A Unification Algorithm for GP

Ivaylo Hristakiev and Detlef Plump
The University of York, UK

Abstract. The graph programming language GP allows to apply sets of rule schemata (or “attributed” rules) nondeterministically. To analyse conflicts of programs statically, graphs labelled with expressions are overlaid to construct critical pairs of rule applications. Each overlay induces a system of equations whose solutions represent different conflicts. We present a rule-based unification algorithm for GP expressions that is terminating and sound. Soundness means that every substitution generated by the algorithm solves the input system of equations. Since GP labels are lists constructed by concatenation, unification modulo associativity and unit laws is required. This problem, which is similar to word unification, is infinitary in general but becomes finitary due to GP’s rule schema syntax.

1 Introduction

A common programming pattern in the graph programming language GP [7, 8] is to apply a set of graph transformation rules as long as possible. To execute such a loop \( \{r_1, \ldots, r_n\} \) on a host graph, in each iteration an applicable rule \( r_i \) is selected and applied. As rule selection and rule matching are nondeterministic, different graphs may result from the loop. Thus, if the programmer wants the loop to implement a function, a useful tool would be a static analysis that establishes or refutes functional behaviour.

The above loop is guaranteed to produce a unique result if the rule set \( \{r_1, \ldots, r_n\} \) is terminating and confluent. However, conventional confluence analysis via critical pairs [6] assumes rules with constant labels whereas GP employs rule schemata (or “attributed” rules) whose graphs are labelled with expressions. Confluence of attributed graph transformation rules has been considered in [4, 2, 3], but we are not aware of algorithms that check confluence over non-trivial attribute algebras such as GP’s which includes list concatenation and Peano arithmetic. The problem is that the equational theory of an attribute algebra needs to be taken into account when constructing critical pairs and checking their joinability.

For example, [4] presents a method of constructing critical pairs in the case where the equational theory of the attribute algebra is represented by a convergent term rewriting system. The algorithm first computes normal forms of the attributes of overlayed nodes and subsequently constructs the most general unifier of the normal forms. This has been shown to be incomplete [2, p.198] in that the constructed set of critical pairs need not represent all possible conflicts.
For, the most general unifier produces identical attributes—but it is necessary to find all substitutions that make attributes equivalent in the equational theory.

Graphs in GP rule schemata are labelled with lists of integer and string expressions, where lists are constructed by concatenation. In host graphs, list entries must be constant values. Integers and strings are subtypes of lists in that they represent lists of length one. As a simple example, consider the program in Figure 1 for calculating shortest distances. The program expects input graphs with non-negative integers as edge labels, and arbitrary lists as node labels. There must be a unique marked node (drawn shaded) whose shortest distance to each reachable node has to be calculated. The rule schemata \texttt{init} and \texttt{add}

\begin{verbatim}
main = init; {add, reduce}!
init(x: list) add(x,y: list; m,n: int)
\begin{align*}
x & \Rightarrow x:0 \\
\begin{array}{c}
x : m \quad n \\
1 \quad 1 \quad 2
\end{array} & \Rightarrow \begin{array}{c}
x : m \\
1 \quad 2
\end{array} \quad y \quad \begin{array}{c}
\quad m \\
\quad n
\end{array} \Rightarrow \begin{array}{c}
\begin{array}{c}
x : m \\
1 \quad 2
\end{array} \quad y : m + n \\
\end{array}
\end{align*}
\end{verbatim}

\begin{verbatim}
reduce(x,y: list; m,n,p: int)
\begin{align*}
x & \Rightarrow x : m \\
\begin{array}{c}
x : m \quad n \\
1 \quad 1 \quad 2
\end{array} & \Rightarrow \begin{array}{c}
x : m \\
1 \quad 2
\end{array} \quad y : p \\
\begin{array}{c}
\quad m \\
\quad n
\end{array} \Rightarrow \begin{array}{c}
\begin{array}{c}
x : m \\
1 \quad 2
\end{array} \quad y : m + n \\
\end{array}
\end{align*}
where m + n < p
\end{verbatim}

Fig. 1. A program calculating shortest distances

append distances to the labels of nodes that have not been visited before, while \texttt{reduce} decreases the distance of nodes that can be reached by a path that is shorter than the current distance.

