e2) ->
       plus_tree (expr_num_to_tree e1) (expr_num_to_tree e2)
    | UOp(Umin,e1) ->
       constant_multiplication (UOp(Umin,one)) (expr_num_to_tree e1)
    | UOp(Sqrt,e1) ->
       VecD([one,[expr_num_to_tree e1]]);;
```

Termination This algorithms terminates on all the examples that been tested, however we have not been able to prove its termination formally.

CONJECTURE 1. *The transformation of any arithmetic expression into an equivalent normalized* dag_elt *using the algorithm from Definition 6.2 terminates.*

However, given this tree representation we are now easily able to transform such a tree into a dag by using pointers in order to represent square root calls as introduced in Chapter 2. It is easy to find the square root arguments and replace them by pointers in order to construct the equivalent dag. The only thing we have to take care of is the right dependency hypothesis. Therefore we are able to produce well-formed dag as introduced in Definition 2.26 with a function tree_to_dag. We present how to anti-unify these dags in order to use the template in the program transformation.

6.2.2 Dag Anti-unification

The dag anti-unification implementation is really close to the algorithm introduced in Section 3.26. We will first present a version where the length of the template is the maximum of the length of the input dags. Then we will present an extension of this algorithm that was implemented by Raphaël BOST during his internship (see [Bos12]).

Dag extension First step of the template computation using dags is the extension of the smaller dags to the maximum length using undefined elements. In order to represent these undefined elements we simply use the 'a option type of OCaml (*i.e.,* Some 'a | None), None representing the not yet specified elements. This way, given a list of dags we can easily transform these dags into lists of dag_elt option list that have the same length and that represent the extended dags.

Dag permutation Given this list of dags with undefined elements of the same length, we apply different permutations to these dags. If n is the length of the dags we compute all the permutations of $\{1, ..., n-1\}$, permutations being only applied to the tail of the dag, the head element remaining not being moved. Then we try all these different permutations on all the input dags. After the application of every permutation we will check that the dag still satisfies the right dependency hypothesis.

Undefined element replacement Before the anti-unification we need to replace the undefined elements by real terms. As explained in Definition 3.26 we have 3 different choices to replace the undefined elements in the tails of the dags (*i.e.,* lbas), with the following function:

```
let rec replace_none_list_bases lbas lpos def_elt refdag locvar =
```

An undefined element can be replaced by:

- a positive expression in the $\mathcal{P}os$ set (*i.e.,* lpos), that is a parameter of the function. The refdag is one of the longest dags of the list (it does not have any undefined element). The replacement of the k-th element of the tails is done only if all the k-th elements are either undefined or equal to the k-th element of the refdag if this

element is in lpos and if this element does not contain any forbidden variable stored in locvar. Indeed, we use this replacement only in order to apply the identity rule of anti-unification (EI) (*i.e.*, $ctemp(e,e) = e$) and therefore this element will be a sub-term of the final template.

- another element of the same dag. We only use elements on the left since it enforces the left dependency hypothesis and the dags corresponding to replacement with elements on the right can be computed using another permutation:
  if a permutation produces:
  $$\frac{PairD(VectD([(a,[\dot{2}])]), VectD([(b,[\dot{2}])]))\ \|\ \#\ |\ d_2}{}$$
  where # can not be replaced by d_2, then another permutation produces
  $$\frac{PairD(VectD([(a,[\dot{1}])]), VectD([(b,[\dot{1}])]))\ \|\ d_2\ |\ \#}{}$$
  where this replacement can happen.
  Then we change the associated pointers, for example, one pointer $\dot{1}$ can be replaced by a pointer $\dot{2}$, *e.g.*,
  $$\frac{PairD(VectD([(a,[\dot{1}])]), VectD([(b,[\dot{2}])]))\ \|\ d_2\ |\ d_2}{}$$
  As for the permutations we try all the possible pointer changes with respect to right dependency.

- a default element, def_elt. We use the constant zero but this is a parameter of the transformation.

Once the undefined elements have been replaced, the only step left is the anti-unification of these dags.

Dag template computation The template computation is exactly that described in Definition 2.28. We first abstract the dag nodes in order to be only interested by their structure using this corresponding type:

```
type temp_elt =
| ExpT
| DivT of temp_elt * temp_elt
| PairT of temp_elt * temp_elt
| VecT of (int list) list
```

And then compute the common structure of 2 dags. As explained in Proposition 2.1 the process can be iterated to compute the common template of any set of dags:

```
let rec greatest_temp_elt t1 t2 =
  match t1,t2 with
    | ExpT, VecT(l) | VecT(l), ExpT ->
        VecT(if (mem [] l) then l else []::l)
    | DivT(t11,t12),DivT(t21,t22) ->
        DivT(greatest_temp_elt t11 t21, greatest_temp_elt t12 t22)
    | DivT(t1,t2),t3 | t3,DivT(t1,t2) ->
        DivT(greatest_temp_elt t1 t3, t2)
    | ExpT, _ -> t2
    | _, ExpT -> t1
    | PairT(t11,t12),PairT(t21,t22) ->
        PairT(greatest_temp_elt t11 t21, greatest_temp_elt t12 t22)
    | VecT(l1),VecT(l2) ->
        VecT(
```

```
          fold_right
            (fun li l ->
               if exists (equal_comm li) l then l else li::l)
            l1
            l2);;
```

Where equal_comm is the equality of lists seen as sets (they contain the same elements). Given this template abstract structure, we now compute the template of a list of dags. In order to avoid substituting later in order to prevent renaming or other variable conflicts, we do not compute the template as a term with the corresponding the substitutions but as an OCaml function of type (program -> dag) and the corresponding program. Therefore we will be able to chose the variable we want to use at the last moment. Given an abstract dag element we produce the associated template with the following function:

