Constrained Anti-Unification for Program Transformation

Pierre Neron, Raphaël Bost

École polytechnique - Inria

Abstract

In this paper a new kind of anti-unification problems is introduced, where the substitutions anti-unifying two terms are constrained to be in a sub-language of that used to express the terms themselves. We give an algorithm for such constrained anti-unification modulo some equational theory. This algorithms uses a representation of terms as directed acyclic graphs. It is then used in a program transformation algorithm that eliminates some function symbols from programs.

Keywords and phrases Anti-unification, Equational Theory, Directed Acyclic Graphs, Program Transformation

1 Introduction

The anti-unification problem has been introduced in 1970 independently by J.C. Reynolds [14] and G.D. Plotkin [12]. 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 (generalization or anti-unifier) of two terms such that substitutions applied to this template produce the input terms *i.e.,* compute a term t such that $\exists \sigma_1, \sigma_2,\ t_1 = t\sigma_1 \textit{ and } t_2 = t\sigma_2$. The unification problem has been widely studied in the fields of proof theory and rewriting (see [2] for a survey) and therefore many papers have presented efficient solutions where equality may be considered modulo various equational theories (*e.g.,* [15, 4]).

Algorithms for first order anti-unification have been introduced in [14, 12, 4] and one for higher order anti-unification, in the Calculus of Construction, is presented in [11]. These algorithms are used, for instance, for proof generalization and, more recently, termination [1] using generalization modulo some equational theory [13]. Anti-unification has also been used to find general properties or solutions of algebraic expressions [7, 10] or to detect code duplication [3, 5]. All of these applications focus on computing the most specific (or least general) template which is the dual of the most general unifier.

Our goal is slightly different, 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 strict subset of that of the input terms. This particular notion of anti-unification is tailored to be used in a program transformation algorithm introduced in [9]. This transformation eliminates square roots and divisions in some embedded straight line programs (programs without loops), used in aeronautics [8] in order to compute exactly with real numbers. This leads us to transform the variable definitions that contain these symbols in order to eliminate these operations from their bodies. Given a variable definition $\mathsf{let\ x = b\ in\ sc}$ we call b the body and sc the scope of the variable definition. A naive inlining (*i.e.,* $\mathsf{sc[x := b]}$) provokes an explosion of the size of the produced code. To avoid this explosion, we use the constrained anti-unification of arithmetic terms to eliminate square roots and divisions from the variable definition bodies. For instance, since $(\mathsf{x}_1 + \sqrt{\mathsf{x}_2})/\mathsf{x}_3$ is a template of $a + \sqrt{b \cdot c + d}$ and $e \cdot f/(g + h)$, we can do the following transformation:

$$\begin{array}{lcl} \mathsf{let\ x} = & & \mathsf{let\ (x_1,x_2,x_3)} = \\ \quad \mathsf{if\ F\ then}\ a + \sqrt{b \cdot c + d}\ \mathsf{else}\ e \cdot f/(g+h) & \longrightarrow & \quad \mathsf{if\ F\ then}\ (a, b \cdot c + d, 1)\ \mathsf{else}\ (e \cdot f, 0, g + h) \\ \mathsf{in\ SC} & & \mathsf{in\ SC[x := (x_1 + \sqrt{x_2})/x_3]} \end{array}$$

That eliminates square roots and divisions from the body of the variable definition. Therefore, the language allowed in the substitutions, that are used to define the new body of the variable definition, contains neither the square root operator nor the division one. Moreover, we have to compute a "small" template since we want to minimize the size of the new scope. This template computation is done modulo the arithmetic equational theory and uses a dag representation of expressions in order to minimize the size of this template.

In this paper, we first state a formal definition of this constrained anti-unification problem and give some properties such as sufficient conditions on the equational theory for completeness, then we introduce a constrained anti-unification algorithm for arithmetic expressions and finally we present how this algorithm is used for the program transformation.