To construct the conflicts of the rule schemata \texttt{add} and \texttt{reduce}, their left-hand sides are overlayed. For example, the structure of the left-hand graph of \texttt{reduce} can match the following structure in two different ways:

Consider a copy of \texttt{reduce} in which the variables have been renamed to \texttt{x'}, \texttt{m'}, etc. To match \texttt{reduce} and its copy differently requires solving the system of equations \{\texttt{x} \, m = ' \texttt{y}' \, p', \, y \, p = ' \texttt{x}' \, m'\}. Solutions to these equations should be as general as possible to represent all potential conflicts resulting from the above overlay. In this simple example, it is clear that the substitution

\[ \sigma = \{ x' \mapsto y, \, m' \mapsto p, \, y' \mapsto x, \, p' \mapsto m \} \]

is a most general solution. It gives rise to the following critical pair:\footnote{For simplicity, we ignore the condition of \texttt{reduce}.}
In general though, equations can arise that have several independent solutions. For example, the equation $n \cdot x = y \cdot 2$ (with $n$ of type `int` and $x,y$ of type `list`) has the minimal solutions

$$
\sigma_1 = \{ x, y \mapsto \text{empty}, n \mapsto 2 \} \quad \text{and} \quad \sigma_2 = \{ x \mapsto z:2, y \mapsto n:z \}
$$

where `empty` represents the empty list and $z$ is a list variable.

Seen algebraically, we need to solve equations modulo the associativity and unit laws

$$
\text{AU} = \{ x : (y : z) = (x : y) : z, \text{empty} : x = x, x : \text{empty} = x \}.
$$

This problem is similar to word unification [1], which attempts to solve equations modulo associativity. Solvability of word unification is decidable, albeit in PSPACE [5], but there is not always a finite complete set of solutions. The same holds for AU-unification (see Subsection 3.3). Fortunately, GP’s syntax for left-hand sides of rule schemata forbids labels with more than one list variable. We conjecture that this guarantees that left-hand overlays induce equation systems possessing finite complete sets of solutions.

This paper is the first step towards a static confluence analysis for GP programs. In Section 3, we present a rule-based unification algorithm for systems of equations with left-hand expressions of rule schemata. We show that the algorithm always terminates and that it is sound in that each substitution generated by the algorithm is an AU-unifier of the input problem.

## 2 Rule Schemata

We refer to [7, 8] for the definition of GP and more example programs. In this section, we define (unconditional) rule schemata which are the “building blocks” of graph programs.

A graph over a label set $C$ is a system $G = (V, E, s, t, l, m)$, where $V$ and $E$ are finite sets of nodes (or vertices) and edges, $s, t : E \to V$ are the source and target functions for edges, $l : V \to C$ is the node labelling function and $m : E \to C$ is the edge labelling function. We write $\mathcal{G}(C)$ for the class of all graphs over $C$.

Figure 2 shows an example for the declaration of a rule schema. The types `int` and `string` represent integers and character strings. Type `atom` is the union of `int` and `string`, and `list` represents lists of atoms. Given lists $l_1$ and $l_2$, we write $l_1 : l_2$ for the concatenation of $l_1$ and $l_2$. The empty list is denoted by `empty`. In pictures of graphs, nodes or edges without label (such as the dashed edge in Figure 2) are implicitly labelled with the empty list. We equate lists of length one with their entry to obtain the syntactic and semantic `subtype` relationships shown in Figure 3. Hence, for example, all labels in Figure 2 are list expressions.
bridge(x, y: list; a: atom; n: int; s, t: string)

![Diagram of rule schema]

**Fig. 2.** Declaration of a rule schema

Also, GP 2 allows to mark nodes and edges. For example, the outermost nodes in Figure 2 are marked by a grey shading, and the dashed edge is a marked edge (labelled with the empty list). Figure 4 gives a grammar in Extended Backus-Naur Form defining the abstract syntax of labels. (In this paper, we omit string concatenation because it would inflate the unification algorithm without posing an extra challenge.) The functions `llength` and `slength` return the length of a list resp. string, while `indeg` and `outdeg` access the indegree resp. outdegree of a left-hand node in the host graph.