```
let rec temp_elt_to_fundag_elt t =
   match t with
     | ExpT ->
       (fun x -> ExprD x)
     | DivT(t1,t2) ->
       (fun (Pair(x,y)) ->
       DivD(temp_elt_to_fundag_elt t1 x, temp_elt_to_fundag_elt t2 y))
     | PairT(t1,t2) ->
       (fun (Pair(x,y)) ->
       PairD(temp_elt_to_fundag_elt t1 x, temp_elt_to_fundag_elt t2 y))
     | VecT([li]) ->
       (fun ai ->
       VecD([ai,(map (fun x -> PtD x) li)]))
     | VecT(li::l) ->
         let fl = temp_elt_to_fundag_elt (VecT(l)) in
           (fun (Pair(a1,a2)) ->
              let (VecD l2) = fl a2 in
                VecD((a1,(map (fun x -> PtD x) li))::l2))
```

Given this function, we are able to compute the functional template of a set of dags. It is computed node by node, the template of each node being either a constant (if all the nodes are equal) or the function computed by the temp_elt_to_fundag_elt. Then for every set of dags, the construct_template_se_var computes a function, the associated sub-expressions and the associated variable structures (*i.e.*, the returned type is (program -> dag) * (program list) * var) such that:

$$\mathsf{construct_template_se_var\ ldag\ \ldots} = (\mathsf{t,lse,x}) \Longrightarrow \forall\, E,\ [\![\mathsf{t(nth\ k\ lse)}]\!]_E = [\![\mathsf{nth\ k\ ldag}]\!]_E$$

The free variables that are in the template are the ones coming from the equality rule and therefore are contained in expressions in $\mathcal{P}os$:

$$\forall\, p, \mathsf{construct_template_se_var\ ldag\ lpos\ \ldots} = (\mathsf{t,lse,x}) \Longrightarrow \\ FV(\mathsf{t}(p)) \setminus FV(p) \subseteq \{FV(e) \mid e \in \mathsf{lpos}\}$$

This way we know exactly the variable that are used in the template term and can ensure all the properties required either for semantics equality in the program transformation or for the dependency graph.

Template choice This algorithm generates many templates and we have to chose the one we want to use in our transformation to minimize the size of the code produced by this transformation, which mainly depends on the complexity of the elimination in the Boolean expression as explained in Section 1.5. Given a variable definition let $(x_1,\ldots,x_n)$ = b in sc the use of the dag minimize the number of square roots and divisions when all the variables are used together in a Boolean expression *e.g.*, $\sum_i x_i > 0$ since it shares the square roots between these different variables, thus the number of distinct square roots calls in such a Boolean expressions is the length of the tail of the template dag.

Therefore in order to chose the best permutation we have to find another criterion, this is provided by a measure function. For this measure we chose to minimize the opposite case that is for example when all the variables are used separately, *e.g.*, $\bigwedge_i x_i > 0$, and thus the measure function for the template we chose is $\sum_i 4^{\#_{\sqrt{}} x_i}$ where $\#_{\sqrt{}} x_i$ is the number of square roots in the part of the template that corresponds to x_i. This number is an over-approximation of the number of atoms that can be produced by the elimination of the square roots in such formula.

In a future work we might want to specialize that measure function for each template computation, depending on how the variables are really used in the rest of the program to really find the template that fits the best to minimize the size of the transformed program. However this looks far from trivial since the template chosen for the transformation of one definition changes the expressions corresponding to other definitions (as shown by the dependency graph of the program).

We now describe an extension of this template computation where the maximum length is bigger than the maximum length of the input dag.

6.2.3 Template Computation Extension

The work we present in this section has been realized with Raphaël BOST during its master internship. The idea is to implement the extended template as introduced in Section 2.4.4. Let us first discuss a little bit about the complexity of the template computation algorithm.

Complexity Indeed, we want to try many possibilities for our template in order to find the one that will minimize the size of the output code. However the induced complexity is really huge. For the algorithm introduced in Section 6.2.2, given a set of n dags of maximum length l, the number of possible permutations is then $(l!)^n$. In practice, due to the usual structure of programs that do not have many cases with a lot of distinct square roots it is still manageable. However extending the maximum length of the dags made the number of possible template completely explode.

Optimizations In order to manage this complexity, the template computation has to be implemented more efficiently. For example, generating all the permutations in a first step and then applying them was not possible anymore, therefore they have to be generated directly when we want to use then. This permutation generation has been done by using the Steinhaus-Johnson-Trotter algorithm [Joh63].

Another optimization is that the template choices does not depend anymore on a fixed measure function but we wanted to try all the possible template to see how this influences the size of the output program. Therefore every time we have to use a template, we continue the transformation with all the possible templates, modulo some equivalences, in order to reduce the set of output programs.

By doing this, we realized that even if the set of template is very large, many of the output programs are equivalent. Hence we introduce the idea to only chose a random subset of all the permutations and other possible choices. Most of the time, one of the programs with the smallest size comes out, but this also allows us to try even longest template and therefore in some case to really improve the size of the output program. More details about this work can be found in Raphaël BOST master thesis [Bos12]

Given these different template computation implementations, we discuss the other part we did not formalize in PVS, that is the transformation of programs with function definitions.

6.3 PROGRAM WITH FUNCTIONS TRANSFORMATION

The implementation of the transformation of the program with function is almost straightforward given the description of the algorithm introduced in Chapter 4. The rules $Elim_{fin}$ and $Elim_{fout}$ reuse the template computation and the *Decompose* functions defined for the $Elim_{let}$ rule. However there are new functions that need to be implemented.

6.3.1 $Elim_{fin}$ and $Elim_{fout}$ implementation

One of the functions required for the application of $Elim_{fin}$ and $Elim_{fout}$ transformation is the abstraction of the function calls to compute the *scope program part* as introduced in Definition 4.8. This function has to compute this scope program part and all the associated function calls and the variables that are declared in this scope program part. Thus this function has the following type:

```
val extract_expressions_calls :
  Language.funvar ->
  Language.program ->
   (Language.program list -> Language.program) *
    Language.program list * Language.uvar list
```

And its implementation is quite close to the *Decompose* function, we only have to go deeper in the expression and abstract the function calls. Therefore we have the following predicate:

$$\forall\, \mathsf{p} \in \mathsf{P}^{\top}, \mathsf{extract_expressions_calls\ f\ p} = (\mathsf{scf}, \mathsf{lcall}, \mathsf{lvar}) \Longrightarrow \mathsf{p} = \mathsf{scf}(\mathsf{lcall}) \wedge \forall\, e \in \mathsf{lcall},\ FV(e) \backslash FV(\mathsf{p}) \subseteq \mathsf{lvar}$$

Given this function we are now able to define the functions corresponding to the rules $Elim_{fin}$ and $Elim_{fout}$ as defined in definitions 4.9 and 4.12.

Thus the only step left is to define how, given a transformation item, we find the corresponding definition as introduced in Definition 4.20, in order to apply the corresponding rule. To construct the set of positive expressions that can be used in the template we have

to find the positive expressions we know are valid in the context of this targeted definition. Therefore every time we transform a variable, we record the pair (new variable defined, square roots of the template) in a list. This way, for the transformation of a given definition, we only consider the positive expressions that only use variables that are free in the body of the targeted definition.

Therefore, for the use of the $Elim_{fin}$ rule in a program p on the definition of f we define the following function:

```
let rec elim_fun_in_top f p lposvar currposvar n =
match p with
  | Letin(v,e1,e2) ->
    let nposvar =
      try assoc v lposvar with
        | _ -> []
    in
    let k,np =
     elim_fun_in_top f e2 lposvar (conc_no_repeat nposvar currposvar) n
    in
     (k,Letin(v,e1,np))
```

Where lposvar is the global list of possible positive expressions and currposvar the one that will be used in the $Elim_{fin}$ rule. k is only an index for the fresh name generation. The rest of the function is straightforward:

```
  | Dfun(v,(lx,tx),t,b,sc) when v = f ->
    let (nlx,ntx,nst,nb,nsc,k) =
     elim_funin f lx tx t b sc currposvar n
    in
     k,Dfun(v,(nlx,ntx),nst,nb,nsc)
  | Dfun(v,lv,t,b,sc) ->
    let k,np = elim_fun_in_top f sc lposvar currposvar n in
    k,Dfun(v,lv,t,b,np)
```

We define similar functions for the $Elim_{fout}$ and $Elim_{let}$ transformations .

These functions being defined, the only step left is the construction of the dependency graph as introduced in Definition 4.18 in order to apply the transformation in the right order and finally the implementation of the global definition transformation algorithm introduced in Definition 4.25. The implementation of these functions following directly the formal definition presented in Section 4, we will not discuss their implementation any further.

Conclusion We now have a function in OCaml that is able to transform any normalized program defined in the abstract syntax introduced at the beginning of this chapter. However in order to have a practical tool that is able to transform a real program we will have to build interfaces for the corresponding languages. This is the purpose of the next chapter that presents how this transformation has been used to transform programs from an home-made language but also PVS expressions and specifications and inputs of the Yices solver.

7 INTERFACES

OUR TRANSFORMATION AIMS AT PROCESSING REAL EMBEDDED PROGRAMS. Therefore we have to be able to transform real programs, not just toy examples. The motivation of this work comes from the development of highly secure embedded systems for aeronautics, in particular the ACCoRD system developed in the Formal Methods groups at NASA Langley. This system includes conflict detection and resolution algorithms [NMD12] that use square roots and divisions.

Another application of the square root and division elimination is proof automation. Indeed proof verification systems such as PVS, COQ [Cdt] or HOL [NWP02] includes proofs strategies that enables the user to deal with arithmetic problems automatically. However most of these techniques such as the use of SMT solvers [BT07, DdM06a, dMB08] or quantifier elimination [Col76] do not handle all arithmetic operations, in particular division and square root. Being able to transform any goal or hypothesis containing square roots or divisions into an equivalent one that is free of them would allow the use of arithmetic decision procedures to resolve the current goal.

In this chapter we present some concrete transformations using both the OCaml and the PVS implementation of the algorithm. In a first step, we present the language that can be processed by the OCaml implementation, then we describe how we have used a deep embedding and reflection techniques to define a PVS strategy eliminating square roots and divisions in goals and hypothesis. Finally we present the transformation of PVS specifications with the OCaml implementation, generating correctness lemmas, and prove them using the PVS strategy we defined.

7.1 PARSING AND PRINTING

The main language that is used as an input for the program transformation in OCaml is almost that of Definition 4.1, where we use the usual strings for variables names. We also wrote an interpreter for this language in OCaml, that implements the semantics introduced in Definition 3.4 and completed in Definition 4.2 in order to test our transformation in the first steps of the implementation.

We have a parser for this language implemented with ocamlyacc (see the OCaml manual [LDF+12]). It produces the corresponding program in the program type introduced in

Definition 6.1. We use the parenthesis as delimiters for the precedences of the arithmetic and Boolean infix operators. However, in order to minimize the number of parenthesis that have to be used we also implemented the usual precedences of the arithmetic and Boolean operators (as in the OCaml language) and use left associativity. We overload the $-$ operator to use it both as a unary and binary operator.

EXAMPLE 7.1 (Operator precedences). The following expressions:

```
x + y + z
fst x - y
z - x * y  > n
a || n < x && b
```

are parsed the following way:

```
BOp(Plus,BOp(Plus,Value "x",Value "y"),Value "z")
BOp(Plus,Fst(Value "x"),Value "y")
BOp(Gt,BOp(Plus,Value "z",
                UOp(Umin,BOp(Times,Value "x",Value "y"))),
        Value "n")
BOp(Or,Value "a",BOp(And,BOp(Lt,Value "n",Value "y"),Value "z"))
```

The only other modification of the language from Definition 4.1 is that the if then else constructor has to end with an fi token. Therefore the implementation of the parser is quite straightforward.

We also wrote a printer for this language to be able to evaluate the size of the produced program. As for the parser we wanted to avoid redundant parentheses, therefore this printer only prints parenthesis when they are necessary. A common way to handle this is by associating to every operator a precedence level. Given an expression e_1 op e_2 we only print parenthesis around the expressions e_1 and e_2 when the precedence of their head operator is lower than the one of op. We also try to take care about the indentation in order to produce a program that is readable by a human. Indeed we will use a variation of this printer later for the transformation of PVS specification where we will have to prove some properties on the produced program.

In the following section we present a first utilization of the elimination of square and division as a strategy transforming goal and hypothesis in a PVS proof.

7.2 PVS STRATEGY

As introduced in Chapter 5, the transformation that removes square roots and divisions from programs has been defined and proved correct in PVS. We now aim at using this implementation of the transformation and the proof of the semantics equivalence between the input and the output formulas to define a PVS strategy [AVM03]. This strategy, `elim-sqrt`, transforms any goal or hypothesis by eliminating square roots and divisions from it *e.g.*,

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

This is realized by doing a deep embedding [WN04] of a fragment of PVS inside PVS

in order to use computational reflection for transformation computation [LC09, Har95b, Rue97, Bou97]. This is a big difference with PVS or Coq `fields` strategies, that are written in the strategy language, since the size of the proof does not depend on the input terms.

7.2.1 Deep embedding

To define such a transformation we use the proven PVS computable specification of the transformation that have been described in Chapter 5. It defined the `elim` function that removes square roots and divisions and preserves the type and the semantics of a program. To use this transformation, we have to transpose a PVS statement into the formalism used for this specification that is the `program` datatype used in Section 5.2. We achieve this transformation by doing a deep embedding of a fragment of PVS inside PVS.

Deep embedding

Given a proof context in PVS, we aim at transforming a statement (either a goal or an hypothesis) into an equivalent one which is free of divisions and square roots. First of all, as we can see in Definition 5.1 the formalism only represents a fragment of PVS, therefore the statement we want to transform has to match this formalism. Given such a statement, we call it `S`, the first step of this embedding is to compute the equivalent `p : program` and the corresponding evaluation environment `env` such that:

```
sem(p,env) = boolv(S)
```

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

EXAMPLE 7.2 (Equivalent program in environment).

```
  |---
{1} sqrt(x`1) > y
```

$\longrightarrow$

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

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

Equivalent `program` computation

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

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

Corresponding evaluation environment

As we can see in Example 7.2, the correspondence between identifiers and variables is not straightforward either. Indeed, we need to build the value corresponding to each identifier. Given an identifier `"X"` and its associated variable `x`, if `x` has a basic type, `number` or `bool`, then the semantics of `value("X")` is `x`, but if `x` is a tuple, then we need to extract its elements and build the corresponding `prog_val`. For example if `x` is a triple of type `[bool,real,real]` then the associated value `prog_val` is:

```
pairv(boolv(x‘1),pairv(numv(x‘2),numv(x‘3))).
```

Given a PVS statement `E`, we are able to compute the corresponding `program p` and environment `env`, such that `E` can be replaced by the semantics of `p`, *i.e.,* `bo(sem(p,env))`. This allows us to work on the `program p` in order to apply the transformation.

7.2.2 Strategy definition

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

Fig. 7.1 describes the main steps of the `elim-sqrt` strategy:

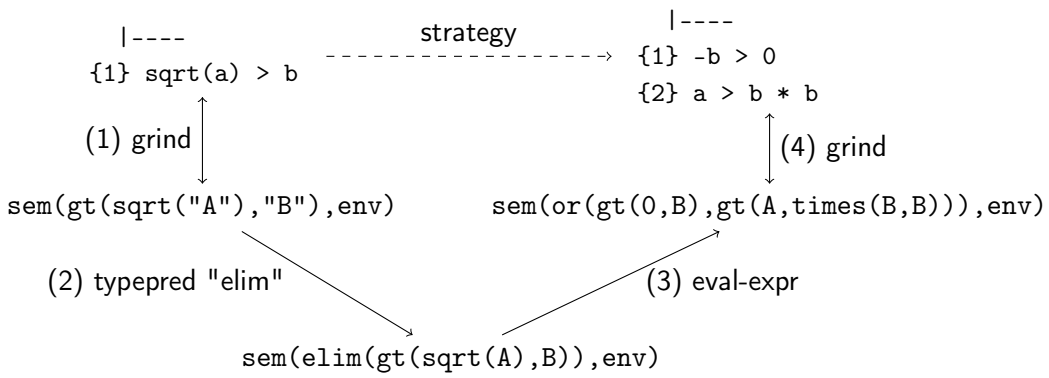


Figure 7.1: `elim-sqrt` strategy outlines

(1) we introduce the equivalent `program` and environment and prove this equivalence using symbolic evaluation with grind

(2) using the following type predicate of `elim` we apply this function to the `program`