2 Anti-Unification with Language Constraint

In this section we first describe the problem of constrained anti-unification on tree-like terms. As usual, given a set of variable $\mathcal{X}$ and a signature Σ, we denote the set of terms by $\mathcal{T}(\Sigma, \mathcal{X})$, and the set of symbols in Σ of arity m by $\Sigma^{(m)}$. A substitution σ, is a mapping from a finite subset of $\mathcal{X}$, its domain, denoted by $\mathcal{D}(\sigma)$, to $\mathcal{T}(\Sigma, \mathcal{X})$, $\mathcal{I}(\sigma)$ being the image $(\{t \in \mathcal{T}(\Sigma, \mathcal{X}) | \exists x \in \mathcal{D}(\sigma), t = \sigma(x)\})$. $t\sigma$ is the application of σ to a term t. $[x \mapsto e]$ is the substitution that replaces x by e. The composition $(\sigma_1\sigma_2)$ is the substition such that for all term t, we have $t(\sigma_1\sigma_2) = (t\sigma_1)\sigma_2$. We will 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.

2.1 Definition

Given these definitions and notations, we define the constrained anti-unification:

▶ Definition 2.1 (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 constrained template of s, denoted $s \preccurlyeq_{\overline{\Sigma}} t$ when there exists a substitution σ such that $t\sigma = s$ and $\mathcal{I}(\sigma) \subset \mathcal{T}(\Sigma \backslash \overline{\Sigma}, \mathcal{X})$.

Therefore all the forbidden symbols (*i.e.*, the symbols in $\overline{\Sigma}$) that are in s appear in t and not in the substitution. We extend this definition to define the template of a set of terms:

▶ Definition 2.2 (Template of finite set). Given $\overline{\Sigma} \subseteq \Sigma$ and a finite set of terms $\mathcal{S} \subseteq \mathcal{T}(\Sigma, \mathcal{X})$, a term t in $\mathcal{T}(\Sigma, \mathcal{X})$ is a template of $\mathcal{S}$, when for all s in $\mathcal{S}$, $s \preccurlyeq_{\overline{\Sigma}} t$.

We call such a template a $\overline{\Sigma}$-template. Let us first give an example of $\overline{\Sigma}$-template:

▶ 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 $[x \mapsto b,\ y \mapsto g(x)]$.

▶ Remark. For unconstrained problems, there is always a template: a (fresh) variable. This is no longer the case when we add constraints *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 $\mathcal{I}(\sigma\sigma') \subseteq \mathcal{T}(\Sigma, \mathcal{X})$ ◀

Without any equational theory, the constrained anti-unification is straightforward

▶ Algorithm 2.3 (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)

Unlike in the usual anti-unification, we are not interested in computing the most specific template, thus when the two terms are in $\mathcal{T}(\Sigma\backslash\overline{\Sigma}, \mathcal{X})$, we terminates the template computation by using a fresh variable.

This anti-unification fails on terms that have different head symbols. In the next section we introduce anti-unification modulo an equational theory in order to be able to anti-unify larger sets of terms.

2.2 Anti-Unification Modulo an Equational Theory

In this section we present that some properties on the sets Σ and $\overline{\Sigma}$ and the use of equational theories might enlarge the set of terms that can be anti-unified and even allow the constrained anti-unification to be complete.

▶ Definition 2.4 (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.

One simple property that can be used to anti-unify terms with different head symbols is the use of neutral elements. It allows us to extend the set of rules for template computation:

▶ Definition 2.5 (Neutral element rule). If f in $\Sigma^{(2)}$ has a right neutral element e_f in $\Sigma^{(0)}$ then for all terms s, t_1, t_2 we can apply the following rules:

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

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

We can define symmetrical rules (RLI) and (LLI) when f admits a left neutral element.

▶ Example 2.2 (Rational number arithmetic). If $\Sigma = \mathbb{Q} \cup \{+, -, \times, /\}$ and all the binary operators are forbidden ($\overline{\Sigma} = \{+, -, \times, /\}$) you can always anti-unify *e.g.,* $(x \times y + (t \times u/v)) - (w + z)$ is a template of $a + (b \times c)/d$ and $a' \times b' - (c' + d')$)

There is a condition that ensure the completeness of the constrained anti-unification. This is the existence of a switch operator:

▶ Definition 2.6 (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.

▶ Proposition 2.2 (Switch completeness). If $\mathcal{T}(\Sigma, \mathcal{X})$ allows 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. ◀

In many usual theories, this operator can be constructed and allows the completeness of the constrained anti-unification.

▶ Example 2.3 (Switch operators).
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 $e_2 = false$.