Figure 4 defines four syntactic categories of expressions: Integer, String, Atom and List, where Integer and String are subsets of Atom which in turn is a subset of List. Category Node is the set of node identifiers used in rule schemata. Moreover, IVar, SVar, AVar and LVar are the sets of variables of type `int`, `string`, `atom` and `list`. We assume that these sets are disjoint and define `Var = IVar ∪ SVar ∪ AVar ∪ LVar`. The mark components of labels are represented graphically rather than textually.

Each expression `l` has a unique smallest type, denoted by `type(l)`, which can be read off the hierarchy in Figure 3 after `l` has been normalised with the rewrite rules shown at the beginning of Subsection 3.2. We write `type(l₁) < type(l₂)` or `type(l₁) ≤ type(l₂)` to compare types according to the subtype hierarchy. If the types of `l₁` and `l₂` are incomparable, we write `type(l₁) || type(l₂).

![Subtype hierarchy for labels]

**Fig. 3.** Subtype hierarchy for labels
The values of rule schema variables at execution time are determined by graph matching. To ensure that matches induce unique “actual parameters”, expressions in the left graph of a rule schema must have a simple shape.

**Definition 1 (Simple expression).** A simple expression contains no arithmetic operators (with the possible exception of a unary minus preceding a sequence of digits), no length or degree operators, and at most one occurrence of a list variable.

For example, given the variable declarations of Figure 2, a:x and y:n:n are simple expressions whereas n*2 or x:y are not simple.

**Definition 2 (Rule schema).** A rule schema \((L, R, I)\) consists of graphs \(L, R\) in \(G(\text{Label})\) and a set \(I\), the interface, such that \(I \subseteq V_L \cap V_R\). All labels in \(L\) must be simple and all variables occurring in \(R\) must also occur in \(L\).

When a rule schema is graphically declared, as in Figure 2, the interface \(I\) is represented by the node numbers in \(L\) and \(R\). Nodes without numbers in \(L\) are to be deleted and nodes without numbers in \(R\) are to be created. All variables in \(R\) have to occur in \(L\) so that for a given match of \(L\) in a host graph, applying the rule schema produces a graph that is unique up to isomorphism.

### 3 Unification

We start with introducing some technical notions such as substitutions, unification problems and complete sets of unifiers. Then, in Subsection 3.2, we present our unification algorithm. In Subsection 3.3, we prove that the algorithm terminates and is sound.
3.1 Preliminaries

A substitution is a family of mappings \( \sigma = (\sigma_X)_{X \in \{I,S,A,L\}} \) where \( \sigma_I: \text{IVar} \to \text{Integer} \), \( \sigma_S: \text{SVar} \to \text{String} \), \( \sigma_A: \text{AVar} \to \text{Atom} \), \( \sigma_L: \text{LVar} \to \text{List} \). Here Integer, String, Atom and List are the sets of expressions defined by the GP label grammar of Figure 4. For example, if \( z \in \text{LVar} \), \( x \in \text{IVar} \) and \( y \in \text{SVar} \), then we write \( \sigma = \{ x \mapsto x + 1, z \mapsto y : -x : y \} \) for the substitution that maps \( x \) to \( x + 1 \), \( z \) to \( y : -x : y \) and every other variable to itself.

Applying a substitution \( \sigma \) to an expression \( t \), denoted by \( t\sigma \), means to replace every variable \( x \) in \( t \) by \( \sigma(x) \) simultaneously. In the above example, \( \sigma(z : -x) = y : -x : y : -(x + 1) \).

By \( \text{Dom}(\sigma) \) we denote the set \( \{ x \in \text{Var} \mid \sigma(x) \neq x \} \) and by \( \text{VRan}(\sigma) \) the set of variables occurring in the expressions \( \{ \sigma(x) \mid x \in \text{Var} \} \). A substitution \( \sigma \) is idempotent if \( \text{Dom}(\sigma) \cap \text{VRan}(\sigma) = \emptyset \).

Definition 3 (Unification problem). A unification problem is a finite multiset of equations \( P = \{ s_1 \equiv t_1, \ldots, s_n \equiv t_n \} \) between simple list expressions.