```
FORALL (env : program_environment) :
   some?(sem(p,env))) IMPLIES (sem(p,env) = sem(pp,env))
```

(3) we compute the elimination using computational reflection eval-expr

(4) we return into the PVS language itself using symbolic evaluation of the square root and division free `program` semantics

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

From PVS expression to `program datatype`

As mentioned in Section 7.2.1 the transformation is defined using the `program` abstract datatype, the first step of the strategy is therefore to transpose the PVS statement into this datatype. In 7.2.1 we introduced a lisp function that, given a PVS statement, builds the corresponding `program, p` and environment `env`. The first step of the strategy is to introduce this `program` equivalent to `H` using its Boolean semantics `bo(sem(p,env))`. The extraction of the Boolean part of the semantics with `bo` such as the use of the type of the `elim` function will require to prove that `sem(p,env)` does not fail and is a Boolean `prog_val`, this can be done by doing a symbolic evaluation of `sem(p,env)` but this evaluation is not very efficient. Therefore in order to do it only once, we introduce explicitly this hypothesis with the following command:

```
(case "boolv?(sem(p,env)) AND bo(sem(p,env))")
```

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

`elim` function introduction

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

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

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

Computational reflection

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

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

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

Semantics evaluation

From our new square root and division free `program p'` we want to get the corresponding PVS expression. Therefore we have to compute the semantics of this `program`. This is done once again by symbolic evaluation and in the end we get a new PVS statement `H'`, equivalent to `H`, free of square roots and divisions. However in this case we already know that the program does not fail, therefore we are able to write another semantics function, that computes the semantics of programs that do not fail.

```
semantics_opt(p: program, env | some?(semantics(p,env))) :
   RECURSIVE {v : prog_value | val(semantics(p,env)) = v} =
```

This optimized semantics is really straightforward and thus its symbolic evaluation is more efficient. For example, for the projection case, the general semantics have to test that the sub-term semantics does not fail and is a pair:

```
  fst(p1) : LET t = (semantics(p1,env)) IN
   IF some?(t) & pairv?(val(t)) THEN Some(vl(val(t))) ELSE None ENDIF,
```

While the optimized semantics directly extract the first element of the semantics of the argument:

```
  fst(p1) : (vl(semantics_opt(p1,env))) ,
```

Square roots and divisions being eliminated in this hypothesis we can now continue the proof using our favorite arithmetic strategy.

Conclusion

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

This strategy has been tested on various examples, from simple comparisons to more complex statements that embed variable definitions and conditional expressions. The

strategy takes between twenty seconds to few minutes mainly depending on the number of square roots. These results can be explained by the low performances of the PVS symbolic evaluation whereas the transformation itself that uses reflection, is almost instantaneous.

This strategy is also the first step of a larger scale transformation that aims at eliminating square roots and divisions from full PVS specifications. This transformation is presented in the following section.

7.3 PVS THEORY TRANSFORMATION

In this section, we aim at transforming complete PVS theories that are mainly built with PVS function definitions. The PVS specification of the transformation does not handle programs that include function definitions. Therefore we have to use our OCaml implementation in order to transform such programs.

7.3.1 PVS to OCaml

Besides the theorem proving part, such as lemmas and theorem declarations or typing judgments, a PVS specification is basically a list of variable and function definitions. If the features used in these functions and variable definitions are similar to the ones that are in the abstract input language of our transformation, *i.e.,* arithmetic and Boolean operators, projection, variable definitions and conditional expressions, then we should be able to translate these specifications into this input language. However, even if the function definitions are limited to the use of these features, the type system in PVS is much richer than the one in our language. Thus we do not want to write a new PVS parser in order to have these PVS specifications in the abstract syntax used in OCaml.

Thus we first use the PVS parser and the `pvsio` features that enables us to handle input and output of the PVS system. This way we are able to use the lisp parser of PVS, to parse the theory we target. We end up with a lisp structure representing the current PVS specification and it is quite straightforward, as for the strategy case introduced in Section 7.2.1 , to generate the corresponding program in the language that the OCaml implementation can process (let us call it the letf-language). Indeed, while for the strategy we generated the string corresponding to the PVS abstract syntax of the language, this time we simply generate the concrete syntax of the letf-language. We do not detail this translation since it is quite straightforward as in the strategy case.

Main problems in this translation are coming from the different representation of tuples in PVS (*i.e.,* arrays), lisp (*i.e.,* list) and the input language (*i.e.,* binary trees). Another particular case is the type declaration of the function, but in lisp we are able to access the *supertype* of a term, *i.e.,* number or Booleans. This allows us to have the corresponding first order type for the functions arguments and outputs. The last issue if that, unlike the transformation language, PVS definitions do not always have a scope, a specification can simply be a list of definitions without returned expression. In order to translate this in our language we only add as final scope a new fresh variable that does not depends on the previous definitions.

We also have to use the importing (dependencies) of the input program to allow the output of the OCaml transformation to be directly typechecked by PVS, therefore we

also transfer this list and lightly modify the OCaml parser to handle such declarations. Therefore the `pvsio` generates programs in the following way:

EXAMPLE 7.3 (From PVS to OCaml). The following PVS specification:

```
example : THEORY
BEGIN

IMPORTING reals@sqrt, Elim

a : real = sqrt(2)
f(x : real, y,z : nnreal) : real = x*y + sqrt(2+z)
f_test(x1 : real, y1,z1 : nnreal) : bool =
   f(x1,y1,z1) > 0
END example
```

produces this equivalent program in letf-language:

```
example : THEORY

IMPORTING reals@sqrt, Elim

BEGIN
let a = (sqrt 2) in
letf f (x,(y,z)) : (Num*(Num*Num)) -> Num =
   ((x * y) + (sqrt (2 + z))) ;
letf f_test (x1,(y1,z1)) : (Num*(Num*Num)) -> Bo =
   (f((x1,(y1,z1))) > 0) ;
Token_pvs
END example
```

Token_pvs being the free variable representing the final scope.

This program is never seen in practice since the only goal of producing such a program is to link it with the transformation. Therefore the lisp printer is really straightforward and does not handle operation precedences for example but uses redundant parenthesis.

Given a program in this form, we are now able to parse it with the OCaml implementation and thus eliminating square roots and divisions from it. The only step left is to generate the new PVS code, given a transformed program in the abstract syntax of the transformation. This generation is really straightforward since this language is almost a subset of the PVS language. Once again the only troubles come from the tuple representation since PVS does not allow multi-level of matching. Thus

```
let (a,(b,c)) = if t then (x1,(x2,x3)) else (y1,(y2,y3)) fi in ...
```

is printed as

```
LET (a,b,c) = IF t THEN (x1,x2,x3) ELSE (y1,y2,y3) ENDIF IN ...
```

However, since the specification in PVS does not always return a value, that is required to define the semantics of a program, one can wonder how we can now specify that the transformed program is equivalent to the input one. This is presented in the following section.

7.3.2 Specification with subtyping

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 is a function then it is never called in the specification we target and thus the input type does not change either:

EXAMPLE 7.4 (Boolean function transformation). The following function:

```
f_test(x,y : real, z : nnreal) : bool =
  LET sq = y*sqrt(z) IN
   x + sq > 0
```

is transformed into