When a switch element exists we define a switch rule that can always be applied:

▶ Definition 2.7 (Switch rule).
(SW) $ctmp(t_1, t_2) = sw[x \mapsto e, y \mapsto s, z \mapsto t]$

As one can notice, in every theory that has the following properties:

- a binary function f, with an identity element e_f
- a binary function g, with e_f as absorbing element and an identity element e_g

we can construct a similar switch operator using two switch variables. If e_f and e_g are $\overline{\Sigma}$-anti-unifiable, then $sw_2 = f(g(s_1, s), g(s_2, t))$, is a $\overline{\Sigma}$-template of s (with $[s_1 \mapsto e_g, s_2 \mapsto e_f]$) and t (with $[s_1 \mapsto e_f, s_2 \mapsto e_g]$).

However, since we are looking for a small template, we avoid the use of the switch rule as much as possible since the size of the produced template is the sum of the sizes of the input elements. The main criteria to chose the constrained template is the number of forbidden function occurrences that we want to minimize. We introduce directed acyclic graphs as representations of terms to help us to minimize this number in the constrained template.

2.3 Anti-Unification on Dag-like Terms

We want to minimize the number of forbidden function occurrences in the template t. More precisely, we want to minimize the number of function calls on distinct elements, that is the cardinal of the following set, $\{f(a) \mid f(a) \ll t \wedge f \in \overline{\Sigma}\}$. For example the number of distinct f calls in $g(f(b), f(a), f(a))$ is 2. In order to compute a template which minimizes this number, we adopt a dag based representation. This dag representation uses pointers to represent sharing. Therefore the dag nodes are represented by terms extended with pointers in $\mathbb{N}^*$, *i.e.*, $\mathcal{T}(\Sigma, \mathcal{X}, \mathbb{N}^*)$, and a dag is a list of such nodes. We call the length of the dag the length of the list. We denote by :: the `cons` infix constructor, by $[dn_i]_1^n$ the list $[dn_1; ...; dn_n]$ and by $pt(dn)$ the set of pointers in node dn. Since the anti-unification of dags is more restrictive, we only apply sharing to sub-terms that are forbidden functions arguments. Therefore, assuming f is the only forbidden function symbol, the term $g(f(a), f(h(b, f(a))))$ is represented by:

$$\begin{array}{c||c|c} {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 2} \\ g(f(\dot{2}), f(\dot{1})) & h(b, f(\dot{2})) & a \\ \hline \end{array}$$

We use array representation for dags and separate the first element (the root) for clarity reasons. To assure acyclic behaviors we introduce a *right dependency condition*.

▶ Definition 2.8 (Right dependency condition). Given $d = [d_i]_0^n$ a dag, d is a *right dependency dag* when all pointers in node d_i are strictly greater than i (*i.e.*, $pt(d_i) \subseteq \{i + 1; ...; n\}$).

Thus we define the following order denoted $\dot{\gg}$ on dags.

▶ Definition 2.9 (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 usual strict sub-term relation.

The relation $\dot{\gg}$ is well founded since it is the lexical order based on well founded relations $<$ on $\{0, ..., n\}$ and $\gg$ on terms (*i.e.*, $dn_0 \gg dg_0 \Rightarrow min(pt(dn_0)) \leq min(pt(dg_0))$). Therefore it provides us a termination criteria for a set of rule and an induction scheme for right dependency dags. Using that order, we can define the function that transform any dag into its corresponding term.

▶ Definition 2.10 (Dag to term). The associated term of a dag d, denoted by $d^\top$, 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 equality of associated terms means that the pointers identifiers are irrelevant:

▶ Example 2.4 (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 2.11. Given τ a permutation of $\{1, ..., n\}$, the application of τ to a dag node is defined by:
- $\tau(x) = x$
- $\tau(f(d_1, ..., d_m) = f(\tau(d_1), ..., \tau(d_m))$
- $\tau(\dot{k}) = (\dot{\tau(k)})$

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

Proof. By induction using the order from definition 2.9, we only give the pointer case:

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

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

▶ Proposition 2.4 (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 defines a right dependency dag. ◀

In order to minimize the number of forbidden function calls (*i.e.*, the length of the dag), the anti-unification of dags requires to find for each pointer another unique pointer it will be anti-unified with, the corresponding terms being then anti-unified. This is equivalent to rename node identifiers in the dags in a first step and then only anti-unifying pointers and terms with the same identifiers in a second, *e.g.*,

$$\frac{g(f(\dot{1}), f(\dot{2})) \parallel a \mid b}{g(f(\dot{2}), f(\dot{1})) \parallel c \mid d} \longrightarrow \frac{g(f(\dot{1}), f(\dot{2})) \parallel a \mid b}{g(f(\dot{1}), f(\dot{2})) \parallel d \mid c} \longrightarrow \frac{g(f(\dot{1}), f(\dot{2})) \parallel x \mid y}{}$$

Thus we can try different permutations on the input terms before anti-unifying in order to find the more suitable dag representation for template computation.