The symbol \( \equiv \) signifies that the equations must be solved rather than having to hold for all values of variables.

Consider the equational axioms for associativity and unity,

\[
\text{AU} = \{ x : (y : z) = (x : y) : z, \text{empty} : x = x, x : \text{empty} = x \}
\]

where \( x, y, z \) are variables of type \text{list}, and let \( =_{\text{AU}} \) be the equivalence relation on expressions generated by these axioms.

Definition 4 (Unifier). A unifier of a problem \( P = \{ s_1 \equiv t_1, \ldots, s_n \equiv t_n \} \) is a substitution \( \sigma \) such that

\[
s_1\sigma =_{\text{AU}} t_1\sigma, \ldots, s_n\sigma =_{\text{AU}} t_n\sigma.
\]

The set of all unifiers of \( P \) is denoted by \( \mathcal{U}(P) \). We say that \( P \) is unifiable if \( \mathcal{U}(P) \neq \emptyset \).

A substitution \( \sigma \) is more general on a set of variables \( X \) than a substitution \( \theta \) if there exists a substitution \( \lambda \) such that \( x\theta =_{\text{AU}} x\sigma\lambda \) for all \( x \in X \). In this case we write \( \sigma \leq_X \theta \) and say that \( \theta \) is an instance of \( \sigma \) on \( X \). Substitutions \( \sigma \) and \( \theta \) are equivalent on \( X \), denoted by \( \sigma =_X \theta \), if \( \sigma \leq_X \theta \) and \( \theta \leq_X \sigma \).

Definition 5 (Complete set of unifiers). A set \( C \) of substitutions is a complete set of unifiers of a unification problem \( P \) if

1. \( C \subseteq \mathcal{U}(P) \), that is, each substitution in \( C \) is a unifier of \( P \), and
2. for each \( \theta \in \mathcal{U}(P) \) there exists \( \sigma \in C \) such that \( \sigma \leq_X \theta \), where \( X = \text{Var}(P) \).

Set \( C \) is also minimal if it satisfies
3. each two substitutions in \( \mathcal{C} \) are incomparable with respect to \( \leq_X \), that is, for all \( \sigma, \sigma' \in \mathcal{C} \), \( \sigma \leq_X \sigma' \) implies \( \sigma = \sigma' \).

If a unification problem \( P \) is not unifiable, then the empty set is a minimal complete set of unifiers of \( P \).

We call a variable \( x \) solved in \( P \) if it occurs exactly once in \( P \), namely on the left-hand side of an equation \( x = \mathit{L} \) with \( \text{type}(x) \geq \text{type}(\mathit{L}) \).

**Definition 6 (Solved form).** A unification problem \( P = \{ x_1 =? t_1, \ldots, x_n =? t_n \} \) is in solved form if the variables \( x_i \) are pairwise distinct and solved in \( P \). In this case we define the substitution \( \rightarrow P = \{ x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \} \).

For example, if \( a \) is an atom variable and \( x \) a list variable, then the problems \( \{ x = a \} \) and \( \{ x : 1 = a \} \) are in solved form whereas \( \{ a = x : 2 \} \) and \( \{ a : 1 = 2 : 1 \} \) are not solved. For simplicity, we replace =? with = in unification problems from now on.

The minimal complete set of unifiers of the problem \( \{ a : x = y : 2 \} \) (where \( a \) is an atom variable and \( x, y \) are list variables) is \( \{ \sigma_1, \sigma_2 \} \) with

\[ \sigma_1 = \{ a \mapsto 2, x \mapsto \text{empty}, y \mapsto \text{empty} \} \quad \text{and} \quad \sigma_2 = \{ x \mapsto z : 2, y \mapsto a : z \}. \]

We have \( \sigma_1(a : x) = 2 : \text{empty} =_{\text{AU}} 2 =_{\text{AU}} \text{empty} : 2 = \sigma_1(y : 2) \) and \( \sigma_2(a : x) = a : z : 2 = \sigma_2(y : 2) \). Other unifiers such as \( \sigma_3 = \{ x \mapsto 2, y \mapsto a \} \) are instances of \( \sigma_2 \).

