Generalization in Maple

Cosmin Oancea Clare So Stephen M. Watt

Ontario Research Centre for Computer Algebra
Department of Computer Science
University of Western Ontario
London Ontario, CANADA N6A 5B7

{coancea,clare,watt}@orcca.on.ca

Abstract

We explore the notion of generalization in the setting of symbolic mathematical computing. By "generalization" we mean the process of taking a number of instances of mathematical expressions and producing new expressions that may be specialized to all the instances. We first identify a number of ways in which generalization may be useful in the setting of computer algebra, and formalize this generalization as an antiunification problem. We present a single-pass algorithm for antiunification and give some examples.

Keywords: Antiunification, generalization, pattern matching, computer algebra

1 Introduction

Pattern matching has long been an important tool in symbolic mathematical computation, and all major general purpose computer algebra systems have facilities to do this in various forms. An important theoretical basis for pattern matching is the process of *unification*, explored by Robinson in 1965 [4]. Unification takes two patterns or a pattern and a subject expression, and determines substitutions to make the two equal. This concept has found many applications in computing, and has become part of the standard technology for type inference [1].

This paper looks at the dual process: taking expressions and finding a pattern that may be specialized by some substitution to give those expressions. This problem has been studied by Reynolds [3], under the name "antiunification," and by Plotkin [2], under the name "generalization." This topic, however, has not been as studied nearly as thoroughly as unification.

Our motivation to study anti-unification arose in the context of examining large collections of expressions for common properties. For example, in order to build a database to guide mathematical handwriting recognition, we have extracted all the mathematical expressions from five years worth of articles from `www.arXiv.org`. These 20,000 articles contain millions of subexpressions. In this context we wish to discover patterns, for example the similarity between $\sqrt{x^2+y^2}$ and $\sqrt{A^2+b^2}$.

Pattern matching is an important tool in symbolic mathematical computation, and all major general purpose systems for computer algebra have facilities to do pattern matching in various ways. It is only natural that the dual process fit this context as well. In fact, we believe that automated generalization can have as useful a role in symbolic mathematical computation as pattern matching and solving.

In the following, we first briefly discuss unification generally. Then we discuss antiunification and present Plotkin's algorithm. We describe our single-pass anti-unification algorithm, and present a Maple implementation. We conclude with some examples.

2 Unification

Unification takes two expressions containing variables and determines whether there are substitutions for the variables that make the expressions equal. If there is such a substitution, we say the two expressions *unify* and the substitution is called a *unifier*.

Example Suppose we have expressions

$$\begin{aligned} E_1 &= \alpha^2 + u \times \alpha + \beta \\ E_2 &= (x+1)^2 + \gamma \times (x+1) + \delta \end{aligned}$$

Then E_1 and E_2 have the following as a unifier (σ_1, σ_2), $\sigma_1 = \{\alpha \mapsto (x+1), \beta \mapsto (x+2)\}$, $\sigma_2 = \{\gamma \mapsto u, \delta \mapsto (x+2)\}$, for which $\sigma_1(E_1) = \sigma_2(E_2) = (x+1)^2 + u \times (x+1) + (x+2)$ ∎

To fix terminology, we consider a set of expressions $E(\Sigma, V)$ formed from some set of literal symbols Σ, e.g. $\{+, \times, \uparrow, ..., x, y, z, ..., 1, 2, 3, ...\}$, and variables V, e.g. $\{\alpha, \beta, \gamma, ...\}$, according to some rules of well-formedness. A *substitution* is a mapping $\sigma : V \to E(\Sigma, V)$. We may lift a substitution to a mapping $E(\Sigma, V) \to E(\Sigma, V)$ by applying the variable substitution to the variables appearing in an expression. For convenience, we assume V to be an infinite set so we can have as many variables as we need.

Two expressions do not necessarily unify, and if they do there is not necessarily a unique unifier. There is however, *most general unifier* which is unique up to renaming of variables. This is the one that imposes the fewest constraints on the variables. In the example above, the most general unifier is $\sigma_1 = \{\alpha \mapsto (x+1)\}, \sigma_2 = \{\gamma \mapsto u, \delta \mapsto \beta\}$

Several variants of unification have been studied, taking into account mathematical properties of the set of expressions. These include associative unification (allowing the term-forming operators to be associative) and AC unification (allowing the term forming operators to be associative and commutative). Any survey will describe algorithms for unification.