▶ Definition 2.12 (Pointer anti-unification). The only rule for pointers is the equality rule:

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

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

▶ Definition 2.13 (Common template of dags). The common template of dags is computed node by node: $ctmp([dn_i]_0^n, [dg_i]_0^n) = [ctmp(dn_i, dg_i)]_0^n$

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 associated term:

▶ Proposition 2.5 (Dag extension). Given $[dn_i]_0^n$ a dag:

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

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

It allows the extension of the smaller input dags to the length of the longest one using any kind of nodes. We will see in the next section how to choose these nodes. We also have to ensure that the common template is an acyclic graph. This is the reason why we enforced the acyclicity with the right dependency condition since the computation of the template node by node when the inputs are right dependency dags produces an acyclic graph.

▶ Proposition 2.6 (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:

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

Proof. By induction, trivial for rules $(V), (R), (F)$,

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

(LRI) the identity element is a term in $\mathcal{T}(\Sigma, \mathcal{X})$ without any pointer, $pt(t_2, e_f) = pt(t_2)$

(SW) sw and e are terms, thus $pt(sw[x \mapsto e, y \mapsto s, z \mapsto t]) = (pt(s) \cup pt(t))$ ◀

The general problem of constrained anti-unification being introduced, we present in the following section a constrained anti-unification algorithm for terms in arithmetic.

3 Constrained Anti-Unification in Arithmetic

This section introduces a constrained anti-unification of arithmetic expressions defined with $+$, $-$, $\times$, $/$, $\sqrt{\cdot}$. The constraint is that 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.

The theory of arithmetic includes enough axioms (*e.g.*, commutativity, associativity, distributivity...) to be able to transform any expression into an equivalent expression that has the following form:

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

where 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 normal form. We introduce a direct acyclic graph representation for tuples of arithmetic expressions that corresponds to this normal form, this representation allows sharing of the square roots by using pointers:

▶ Definition 3.1 (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 in $\mathcal{A}\surd$, the set of square root and division free arithmetic terms. We define the associated arithmetic term $[\![d]\!]$ of the dag d with the following rules:

▶ Definition 3.2 (Dag associated arithmetic term).

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

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

▶ Example 3.1 (Dag representation). $(\frac{a_1}{a_2+a_3\cdot\sqrt{d}},\ b\cdot\sqrt{d}\cdot\sqrt{c_2\cdot\sqrt{d}})$ is represented by:

$$\underline{\mathsf{PairD}(\mathsf{DivD}(\mathsf{ExpD}(a_1),\mathsf{VectD}([(a_2,[]);(a_3,[\dot{2}])])),\mathsf{VectD}([(b,[\dot{2};\dot{1}])]))\ \Big\|\ \mathsf{VectD}([(c_2;\dot{2})])\ \Big|\ \mathsf{ExpD}(d)}$$

For clarity and concision, we prefer an array representation of dags using the semantics even if the algorithm is described with dag constructors *e.g.,*

$$\underline{(\frac{a_1}{a_2+a_3\cdot\sqrt{\dot{2}}},\ b\cdot\sqrt{\dot{2}}\cdot\sqrt{\dot{1}})\ \Big\|\ c_1+c_2\cdot\sqrt{\dot{2}}\ \Big|\ d}$$

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

▶ Example 3.2 (Dag interpretations).

$$[\mathsf{PairD}(\mathsf{VectD}(a,[\dot{1}]),\mathsf{ExpD}(b));\mathsf{ExpD}(c)]^\top = \mathsf{PairD}(\mathsf{VectD}(a,[\mathsf{ExpD}(c)]),\mathsf{ExpD}(b))$$
$$\llbracket[\mathsf{VectD}(a,[\dot{1}]);\mathsf{ExpD}(c)]\rrbracket = a\cdot\sqrt{c} = \llbracket[\mathsf{VectD}(a,[\mathsf{ExpD}(c)])]\rrbracket$$

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

▶ Proposition 3.1 (Tree term semantics and arithmetic term semantics). For every dag d_1 and d_2, if $d_1^\top = d_2^\top$ then the associated arithmetic terms are equal $\llbracket d_1\rrbracket = \llbracket d_2\rrbracket$

In this section, we are interested in anti-unifying modulo arithmetic interpretation since our goal is to use the dag representation to compute a constrained template of arithmetic terms.