```
f_test_e(x, y, z : real) : bool =
  LET sq_1 = y
  IN
  LET (at_p, at_r, at_rel, at_neq) =
   (sq_1 > 0 ,x > 0 ,sq_1 * sq_1 * z - x * x > 0 ,
      sq_1 * sq_1 * z - x * x /= 0)
  IN at_p AND at_r OR at_p AND at_rel OR at_r AND NOT at_rel AND at_neq
```

that have the same semantics.

However transforming the functions that return numerical values changes the types and thus the semantics can not be compared. Moreover, when these functions are used into other functions the relation with the functions of the input programs is completely lost.

We decided to use the sub-typing features of PVS to add a typing predicate to the transformed functions, it allows us to specify the behavior of the transformed functions relatively to the input functions. In fact this predicate only represents the template that have been used to transform the input and the output of the function. In order to realize such a transformation, we add another constructor for functions in the OCaml abstract syntax that enables us to add a predicate to the returned type of the transformed function:

```
    | Dfunt of uvar*(var*types)*(var*types*program)*program*program
```

The program added to the output type is supposed to be a Boolean expression that represents the specification of the new function. This constructor has the following meaning:

```
Dfunt(f,(x,Ti),(y,To,P),body,scope)
```

represents the following PVS definition:

```
f (x : Ti) : { y : To |  P } = body
scope
```

Using this predicate, we are able to transform a function and specify the behavior of the transformed function using the input function, *e.g.*,

EXAMPLE 7.5 (Function subtype specification). The transformation of the following function:

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

res : bool = f(a + b * sqrt(c)) > 0
```

is specified in the type of the transformed function:

```
f_e(t_1, t_2 : real) :
  {f_n_1, f_n_2, f_d_1, f_d_2 : real |
     (f_n_1 + f_n_2 * sqrt(c)) / (f_d_1 + f_d_2 * sqrt(c))
       = f(t_1 + t_2 * sqrt(c)) } =
 (1 + t_1 ,t_2 ,-1 + t_1 ,t_2)

res : bool =
  LET (f_n_1, f_n_2, f_d_1, f_d_2) =
        f_e((a ,b))
  IN ...
```

Therefore we can give the new transformation rules for the functions. We use new names for the functions in order to use the input program as an import in the output program to specify the transformed function with functions defined in this input program:

DEFINITION 7.1 (Function input transformation with subtyping). We use the template to specify the new entry:

$$\begin{array}{l}\mathsf{letf\ f\ x}:\ A \to B = \mathsf{b};\\ scf(\mathsf{f}(Pp_1(T_i\sigma_{1,1},...,T_i\sigma_{1,m_1})),...,\mathsf{f}(Pp_n(T_i\sigma_{n,1},...,T_i\sigma_{n,m_n})))\\ \qquad\longrightarrow\\ \mathsf{letf\ f_e}\ var(\sigma_{1,1})\ :\ A' \to \boldsymbol{\{y : B \mid y = \mathsf{f}(T_i)\}} = \mathsf{b}[\ \mathsf{x} \mapsto T_i\ ];\\ scf(\mathsf{f_e}(Pp_1(arg(\sigma_{1,1}),...,arg(\sigma_{1,m_1}))),...,\mathsf{f_e}(Pp_n(arg(\sigma_{n,1}),...,arg(\sigma_{n,m_n}))))\end{array}$$

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

DEFINITION 7.2 (Function output transformation with subtype). We use the output template to specify the relation with the input function:

$$\begin{array}{l}\mathsf{letf\ f_e\ x}:\ A \to \{y : B \mid y = \mathsf{f}(T_i)\} = Pp(T_o\sigma_1,...,T_o\sigma_n);\\ scf(\mathsf{f}(\mathsf{a}_1),...,\mathsf{f}(\mathsf{a}_m))\\ \qquad\longrightarrow\\ \mathsf{letf\ f_e\ x}:\ A \to \boldsymbol{\{var(\sigma_1) : B' \mid T_o = \mathsf{f}(T_i)\}} = Pp(arg(\sigma_1),...,arg(\sigma_n));\\ scf(\mathsf{let}\ var(\sigma_1) = \mathsf{f_e}(\mathsf{a}_1)\ \mathsf{in}\ T_o,...,\mathsf{let}\ var(\sigma_1) = \mathsf{f_e}(\mathsf{a}_m)\ \mathsf{in}\ T_o)\end{array}$$

where B' is the type of the $arg(\sigma_i)$

We end up with new function definitions that do not contain any square roots or divisions (except in the typing predicate). The typing predicate associated to Boolean functions states that the output of the new function is equal to the output of input function, the template of a Boolean being a single variable.

This way the functions in the transformed program are completely specified with respect to the input functions, the only step left being to prove the type checking conditions resulting from these predicates.

7.3.3 Proving the equivalence

Adding predicates to the type of the produced function imply to prove some type checking conditions (see Section 5.1). The TCC generation by PVS already decompose the functions bodies such that it generates one TCC per case.

EXAMPLE 7.6 (TCC decomposition). Assume we have the following functions:

```
f(x1,y1 : posreal) : real = x1/y1

g(t : bool ,x,y : posreal) : real =
  IF t THEN f(x,(y + 1)) ELSE sqrt(x) + y ENDIF
```

that are transformed into:

```
 f_e(x1, y1 : real) : {f_n, f_d : real | f_n / f_d = f((x1, y1))} =
      (x1, y1)

 g_e(t : bool, x, y : real) : {g_n_1, g_n_2, g_d, sq_0 : real |
   (g_n_1 + g_n_2 * sqrt(sq_0)) / g_d = g((t, x, y))} =
      IF t
      THEN
       LET (f_n, f_d) =
            f_e((x, y + 1))
       IN (f_n, 0, f_d, 0)
      ELSE (y, 1, 1, x)
      ENDIF
```

Then PVS generates for g_e the following TCCS:

```
g_e_TCC2: OBLIGATION
 FORALL (t: bool, x, y: real):
   t IMPLIES
    (FORALL (f_n: real, f_d: real):
       f_d = f_e(x, y + 1)`2 AND f_n = f_e(x, y + 1)`1 IMPLIES
        (f_n + 0 * sqrt(0)) / f_d = g(t, x, y));

g_e_TCC3: OBLIGATION
 FORALL (t: bool, x, y: real):
   NOT t IMPLIES (y + 1 * sqrt(x)) / 1 = g(t, x, y);
```

The different cases corresponding to the conditional constructor (if then else) are already decomposed, and 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 inlining the type of the other functions (*e.g.*, `f_e`) the current function is depending on, we can use the strategy that we introduced in section 7.2 to prove such equalities. The proof of `g_e_TCC2` is done by the following strategy:

EXAMPLE 7.7 (Proof of equivalence predicates). The `g_e_TCC2` obligation requires to prove that `(f_n + 0 * sqrt(0)) / f_d = g(t, x, y))`

i) By expanding g knowing that `t` is true, the new goal is the following formula:
   `(f_n + 0 * sqrt(0)) / f_d = f(x, 1 + y)`

ii) Then by adding the type predicate of `f_e` we have the following hypothesis
   `f_n / f_d = f(x, 1 + y)`

iii) 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 `(assert)` strategy but more complicated predicates require to invoke the subtyping of many other functions and use the `(elim-sqrt)` strategy, in particular when square roots are involved in Boolean expressions. However this is the general scheme of the proof of the equivalences between the input and the output functions:

- First, expand the input function that corresponds to the current TCC
- Then, introduce all the typing predicates corresponding to the functions that are called in the new expression
- Prove the Boolean equivalence to reduce the expanded input function to the case corresponding to the current TCC of the output one
- Prove the expression equality with either native arithmetic strategies or the `(elim-sqrt)` one for more complicated cases

Therefore, even if the OCaml transformation is not proved 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 we defined, that corresponds to the transformation of Boolean expressions that is the core of our transformation.