3 Antiunification

The process of antiunification is the dual of unification. It takes two expressions $E_1, E_2 \in E(\Sigma, V)$ and produces $E_3 \in E(\Sigma, V)$ such that there exist substitutions σ_1 and σ_2 such that $\sigma_1(E_3) = E_1$ and $\sigma_2(E_3) = E_2$. We call the pair of substitutions an *antiunifier*, and the expression E_3 a *generalization* of the expressions. An antiunifier always exists, but is not necessarily unique. There is, however, a unique *most specific antiunifier* that places the most restrictions on the variables. This is also known as the *least general generalization*, and is unique up to renaming of variables.

Example Suppose we have the two expressions

$$\begin{aligned} E_1 &= (x+1)^2 + (y+1)^2 \\ E_2 &= \sin^2(x) + \cos^2(y) \end{aligned}$$

Then the pair have $\alpha + \beta$ as a generalization and $\alpha^2 + \beta^2$ as their unique *most specific* generalization. ∎

Antiunification is less well studied than unification. To our knowledge there is not an analogous rich body of literature studying variants of antiunification under the presence of mathematical properties of the term-forming operators (e.g. associativity, commutativity).

We state Plotkin's algorithm[2] for antiunification in our notation:

Plotkin's Algorithm:

1. Set G_i to $E_i\,(i = 1, 2)$. Set σ_i to $\{\,\}$, the empty substitution, $(i = 1, 2)$.
2. Try to find terms t_1, t_2 which have the same place in G_1, G_2 respectively and such that $t_1 \neq t_2$ and either t_1 and t_2 begin with different function letters [have different operators] or at least one of them is a variable.
3. If there are no such t_1, t_2 then halt. G_1 is a least generalization of $\{E_1, E_2\}$ and $G_1 = G_2$, $\sigma_i(G_i) = E_i\,(i = 1, 2)$.
4. Choose a variable β distinct from any in G_1 or G_2 and wherever t_1 and t_2 occur in the same place in G_1 and G_2, replace each by β.
5. Change σ_i to $\{\beta \mapsto t_i\} \cup \sigma_i\,(i = 1, 2)$.
6. Go to 2.

Note that step 2 requires a traversal of the expressions G_1, G_2 and step 4 requires a search through the expressions for occurrences of t_1, t_2. This leads to $O(n^3)$ worst case performance for a naïve implementation.

4 A Single Pass Algorithm

We begin by defining the procedure *CombineSubs* that takes an expression E and two pairs of substitutions ρ_1, ρ_2 and τ_1, τ_2 and produces a new expression G and pair of substitutions σ_1, σ_2. These are constructed such that the right-hand sides of the substitutions ρ_i and τ_i appear uniquely in σ_i, renaming left-hand side variables of ρ_i if necessary.

Notation We adopt the notation $\sigma = \{\alpha \mapsto (a_1, a_2), \beta \mapsto (b_1, b_2), ...\}$ for the pair of substitutions $\sigma_1 = \{\alpha \mapsto a_1, \beta \mapsto b_1, ...\}, \sigma_2 = \{\alpha \mapsto a_2, \beta \mapsto b_2, ...\}$.

Example We begin with the expression $E = \alpha + x + \beta$ and the substitution pairs $\rho = \{\alpha \mapsto (u, v), \beta \mapsto (1, 2)\}$ and $\tau = \{\gamma \mapsto (u, v), \delta \mapsto (2, 1)\}$.

Then (G, σ) :=*CombineSubs*(E, ρ, τ) gives $G = \gamma + x + \beta$ and $\sigma = \{\gamma \mapsto (u, v), \delta \mapsto (2, 1), \beta \mapsto (1, 2)\}$. ∎

We are now in a position to state our formulation of the antiunification algorithm:

Algorithm *Antiunify*.

Input: two expressions, e_1 and e_2.

Output: one expression g, possibly containing new variables $\phi_1, \phi_2, ...$
a pair of substitutions (σ_1, σ_2) for ϕ_i such that $\sigma_1(g) = e_1, \sigma_2(g) = e_2$

Make a single recursive pass over the input, traversing the two expressions e_1 and e_2 in parallel. At each level, do the following:

1. If $e_1 = e_2$, return e_1 as g with the empty substitution.

2. If e_1 and e_2 both variables, constants or expressions with different operator or arity, then introduce a new variable ϕ_k. Return ϕ_k as the expression g, together with the pair of substitutions $\{\phi_k \mapsto (e_1, e_2)\}$.