### 3.2 Unification Algorithm

We start with some notational conventions for the rest of this section:

- \( L, M \) stand for simple expressions,
- \( x, y, z \) stand for variables of any type (unless otherwise specified),
- \( a, b \) stand for simple string or integer expressions, or atom variables,
- \( s, t \) stand for simple string or integer expressions, or atom variables, or list variables,
- the symbol \( \cup \) denotes multiset union.

**Preprocessing.** Given a unification problem \( P \), we rewrite the terms in \( P \) using the rules

\[ L : \text{empty} \rightarrow L \quad \text{and} \quad \text{empty} : L \rightarrow L \]

where \( L \) ranges over list expressions. These reduction rules are applied exhaustively before any of the transformation rules. For example,

\[ x : \text{empty} : 1 : \text{empty} \rightarrow x : 1 : \text{empty} \rightarrow x : 1. \]

We call this process normalization. In addition, the rules are applied to each instance of a transformation rule (that is, once the formal parameters have been replaced with actual parameters) before it is applied, and also after each transformation rule application.
**Transformation rules.** Figure 5 shows the transformation rules, the essence of our approach, in an inference system style where each rule consists of a premise and a conclusion.

- **Remove:** deletes trivial equations
- **Decomp:** replaces equations between list expressions by equations between their subexpressions
- **Subst1:** propagates a solved variable to the rest of the problem
- **Subst2:** assigns \texttt{empty} to a list variable
- **Subst3:** assigns an atom prefix and a fresh list variable to a list variable
- **Orient1/2:** move variables to left-hand side
- **Orient3:** moves variables of larger type to left-hand side

The rules induce a transformation relation $\Rightarrow$ on unification problems. In order to apply any of the rules to a problem $P$, the problem part of its premise needs to be matched onto $P$. Subsequently, the boolean condition of the premise is checked and the rule instance is normalized so that its premise is identical to $P$.

For example, the rule Orient3 can be matched to $P = \{a : 2 = 1, a = 3\}$ (where $a$ and $1$ are variables of type atom and list, respectively) by setting $y \mapsto a, x \mapsto 1, L \mapsto 1, M \mapsto \texttt{empty}$ and $P \mapsto \{a = 3\}$. The rule instance is then

$$\begin{align*}
\{a : 2 = 1 : \texttt{empty}\} & \cup \{a = 3\} \\
\{1 : \texttt{empty} = a : 2\} & \cup \{a = 3\}
\end{align*}$$

Orient3

which gets normalized to

$$\begin{align*}
\{a : 2 = 1\} & \cup \{a = 3\} \\
\{1 = a : 2\} & \cup \{a = 3\}
\end{align*}$$

Orient3

whose conclusion is the result of applying Orient3 to $P$.

Showing a unification problem cannot be unified can be a lengthy affair because we need to compute all normal forms with respect to $\Rightarrow$. Instead, the rules Occur and Clash1-4, shown in Figure 6, introduce failure. Failure cuts off parts of the search tree for a given problem $P$. This is because if $P \Rightarrow \text{fail}$, then $P$ has no unifiers and it is not necessary to compute a normal form. Effectively, the failure rules have precedence over the other rules. They are justified by the following lemmata.

**Lemma 1.** A normalised equation $x = L$ has no solution if $L$ is a simple expression, $x \in \text{Var}(L)$, $\text{type}(x) = \text{list}$ and $x \neq L$.

**Proof.** Since $x \in \text{Var}(L)$ and $x \neq L$, $L$ is of the form $s_1 : s_2 : \ldots : s_n$ with $n \geq 2$ and $x \in \text{Var}(s_i)$ for some $1 \leq i \leq n$. As $L$ is normalised, none of the terms $s_i$ contains the constant \texttt{empty}. Also, since $L$ is simple, it contains no list variables other than $x$ and $x$ is not repeated. It follows $\sigma(x) \neq \text{AU} \sigma(L)$ for every substitution $\sigma$. \qed
\[
\begin{align*}
\{L = L\} \cup P & \quad \text{Remove} \\
\{s : L = t : M\} \cup P & \quad L \not= \text{empty} \quad \text{Decomp} \\
\{x = L\} \cup P & \quad x \in \text{Var}(P) \quad x \not\in \text{Var}(L) \quad \text{type}(x) \geq \text{type}(L) \\
\{x = L\} \cup P & \quad x \mapsto L \\
\{x : L = M\} \cup P & \quad L \not= \text{empty} \quad \text{type}(x) = \text{list} \quad \text{Subst1} \\
\{x : L = M\} \cup P & \quad L \not= \text{empty} \quad z \text{ is a fresh list variable} \quad \text{type}(x) = \text{list} \quad \text{Subst2} \\
\{a : L = x : M\} \cup P & \quad a \text{ is not a variable} \\
\{y : L = x\} \cup P & \quad L \not= \text{empty} \quad \text{type}(x) = \text{type}(y) \quad \text{Orient1} \\
\{y : L = x\} \cup P & \quad \text{type}(y) < \text{type}(x) \\
\{y : L = x : M\} \cup P & \quad \text{Orient3}
\end{align*}
\]