▶ Definition 3.3 (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 $\llbracket d\rrbracket\sigma_1 =_{\mathcal{A}} \llbracket d_1\rrbracket$ and $\llbracket d\rrbracket\sigma_2 =_{\mathcal{A}} \llbracket d_2\rrbracket$ where $=_{\mathcal{A}}$ denotes 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 contain only pointers. Thus only the root node can contain PairD constructors since the other nodes represent the different square roots of the expression.

▶ Definition 3.4 (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.

The arithmetic 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 $\{\surd,/\}$-anti-unification consist of computing a common template of the dag structure.

The VectD constructor is interpreted as a sum of products and both addition and multiplication are associative and commutative. This means that both lists do not need to be ordered, thus we define the range of a VectD node:

▶ Definition 3.5 (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 sets: $\{\{sq_{11},...,sq_{1p_1}\},...,\{sq_{m1},...,sq_{mp_m}\}\}$

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

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

▶ Definition 3.6 (Arithmetic dag anti-unification). The anti-unification node by node is almost straightforward, we describe some of the rules, the other cases being straightforward:

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

- DivD: We use 1 as neutral element when one of the head symbols is not the division:

(DI) $\mathit{ctmp}(\mathsf{DivD}(d_{11}, d_{12}), t) = \mathsf{DivD}(\mathit{ctmp}(d_{11}, t), \mathit{ctmp}(d_{12}, \mathit{ExprD}(1)))$

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

(E) $\mathit{ctmp}(\mathsf{ExprD}(e_1), \mathsf{ExprD}(e_2)) = \mathsf{ExprD}(x)$

  However, as in usual computation of least general template [1, 5], we record the already anti-unified terms and reuse the variable if possible.

- VectD: we use the template construction using the range from Proposition 3.2:

(VR) $\mathit{ctmp}(\mathsf{VectD}(lv_1), \mathsf{VectD}(lv_2)) = \mathsf{VectD}([(x_i, l_i)])$
with $\mathit{Range}([(x_i, l_i)]) = \mathit{Range}(lv_1) \cup \mathit{Range}(lv_2)$

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

▶ Proposition 3.3 (Anti-unification correctness). The rules introduced in Definition 3.6 compute a $\sqrt{\ }, /$-constrained anti-unifier modulo arithmetic theory.

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

(DI) 1 is a right neutral element for division: $\forall e, e = e/1$
(VR) using Proposition 3.2
$(V0)$ $(\mathsf{VectD}([(0, l_1), ..., (e, []), ..., (0, l_n)]) :: l)^\top = e \times \prod_{x \in \emptyset} \sqrt{x} = e$ ◀

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 produce smaller templates. Therefore, in a first step we will extend the dags with *undefined* elements (#). These elements are not pointed and will be replaced later in the anti-unification process. Replacing undefined elements with appropriate expressions in the dags extension process has two different objectives:

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

   ▶ Example 3.3 (Node duplication).

$$\begin{array}{c||c|c} \sqrt{1},\sqrt{2} & b & a \\ \hline \sqrt{1},\sqrt{1} & c & \# \end{array} = \begin{array}{c||c|c} \sqrt{1},\sqrt{2} & b & a \\ \hline \sqrt{1},\sqrt{1} & c & c \end{array} = \begin{array}{c||c|c} \sqrt{1},\sqrt{2} & b & a \\ \hline \sqrt{1},\sqrt{2} & c & c \end{array} \longrightarrow \begin{array}{c||c|c} \sqrt{1},\sqrt{2} & x & y \\ \hline \end{array}$$

2. Avoid the use of new fresh variables when expressions are already identical variables:

   ▶ Example 3.4 (Avoiding renaming).