3. Otherwise e_1 and e_2 are trees with the same operator and arity.

   Apply *Antiunify* to the corresponding pairs of subexpressions of e_1 and e_2 to get the corresponding subexpression of g. Use *CombineSubs* to combine the substitutions from the subexpressions to get the substitution to pair with g.

5 Maple Implementation

It is straightforward to implement the recursive antiunification algorithm in Maple, as shown in Figure 1. We interpret the Maple node types (e.g. $+$, $*$) as the fixed operators of the expression language, and treat the symbolic function of an unevaluated application as an ordinary subexpression.

In the implementation of *CombineSubs*, we check, for each substitution in `s`, whether there is already a substitution in `t` with the same targets. If there is no such substitution in `t`, then we add this as a new one. If there is, the substitution from `t` is used and a relabeling is made to make `expr` use the `t` substitution's variable instead of the `s` substitutions variable.

6 Examples

We conclude with some examples showing the antiunification of expressions. We do not bother showing the trivial implementation of the `genvar` function, that generates new variables $\phi_i, i = 0, ...$

In the first example, antiunification discovers a correspondence between variables.

```
> antiunify(a*x^2+b*x+c, u*y^2 + v*y + w, genvar);
```

$$\phi_1{\phi_2}^2 + \phi_3\phi_2 + \phi_5,\ \{\phi_2 = [x, y], \phi_1 = [a, u], \phi_3 = [b, v], \phi_5 = [c, w]\}$$

New variables are also introduced for function symbols and for constants that differ.

```
> antiunify(sin(3*x), cos(4*y), genvar);
```

$$\phi_6\left(\phi_7\phi_8\right),\ \{\phi_6 = [\sin, \cos], \phi_7 = [3, 4], \phi_8 = [x, y]\}$$

The next two examples show that corresponding sub-expressions need not have the same shape. Here ϕ_{10} maps to $3x$ in one case and y in the other.

```
> antiunify(sin(3*x), exp(y), genvar);
```

$$\phi_9\left(\phi_{10}\right),\ \{\phi_9 = [\sin, \exp], \phi_{10} = [3\,x, y]\}$$

```
> antiunify(x^2 + y^2, u^n - 1, genvar);
```

$$\phi_{11}{}^{\phi_{12}} + \phi_{13},\ \left\{\phi_{11} = [x, u], \phi_{12} = [2, n], \phi_{13} = [y^2, -1]\right\}$$

Finally we note that two expressions always antiunify, even if only to produce the trivial pattern.

```
> antiunify(sin(3*x), u + v, genvar);
```

$$\phi_{14},\ \{\phi_{14} = [\sin(3\,x), u + v]\}$$

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, pages 153–163, 1970.

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

[4] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, pages 23–41, 1965.

```
# Compute the antiunifier of e1 and e2 together with substitutions.
au := proc(e1, e2)
    local g, s, newop, newargs, i, ai, si;

    if e1 = e2 then
        g := e1; s := {}
    elif whattype(e1) <> whattype(e2) or nops(e1) <> nops(e2) or
         type(e1, name) or type(e1, constant)
    then
        # Make a substitution.
        g := genvar(); s := {g = [e1, e2]}
    else
        newop, s := au(op(0,e1), op(0,e2));
        newargs  := NULL;

        for i to nops(e1) do
            ai, si := au(op(i,e1), op(i,e2));
            ai, s  := CombineSubs(ai, si, s);
            newargs := newargs, ai
        end do;
        g := apply(newop, newargs)
    end if;
    g, s
end proc:

# Modify (expr, s) so that they use any substitutions already in t.
# E.g.  Input   a+x+b,  {a=[u,v], b=[1,2]},   {g=[u,v], d=[2,1]}
#       gives   g+x+b,  {g=[u,v], d=[2,1], b=[1,2]}
CombineSubs := proc(expr, s, t)
    local newt, si, ti, foundone, relabel;
    relabel := {};
    newt := t;
    for si in s do
        foundone := false;
        for ti in t while not foundone do
            if op(2,si) = op(2,ti) then
                relabel  := {op(1,si) = op(1,ti)} union relabel;
                foundone := true;
            end if
        end do;
        if not foundone then
            newt := {si} union newt
        end if
    end do;
    subs(relabel, expr), newt
end proc:
```

Figure 1: Maple Implementation