Fig. 5. Transformation rules

**Lemma 2.** Equations of the form \(a : L = \text{empty}\) or \(\text{empty} = a : L\) have no solution if \(a\) is an atom expression.

**Lemma 3.** An equation \(a : L = b : M\) has no solution if \(a \not= b\) are atom expressions without variables.

**The algorithm.** The algorithm in Figure 7 starts by normalizing the input problem, as explained above. It uses a queue of unification problems to search the derivation tree of \(P\) with respect to \(\Rightarrow\) in a breadth-first manner. The first step is to put the normalized problem \(P\) on the queue.

The variable next holds the head of the queue. If next is in solved form, then next (see Definition 6) is a unifier of the original problem and is added to the set \(U\) of solutions. Otherwise, the next step is to construct all problems \(P'\) such
\[
\begin{align*}
\{x = L\} \cup P & \quad x \in \text{Var}(L) \quad x \neq L \quad \text{type}(x) = \text{list} \\
\text{fail} & \quad \text{Occur} \\
\{a : L = b : M\} \cup P & \quad a \neq b \quad \text{Var}(a) = \emptyset = \text{Var}(b) \\
\text{fail} & \quad \text{Clash1} \\
\{a : L = \text{empty}\} \cup P & \quad \text{Clash2} \\
\{\text{empty} = a : L\} \cup P & \quad \text{Clash3} \\
\{x = L\} \cup P & \quad \text{type}(x) \parallel \text{type}(L) \\
\text{fail} & \quad \text{Clash4}
\end{align*}
\]

Fig. 6. Failure rules

that \textit{next} \Rightarrow P'. If \(P'\) is \textit{fail}, then the derivation tree below \textit{next} is ignored, otherwise \(P'\) gets normalized and enqueued.

An example tree traversed by the algorithm is shown in Figure 8. Nodes are labelled with unification problems and edges represent applications of transformation rules. The root of the tree is the problem \(\{a : x = y : 2\}\) to which the rules \textit{Decomp} and \textit{Orient3} can be applied. The two resulting problems form the second level of the search tree and are processed in turn. Eventually, the unifiers

\[
\begin{align*}
\sigma_1 &= \{x \mapsto 2, y \mapsto a\} \\
\sigma_2 &= \{x \mapsto z : 2, y \mapsto a : z\} \\
\sigma_3 &= \{a \mapsto 2, x \mapsto \text{empty}, y \mapsto \text{empty}\}
\end{align*}
\]

are found, which represent a complete set of unifiers of the initial problem. Note that the set is not minimal because \(\sigma_1\) is an instance of \(\sigma_2\).

The algorithm is similar to the A-unification (word unification) algorithm presented in [9] which looks only at the head of an equation. That algorithm terminates for the special case that the input problem has no repeated variables, and is sound and complete. Our approach can be seen as an extension from A-unification to AU-unification, to handle the unit equations, and presented in the rule-based style of [1]. In addition, our algorithm deals with GP’s subtype system.
Unify(P) : 

\[ U := \emptyset \]

create empty queue Q of unification problems
normalize P
Q.enqueue(P)
while Q is not empty
    next := Q.dequeue()
    if next is in solved form
        U := U ∪ \{ next \}
    else if next ↦ fail
        foreach P' such that next ↦ P'
            normalize P'
            Q.enqueue(P')
        end foreach
    end if
end while
return U

Fig. 7. Unification algorithm

3.3 Termination and Soundness

We show that the unification algorithm terminates if the input problem contains no repeated list variables, where termination of the algorithm follows from termination of the relation ⇒.