$$\begin{array}{c||c|c} \sqrt{1} & x+\sqrt{2} & y \\ \hline 0 & \# & \# \end{array} = \begin{array}{c||c|c} \sqrt{1} & x+\sqrt{2} & y \\ \hline 0 & x+\sqrt{2} & y \end{array} \longrightarrow \begin{array}{c||c|c} z_1\sqrt{1} & x+\sqrt{2} & y \\ \hline \end{array}$$

The node duplication relies on the following transformations:

▶ Definition 3.7 (Node and pointer duplication transformations). The following transformations defines the node duplication (ND) and the pointer exchange (PD):

(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* $d_k = d_l$: $[d_i]_0^n \longrightarrow [[\dot{k}/\dot{l}]d_0, ..., [\dot{k}/\dot{l}]d_{k-1}, d_k, ..., d_n]$

where $[\dot{k}/\dot{l}]d$ is the node d where some occurrences of $\dot{l}$ are replaced by $\dot{k}$.

▶ Proposition 3.4 ((ND) and (PD) correctness). (ND) and (PD) preserve the associated term and the right dependency condition

The renaming of square roots can be avoid by using the following rule:

▶ Definition 3.8 (External node replacement). This rules replaces an undefined node by any element with respect to the right dependency hypothesis:

(PR) *when* $pt(p) \in \{i+1, ..., n\}$ *and* $d_i = \#$: $[d_0, ..., d_n] \longrightarrow [d_0, ..., d_{i-1}, p, d_{i+1}, ..., d_n]$

This rules trivially preserves the semantics since no node that can be reached from the root embeds a pointer to the replaced element since it used to be undefined. In practice, we will only use this rule when it allows to avoid renaming, *i.e.,* when given a set of dags all the i-th nodes are either equal to the same node, p, or undefined, this already ensures that $pt(p) \in \{i+1, ..., n\}$ since p is the i-th node of a 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: Undefined element in node i can be replaced by either:

- another square root of the same dag, using the rules defined in definition 3.7
- The i-th node of one of the dags if all i-th node are either undefined or equal to this expression, using rule PR defined in definition 3.8
- a positive constant:
  when $c \geq 0$, $[d_0, ..., d_{i-1}, \#, d_{i+1}, ..., , d_n] \longrightarrow [d_0, ..., d_{i-1}, \mathsf{Expr}(c), d_{i+1}, ..., d_n]$

Given this undefined elements replacement, we can now define how, by trying different permutations as introduced in section 2.3, we can compute a set of $\{\sqrt{}, /\}$-constrained template for any set of arithmetic expressions:

▶ Definition 3.9 (Arithmetic expression $\{\sqrt{}, /\}$-constrained anti-unification). Given a set of expressions, the following algorithm computes a set of templates of these expressions:

1. Transform every expression into its dag representation.
2. Extend all dags to the same length with undefined elements.
3. Apply a permutation on the dag nodes identifiers with respect to right dependency as introduced in definition 2.11.
4. Replace undefined elements using the different possibilities previously described.
5. Compute the $\{\sqrt{}, /\}$-template node by node using rules of Definition 3.6.

By trying different permutations and different undefined elements replacements, we can compute a set of $\{\sqrt{}, /\}$-templates of the input expressions. Now we will see how this anti-unification is used in a program transformation and why we want to try different templates.

4 Partial Inlining Using Anti-Unification

In this section we present how we use the anti-unification of arithmetic expressions introduced in section 3 to transform the variable definitions in a program transformation. This program transformation aims at removing square roots and divisions in some straight line programs used in embedded systems in order to protect the control flow from rounding errors. All the other aspects of this transformation are described in [9]. We assume that the programs we target are proved to be correct (they do not fail due to square roots of negative number or division by zero) and only ensure the preservation of the semantics in this case.

The techniques used in this program transformation rely on transformations of boolean formulas into formulas that do not depend anymore on square roots or divisions *e.g.,* $\sqrt{x} > y$ becomes $y < 0 \lor x > y^2$. However in order to ensure that these booleans are not computed using square roots or divisions, we also need to ensure that the variables they depend on do not depend either on square roots or divisions. In order to avoid an explosion in the size that an direct inlining would produce, we use a *partial inlining* based on the anti-unification introduced in Section 3 to eliminate divisions and square roots from these variable definitions.