In the following section, we present how we are now able to reduce the size of the output programs by defining comparisons operators as function.

7.3.4 From comparison operator to function

The transformation of functions using anti-unification is efficient regarding the size of the produced code and the equivalence lemmas are relatively easy to prove using the strategy described in section 7.3.3. Therefore we decided to try to use this function transformation in order to factorize the large Boolean expressions produced by the elimination of square roots in Boolean. This is done by first replacing the comparisons operators in the input program by functions that have the same semantics before applying the transformation,*e.g.*,

EXAMPLE 7.8 (Comparisons as functions). The following program :

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

g(x,y, z : posreal) : real =
  IF z + sqrt(y + x) > 0 OR f(y,z) THEN y ELSE sqrt(x) + y ENDIF
```

Is first transformed into the following program:

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

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

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

g(x,y, z : posreal) : real =
  IF gt2(z + sqrt(y + x),0) OR f(y,z) THEN y ELSE sqrt(x) + y ENDIF
```

And only then the elimination of square roots and division is applied.

Since the Boolean function outputs do not create any dependency such transformation do not create cycles in the dependency graph. This allows us to transform such programs, the Boolean formulas from the input program are almost the same, the $>$ operator being replaced by a gt function. But then the transformed gt functions that have the same specification are re-factorized in one unique function corresponding to the template. However, having different functions for different templates is still useful since we want to avoid computation of large expressions as much as possible to limit the size of the fixed point numbers that have to be used for exact computation. This is why we declared different functions for the different occurrences of the comparisons operators in the first place.

EXAMPLE 7.9 (Transformation with comparisons function factorization). The program is transformed, the two gt transformations using the same template, we only use one of them:

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

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

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

Therefore this transformation of Boolean expressions with functions requires only an automatic pre and post process of the transformation:

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

The new comparisons functions can also be generated in a separate file that is a dependency of the transformed program. In this way the transformed program has exactly the same structure as the input one. In order to illustrate this transformation on a real example, we present in Chapter 8 the transformation of a real PVS program for conflict detection in two dimensions.

7.4 Yices

We also implemented an interface for the Yices solver. Yices is an SMT solver developed by the SRI Laboratory [DdM06b]. This solver does not handle division or square root. Therefore we implemented in OCaml a parser for a subset of the Yices language corresponding to the Boolean expressions built with comparisons between arithmetic expression extended with square roots and divisions. The reverse pretty printer produces Yices files from the transformed program where square roots and divisions have been eliminated. This way we are able to pre-process Yices programs before using the solver to allow it to handle formulas using the square root and division operations.

8 APPLICATIONS

A COMPLETE PVS SPECIFICATIONS CAN BE PROCESSED by the OCaml implementation of the transformation linked with `pvsio`. In this chapter we present the transformation of a conflict detection algorithm, namely cd2d that is developed by NASA in the ACCoRD framework. This algorithm aims at detecting loss of separation between two aircrafts in a two-dimensional space. An analysis of numerical stability of this program has been presented in [GMKC13], the algorithm is described in this paper but we recall its main characteristics.

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

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

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

And a conflict occurs when a loss of separation is going to occur before the end of a lookahead time T:

$$conflict?(s, v) \iff \exists\, t \leq T,\ loss?(s + t.v)$$

Where $+$ and $.$ are the usual addition and constant multiplication in $\mathcal{R}^2$. A function named `detection_2D` is defined cd2d, it computes the interval of time where the loss of separation occurs. Indeed the following predicates have been proved in PVS:

```
    detection_2D_correct : THEOREM
    LET (tin,tout) = detection_2D(s,v) IN
      tin < t AND t < tout IMPLIES horizontal_los?(s+t*v)

  detection_2D_complete : THEOREM FORALL (s,v)
    LET (tin,tout) = detection_2D(s,v) IN
      horizontal_los?(s+t*v) IMPLIES
        tin <= t AND t <= tout AND tin < tout

  conflict_detection_2D : THEOREM FORALL (s,v)
```

```
    LET (tin,tout) = detection_2D(s,v) IN
      conflict_2D?(s,v) IFF tin < tout
```

The detection_2D is defined in the following specification. In order to be able to transform the program we had to remove all the lemmas and theorems from the original program.

EXAMPLE 8.1 (Conflict detection algorithm). This PVS specification of the conflict detection algorithm is defined in the ACCoRD system:

```
cd2d  : THEORY
BEGIN

  IMPORTING reals@sqrt, Elim

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

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

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

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

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

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

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

  detection_2D(s,v : [real,real],B,T,D,Entry,Exit : real) : [real,real] =
    IF zero_vect2?(v) AND horizontal_los?(s,D) THEN
      (B,T)
    ELSIF Delta(s,v,D) > 0 THEN
      LET tin  = Theta_D(s,v,Entry,D),
          tout = Theta_D(s,v,Exit,D) IN
        (minmax(tin,B,T),maxmin(tout,T,B))
    ELSE
      (B,B)
    ENDIF

  detect?(st, vt : [real,real], Bt, Tt, Dt, Entryt, Exitt : real) : bool =
    LET (tint,toutt) = detection_2D(st,vt,Bt,Tt,Dt,Entryt,Exitt) IN
      tint < toutt

END cd2d
```

The only square roots and divisions of this program are in the `Theta_D` function, however in the body of the `detection_2D` function, the result of `Theta_D` is then used by the `minmax` and `maxmin` functions. Therefore square roots and divisions will propagate to these other functions during the transformation.

The OCaml implementation of the transformation of programs with function definitions introduced in Chapter 6 and with the subtype predicate generation introduced in Section 7.3 transforms the PVS program from Example 8.1 into the following one:

EXAMPLE 8.2 (Transformed cd2d). The transformation generates the following program, the comparison operators being defined in a separate file, namely cd2d_operators.pvs, as introduced in Section 7.3.4:

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

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

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

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

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

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

```
            ENDIF
        IN
        IF gt0_e((mini_n_1, mini_n_2, mini_d, maxv, sq_2))
        THEN (mini_n_1, mini_n_2, mini_d, sq_2)
        ELSE (maxv, 0, 1, 0)
        ENDIF

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

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

    detection_2D_e(s, v : [real,real], B, T, D, Entry, Exit : real) :
      {detection_2D1_n_1, detection_2D1_n_2,
        detection_2D1_d, detection_2D2_n_1,
         detection_2D2_n_2, detection_2D2_d, sq_7, sq_8 : real |
            ((detection_2D1_n_1 + detection_2D1_n_2 * sqrt(sq_8)) /
                detection_2D1_d,
             (detection_2D2_n_1 + detection_2D2_n_2 * sqrt(sq_7)) /
                detection_2D2_d) =
                   detection_2D((s, v, B, T, D, Entry, Exit))} =
     IF zero_vect2?_e(v) AND horizontal_los?_e((s, D))
     THEN (B, 0, 1, T, 0, 1, 0, 0)
     ELSE
     IF Delta_e((s, v, D)) > 0
     THEN
     LET (Theta_D_n_1, Theta_D_n_2, Theta_D_d, sq_0) =
          Theta_D_e((s, v, Entry, D))
     IN
     LET (new_Theta_D_n_1, new_Theta_D_n_2, new_Theta_D_d, new_sq_0) =
          Theta_D_e((s, v, Exit, D))
     IN
     LET (maxmin_n_1, maxmin_n_2, maxmin_d, sq_3) =
          maxmin_e((new_Theta_D_n_1, new_Theta_D_n_2,
                    new_Theta_D_d, T, B, new_sq_0))
     IN
     LET (minmax_n_1, minmax_n_2, minmax_d, sq_6) =
          minmax_e((Theta_D_n_1, Theta_D_n_2, Theta_D_d, B, T, sq_0))
     IN (minmax_n_1, minmax_n_2, minmax_d, maxmin_n_1,
           maxmin_n_2, maxmin_d, sq_3, sq_6)
     ELSE (B, 0, 1, B, 0, 1, 0, 0)
```

```
99      ENDIF
        ENDIF

      detect?_e(st, vt : [real,real], Bt, Tt, Dt, Entryt, Exitt : real) :
        {res : bool | res = detect?((st, vt, Bt, Tt, Dt, Entryt, Exitt))} =
104   LET (detection_2D1_n_1, detection_2D1_n_2, detection_2D1_d,
        detection_2D2_n_1, detection_2D2_n_2, detection_2D2_d, sq_7, sq_8) =
                detection_2D_e((st, vt, Bt, Tt, Dt, Entryt, Exitt))
      IN lt3_e((detection_2D1_n_1, detection_2D1_n_2, detection_2D1_d,
           detection_2D2_n_1, detection_2D2_n_2, detection_2D2_d, sq_8, sq_7))
109 END cd2d_elim
```

As one can notice, the number of lines in the output program is more than twice the length of the input one. However this is mainly due to the length of the sub-typing predicates associated to the transformed functions.

The comparisons are replaced by function that are defined in the `cd2d_operators` theory, such as `gt_0_e` or `lt_1_e` used in the `minmax` and `maxmin` function. And their use is factorized since both `minmax_e` and `maxmin_e` use the same comparison functions.

This transformed program is therefore equivalent to the input one according to the type predicates embedded in the type of the functions and it does not use square roots or divisions anymore except in these predicates. Therefore, being able to construct an exact implementation of real numbers computations with addition, subtraction and multiplication would enable an exact execution of this program.

CONCLUSION

THIS THESIS has presented a way to deal with the usual inexactness introduced by the computation over real numbers. This is achieved with a program transformation that removes square roots and divisions from straight lines programs. This transformation has been implemented in OCamL and partially proven in the PVS proof assistant. As presented in Chapter 8, the OCaml/`pvsio` transformation handles the transformation of PVS specifications. The proof of the sub-typing predicates generated by this transformation is quite easy in PVS using the strategy we presented in Chapter 7. Thus we provide a transformation of PVS programs that enables the user to certify with minimum efforts the equivalence between the input and the output program. But the work presented in this thesis can be pushed further in many directions.

Exact computation with $+, -, \times$ Even after the transformation, the exactness of the computations still relies on an exact implementation of the addition, subtraction and multiplication in an embedded system. In Section 3.4.1 we outlined an analysis of the program in order to implement these exact operations. However, the real implementation, that involves compilation issues was not in the scope of this thesis and still remains to be done in order to have a real embedded system computing exactly.

Loops or recursion Chapter 4 introduced an extension of the first program transformation by adding the function definitions to the language processed by the transformation. However, if targeting a real Turing-complete language is not in the scope yet, the unbounded behaviors associated to such languages not being allowed in embedded systems, features such as bounded loops are conceivable. In Section 4.5, we briefly introduced an idea to extend the transformation to loops that might be worth to be studied deeper.

Certified transformation The transformation is only partially proved in PVS since neither the anti-unification nor the function transformation are specified in this system. As shown in Chapter 8 we are able to generate and prove lemmas associated to the transformation of a PVS specification to certify that the transformed program is correct. It could

be worth to have a automated certified transformation, either by defining a strategy that always prove the generated type-checking conditions associated to the sup-typing predicates stating the correctness of the transformed program. But the best way would be to complete the specification and the proof of the transformation in the PVS system so that the transformation would be certified itself instead of certifying the programs it generates. Therefore such a transformation would allow us to have a certified transformation for programing languages that do not rely on a particular proof system to prove properties of these programs.

Other operations Last but not least, the transformation scheme introduced in this thesis for the elimination of square roots and divisions might be reused to eliminate other operations. In particular, the extension of the transformation to eliminate the cubic extractions seems a reasonable goal, but this must be pushed further.

BIBLIOGRAPHY

[AEMO08] María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. A modular equational generalization algorithm. In Michael Hanus, editor, *LOPSTR*, volume 5438 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2008. 19, 35

[AVM03] Myla Archer, Ben Di Vito, and César Muñoz. Developing user strategies in PVS: A tutorial. In *Proceedings of Design and Application of Strategies/-Tactics in Higher Order Logics STRATA'03*, NASA/CP-2003-212448, NASA LaRC,Hampton VA 23681-2199, USA, September 2003. 118

[BBF+00] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert de Simone. ESTEREL: a formal method applied to avionic software development. *Sci. Comput. Program.*, 36(1):5–25, 2000. 2

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In Ron Cytron and Rajiv Gupta, editors, *PLDI*, pages 196–207. ACM, 2003. 2

[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. 2

[BKZ09] Peter E. Bulychev, Egor V. Kostylev, and Vladimir A. Zakharov. Anti-unification algorithms and their applications in program analysis. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009. 19

[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. 2

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998. 51

[Bos03] Alin Bostan. *Algorithmique efficace pour des opérations de base en Calcul formel*. PhD thesis, Ecole polytechnique, 2003. 2

[Bos12] Raphaël Bost. *Nombres réels et transformation de programmes*. Master thesis, Ecole polytechnique, 2012. 111, 115

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997. 118

[Bro03] Christopher W. Brown. An overview of QEPCAD B: a tool for real quantifier elimination and formula simplification. *Journal of Japan Society for Symbolic and Algebraic Computation*, 10(1):13–22, 2003. 13

[BS01] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001. 19

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007. 117

[Cdt] The Coq development team. *The Coq proof assistant reference manual*. 117

[CGL12] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, 2012. 49

[CGLN11] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navid-Pour. Proving programs robust. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT FSE*, pages 102–112. ACM, 2011. 49

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag. 2

[CM95] Victor A. Carreño and Paul S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS, 1995. 2

[CM10] Cyril Cohen and Assia Mahboubi. A formal quantifier elimination for algebraically closed fields. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *AISC/MKM/Calculemus*, volume 6167 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2010. 13

[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. 2

[Coh69] Paul J. Cohen. Decision procedures for real and p-adic fields. *Communications on pure and applied mathematics*, 22(2):131–151, 1969. 13

[Coh12a] 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. 2

[Coh12b] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, Ecole polytechnique, 2012. 2

[Col76] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis. *SIGSAM Bull.*, 10:10–12, February 1976. 13, 117

[DD03] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: on the impact of the CPS transformation. *J. Funct. Program.*, 13(5):867–904, 2003. 3

[DdM06a] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006. 117

[DdM06b] Bruno Dutertre and Leonardo Mendonça de Moura. The Yices SMT solver. Technical report, SRI International, 2006. 130

[DGL04] Pietro Di Gianantonio and Pier Luca Lanzi. Lazy algorithms for exact real arithmetic. *Electron. Notes Theor. Comput. Sci.*, 104:113–128, November 2004. 2