In order to present the transformation of the variable definitions we need to define the languages involved at that point of the transformation:

▶ Definition 4.1 (Expressions and programs normal form). The unary expressions E_u are built with operators, the expressions E with pairs and unary expressions and the programs P allows variable definitions and tests.

$$\begin{array}{lcl} \mathsf{E}_u & := & \mathsf{Var} \mid \mathsf{Const} \mid \mathsf{uop}\ \mathsf{E}_u \mid \mathsf{E}_u\ \mathsf{op}\ \mathsf{E}_u \mid \mathsf{fst}\ \mathsf{E}_u \mid \mathsf{snd}\ \mathsf{E}_u \\ \mathsf{E} & := & (\mathsf{E},\ \mathsf{E}) \mid \mathsf{E}_u \\ \mathsf{P} & := & \mathsf{let\ Var} = \mathsf{P\ in\ P} \mid \mathsf{if\ P\ then\ P\ else\ P} \mid \mathsf{E} \end{array}$$

where the set constants Const is included in $\mathbb{R} \cup \{True,\ False\}$, the set of binary operators bop is $\{+, \times, /, =, \neq, >, \geq, <, \leq, \land, \lor\}$ and the set of unary operator uop is $\{\sqrt{\ }, -, \neg\}$.

Therefore our aim is to use this anti-unification in order to remove the square roots and divisions from the variable definitions before eliminating these operations in all the Boolean expressions. The global algorithm is recursive, therefore, given a variable definition let x = body in sc, we can assume that the square roots and divisions have been eliminated from all the boolean part of the tests and all the variable definitions that are in body. In this case, the body allows a decomposition into a square root and division free program part, representing all these local variable definitions and the test cases, and the possible returned expression corresponding to the different test cases that possibly contain divisions and square roots. We give the following example:

▶ Example 4.1. The program if F then let z = a in z + $\sqrt{\mathsf{b}}$ else c/d denoted by p is decomposed in a cases construction (represented as a meta-function):

$$cases = fun\ (x, y) \rightarrow \mathsf{if\ F\ then\ let\ y = a\ in}\ x\ \mathsf{else}\ y$$

And a tuple of returned expressions $lexp = (\mathrm{z}+\sqrt{\mathrm{b}},\mathrm{c/d})$ such that $cases(lexp) = \mathsf{p}$

Now using the constrained anti-unification we use the following rule to eliminate the square roots and divisions from the body of a variable definition:

▶ Proposition 4.1 (Variable definition transformation). If T is a $\{\sqrt{\ }, /\}$constrained template of all the e_i, *i.e.,* $\forall i, e_i = T\sigma_i$, we have the following semantics equality:

$$\mathsf{let\ x} = cases(e_1, ..., e_n)\ \mathsf{in\ p2} \overset{sem}{=} \mathsf{let}\ var(\sigma_1) = cases(arg(\sigma_1), ..., arg(\sigma_n))\ \mathsf{in\ p2[\ x} \mapsto T\ ]$$

This property of the cases decomposition has been formally proven in the Pvs proof assistant on a slightly different representation of variables, using projections instead of multiple variable definitions (see [9]). Moreover since we use a $\{\sqrt{\ }, /\}$ constrained template, none of the substitution arguments (*i.e.,* $arg(\sigma_i)$) contain any square root or division and therefore the new body is free of these operations

▶ Example 4.2. We present an example of such a partial inlining:

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

The decomposition of the set of expressions coming from the different test cases directly uses the anti-unification algorithm introduced in section 3. However there is one particular case we have to take care of when we want to avoid renaming using rule (PR) from definition 3.8. In this case we need to ensure the elements we want to replace the undefined with are positive. For example the following transformation is not allowed:

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

since 0 is not absorbing for failure (square root of negative numbers) and therefore the produced scope $\mathsf{P[x \mapsto x.\sqrt{y}]}$ would fail when $\mathsf{y} \leq 0$. However, using the hypothesis that the input program does not fail, we are able to build a set of expressions that are positive by taking for example the square roots that were defined previously in the program. We will only use such expressions to apply the undefined element replacement from Definition 3.8.

5 Algorithm Implementation

Due to the different choices, the number of possible templates increases very fast with the number of expressions to anti-unify and the length of the corresponding dags. Indeed, let n be the number of dags and $l+1$ the maximal length, since for each dag we have $l!$ possible pointers identifiers permutations (the head node have to stay as head element), and even if some of them do not respect the *right dependency condition*, the maximal number of permutations can grow up to $l!^n$. The different replacements of undefined elements also increase the set of possible templates. However, experiments show that, in most cases, there are a lots of templates that are equivalent regarding the size of the output program, this allows us to only use a random fraction of all the possible permutation and still get the best possible results.

This anti-unification has been tested during the transformation of different programs using the OCamL implementation of the program transformation (see [9]). We also tested the anti-unification by transforming simple variable definitions with different formulas as scope (*e.g.,* `let x,y = ... in x * y > a`) in order to only reflect the consequences of the anti-unification choices. We have the following convention: `4_3_def` define 3 tuples of 4 elements, for each file and each percentage of permutations tried, we give the best output size (in KB) and time to transform:

% permutations	10		1		0.1	
`5_2_def_scope1`	1464	1m15	1464	9s1	1465	1s2
`5_2_def_scope2`	3552	1m15	3552	9s1	3553	1s1
`4_3_def_scope1`			587	1m17	613	2s1
`4_3_def_scope2`			5265	1m17	5126	2s2

From these results, we can see that:

- time taken by the transformation depends on the template computation, not on the scope
- output size depends on the scope of the variable definition, not on the body
- redundancy allows us to take a small sample of permutations to get the optimal template

However, even if we eliminate some of the templates, the worst cases can produce programs that are hundreds of time bigger than the best one. Avoiding renaming expressions is also crucial on longer programs since when we do not take care of such renaming, the size of the produced program can be thousands of time bigger than the smaller one.

6 Conclusion

In this paper we introduced a constrained anti-unification, that is an anti-unification where the language allowed in the terms of the substitution is a subset of the one used for the input terms. Several general properties of such anti-unification have been presented in order to extend the set of possible templates using some equational theory and therefore to find the more suitable one in each context.

We used this anti-unification as a tool to transform variables definitions with conditional expressions in a program transformation that eliminates square roots and divisions. Due to the rich equational theory we are allowed to use in this context (*i.e.,* arithmetic), the number of possible templates increase exponentially with the complexity and is not easy to handle. However, the use of this specialized anti-unification algorithm allowed us to partially inline variable definitions in order to remove square roots and divisions from them by almost only inlining the operations we want to eliminate. This allowed us to produce programs whose output size is still reasonable in comparison to a complete inlining.

We now aim at proving the correctness of this anti-unification in the Pvs proof assistant for any permutation and any undefined element replacement in order to complete the proof of the program transformation. We also plan to use this template computation to extend the language this transformation handles to bounded loops and function definitions.

References

1 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. A modular equational generalization algorithm. In M. Hanus, editor, *LOPSTR*, volume 5438 of *LNCS*, pages 24–39, 2008.

2 F. Baader and W. Snyder. Unification theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. 2001.

3 P. E. Bulychev, E. V. Kostylev, and V. A. Zakharov. Anti-unification algorithms and their applications in program analysis. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *LNCS*, pages 413–423, 2009.

4 G. Huet. *Resolution d'Equations dans les langages d'ordre 1, 2,..., ω*. PhD thesis, Université de Paris VII, 1976.

5 T. Kutsia, J. Levy, and M. Villaret. Anti-unification for unranked terms and hedges. In M. Schmidt-Schauß, editor, *RTA*, volume 10 of *LIPIcs*, pages 219–234. Dagstuhl, 2011.

6 S. Lang. *Algebraic Number Theory.* Graduate Texts in Mathematics. Springer, 1994.

7 J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming.*, pages 587–625. Morgan Kaufmann, 1988.

8 A. Narkawicz, C. Muñoz, and G. Dowek. Formal verification of air traffic prevention bands algorithms. Technical Memorandum NASA/TM-2010-216706, NASA, 2010.

9 P. Neron. A formal proof of square root and division elimination in embedded programs. In C. Hawblitzel and D. Miller, editors, *CPP*, volume 7679 of *LNCS*, pages 256–272, 2012.

10 C. Oancea, C. So, and S. M. Watt. Generalization in maple, 2005.

11 F. Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

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

13 L. Pottier. Generalisation de termes en theorie equationnelle. Cas associatif-commutatif. Rapport de recherche RR-1056, INRIA, 1989.

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

15 J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.