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. 117

[DMM05] Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In *IEEE Symposium on Computer Arithmetic*, pages 188–195, 2005. 2

[DRT01] Marc Daumas, Laurence Rideau, and Laurent Thery. A Generic Library for Floating-Point Numbers and Its Application to Exact Computing. In *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 169–184, Edinburgh, Royaume-Uni, 2001. Springer Berlin / Heidelberg. 2

[DS96a] Andreas Dolzmann and Thomas Sturm. Redlog computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31:2–9, 1996. 13

[DS96b] Andreas Dolzmann and Thomas Sturm. *Redlog User Manual*, 1996. 13

[GMKC13] Alwyn Goodloe, César Muñoz, Florent Kirchner, and Loïc Correnson. Verification of numerical programs: From real numbers to floating point numbers. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *Proceedings of the 5th NASA Formal Methods Symposium (NFM 2013)*, volume 7871 of *Lecture Notes in Computer Science*, pages 441–446, Moffett Field, CA, May 2013. 131

[GMP02] Eric Goubault, Matthieu Martel, and Sylvie Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In Daniel Le Métayer, editor, *ESOP*, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212. Springer, 2002. 1, 2

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991. 1

[GP11] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2011. 1

[Har95a] 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. 2

[Har95b] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz`. 118

[Har97] John Harrison. Floating point verification in HOL Light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997. 2

[Hue76] Gérard Huet. *Resolution d'Equations dans les langages d'ordre 1, 2,..., ω*. PhD thesis, Université de Paris VII, 1976. 19

[IEE85] IEEE. *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985. 1, 48

[Joh63] Selmer M. Johnson. *Generation of Permutations by Adjacent Transposition*. Memorandum (Rand Corporation). Rand Corporation, 1963. 114

[KLV11] Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. In Manfred Schmidt-Schauß, editor, *RTA*, volume 10 of *LIPIcs*, pages 219–234. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011. 19, 35

[KS11] Robbert Krebbers and Bas Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1), 2011. 2

[Lan94] Serge Lang. *Algebraic Number Theory*. Graduate Texts in Mathematics. Springer, 1994. 32

[LC09] Stéphane Lescuyer and Sylvain Conchon. Improving Coq propositional reasoning using a lazy CNF conversion scheme. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCoS*, volume 5749 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2009. 118

[LDF+12] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 4.00): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, July 2012. 107, 117

[LMM88] Jean-Louis Lassez, Michael J. Maher, and Kim Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming.*, pages 587–625. Morgan Kaufmann, 1988. 19

[Mar07] Matthieu Martel. Semantics-based transformation of arithmetic expressions. In *SAS*, pages 298–314, 2007. 3

[Mar09] Matthieu Martel. Program transformation for numerical precision. In *PEPM*, pages 101–110, 2009. 3

[MBdD+10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9. 1

[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. 2, 41

[Min95] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS, 1995. 2

[Min04] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In David A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004. 2

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008. 1

[Moo95] Ramon E. Moore. *Methods and applications of interval analysis*. SIAM studies in applied mathematics. SIAM, 1995. 2

[Nap14] John Napier. *Mirifici logarithmorum canonis descriptio*. Hart, Edimburgh, 1614. English translation by Edward Wright: *A description of the admirable table of logarithmes*, London, 1616.

[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. 4

[Ner13] Pierre Neron. Square root and division elimination in pvs. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP*, volume 7998 of *Lecture Notes in Computer Science*, pages 457–462. Springer, 2013. 4

[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. 2, 41, 117

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, Berlin, Heidelberg, 2002. 117

[Obe07] Erick L. Oberstar. Fixed-point representation & fractional math. *Oberstar Consulting, revision*, 1, 2007. 2

[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. 2

[Oka00] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 131–136, New York, NY, USA, 2000. ACM. 28

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer. 91

[OSW05] Cosmin Oancea, Clare So, and Stephen M. Watt. Generalization in Maple, 2005. 19

[Pfe91] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991. 19

[Plo70] Gordon D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970. 19

[Pot89] Loïc Pottier. Generalisation de termes en theorie equationnelle. Cas associatif-commutatif. Rapport de recherche RR-1056, INRIA, 1989. 19

[PS83] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, September 1983. 3

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine intelligence*, 5(1):135–151, 1970. 19

[Rob65] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. 19

[Rue97] Harald Rueß. Computational reflection in the calculus of constructions and its application to theorem proving. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, Lecture Notes in Computer Science, Nancy, France, April 1997. Springer. 118

[Sei54] Abraham Seidenberg. A new decision method for elementary algebra. *The Annals of Mathematics*, 60(2):365–374, 1954. 13

[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. 2

[Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. Univ. of California Press, 2nd edition, 1951. 13

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of the London Mathematical Society*, 42:230–265, 1936. 1

[Vui87] Jean Vuillemin. Exact real arithmetic with continued fractions. Rapport de recherche RR-760, INRIA, 1987. 2

[Wei94] Volker Weispfenning. Quantifier elimination for real algebra - the cubic case. In *ISSAC*, pages 258–263, 1994. 13

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, 8(2):85–101, 1997. 13

[Wie80] Edwin Wiedmer. Computing with Infinite Objects. *Theoretical Computer Science*, 10:133–155, 1980. 2

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2004. 118

Résumé

Cette thèse présente un algorithme qui élimine les racines carrées et les divisions dans des programmes sans boucles, utilisés dans des systèmes embarqués, tout en préservant la sémantique. L'élimination de ces opérations permet d'éviter les erreurs d'arrondis à l'exécution, ces erreurs d'arrondis pouvant entraîner un comportement complètement inattendu de la part du programme. Cette transformation respecte les contraintes du code embarqué, en particulier la nécessité pour le programme produit de s'exécuter en mémoire fixe. Cette transformation utilise deux algorithmes fondamentaux développés dans cette thèse. Le premier permet d'éliminer les racines carrées et les divisions des expressions booléennes contenant des comparaisons d'expressions arithmétiques. Le second est un algorithme qui résout un problème d'anti-unification particulier, que nous appelons *anti-unification contrainte*. Cette transformation de programme est définie et prouvée dans l'assistant de preuves PVS. Elle est aussi implantée comme une stratégie de ce système. L'anti-unification contrainte est aussi utilisée pour étendre la transformation à des programmes contenant des fonctions. Elle permet ainsi d'éliminer les racines carrées et les divisions de spécifications écrites en PVS. La robustesse de cette méthode est mise en valeur par un exemple conséquent: l'élimination des racines carrées et des divisions dans un programme de détection des conflits aériens.

Abstract

This thesis presents an algorithm that eliminates square root and division operations in some straight-line programs used in embedded systems while preserving the semantics. Eliminating these two operations allows to avoid errors at runtime due to rounding. These errors can lead to a completely unexpected behavior from the program. This transformation respects the constraints of embedded systems, such as the need for the program to be executed in a fixed size memory. The transformation uses two fundamental algorithms developed in this thesis. The first one allows to eliminate square roots and divisions from Boolean expressions built with comparisons of arithmetic expressions. The second one is an algorithm that solves a particular anti-unification problem, that we call *constrained anti-unification*. This program transformation is defined and proven in the PVS proof assistant. It is also implemented as a strategy for this system. Constrained anti-unification is also used to extend this transformation to programs containing functions. It allows to eliminate square roots and divisions from PVS specifications. Robustness of this method is highlighted by a major example: the elimination of square roots and divisions in a conflict detection algorithm used in aeronautics.