We first demonstrate that the algorithm need not terminate on unification problems with repeated list variables. A counterexample is the unification problem \{x:1 = 1:x\} which initiates the following infinite sequence:

\[
\begin{align*}
\{x:1 = 1:x\} & \Rightarrow Subst3 \{x = 1:z_1, z_1:1 = x\} \\
& \Rightarrow Subst1 \{x = 1:z_1, z_1:1 = 1:z_1\} \\
& \Rightarrow Subst3 \{x = 1:z_1, z_1 = 1:z_2, z_2:1 = z_1\} \\
& \Rightarrow Subst1 \{x = 1:1:z_2, z_1 = 1:z_2, z_2:1 = 1:z_2\} \\
& \Rightarrow Subst3 \{x = 1:1:z_2, z_1 = 1:z_2, z_2 = 1:z_3, z_3:1 = z_2\} \\
& \Rightarrow Subst1 \{x = 1:1:1:z_3, z_1 = 1:1:z_3, z_2 = 1 : z_3, z_3:1 = 1 : z_3\} \\
& \Rightarrow Subst3 \ldots
\end{align*}
\]

Note that \{x:1 = 1:x\} has an infinite number of solutions that are mutually incomparable: \{x ↦ empty\}, \{x ↦ 1\}, \{x ↦ 1:1\}, . . . We remark that the A-unification algorithm of [9] also diverges on this problem.

To prove that the transformation relation ⇒ terminates on problems without repeated list variables, we need to consider an invariant which is implied by this property. This is because rule Subst3 introduces a repeated list variable. We
say that a unification problem $P$ satisfies the repeated variable condition if for every list variable $x$, the subproblem $P \setminus \{y = L \mid y \neq x\}$ contains at most one occurrence of $x$.

**Lemma 4 (Invariance).** For each transformation $P \Rightarrow P'$ where $P$ satisfies the repeated variable condition, $P'$ also satisfies this condition.

The proof is by a careful but straightforward inspection of all rules in Figure 5. We are now ready to state our termination result.

**Theorem 1 (Termination).** If $P$ is a unification problem without repeated list variables, then there is no infinite sequence $P \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \ldots$

**Proof.** Define the size $|L|$ of an expression $L$ by

- 0 if $L = \text{empty}$,
- 1 if $L$ is an expression of category Atom (see Figure 4) or a list variable,
- $|M| + |N| + 1$ if $L = M : N$.

We define a lexicographic termination order by assigning to a unification problem $P$ the tuple $(n_1, n_2, n_3, n_4, n_5, n_6)$, where
- $n_1$ is the number of unsolved variables in $P$;
- $n_2$ is the size of $P \setminus Q$ where $Q = \{ x = L \mid x \in \text{Var} \}$, that is, $n_2 = \sum_{(L = R) \in P \setminus Q}(|L| + |R|)$;
- $n_3$ is the size of $P$, that is, $n_3 = \sum_{(L = R) \in P}(|L| + |R|)$;
- $n_4$ is the number of equations $y:L = x : M$ in $P$ where $L$ is not a variable;
- $n_5$ is the number of equations $y:L = x : M$ in $P$ where $\text{type}(x) = \text{type}(y)$ and $L \neq \emptyset$;
- $n_6$ is the number of equations $y:L = x : M$ in $P$ where $\text{type}(y) < \text{type}(x)$.

The table in Figure 9 shows that for each transformation step $P \Rightarrow P'$, the tuple associated with $P'$ is strictly smaller than the tuple associated with $P$ in the lexicographic order induced by the components $n_1$ to $n_6$.

<table>
<thead>
<tr>
<th></th>
<th>$n_1$</th>
<th>$n_2$</th>
<th>$n_3$</th>
<th>$n_4$</th>
<th>$n_5$</th>
<th>$n_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subst1</td>
<td>&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subst3</td>
<td>≥</td>
<td>&lt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subst2</td>
<td>≥</td>
<td>≥</td>
<td>&lt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decomp</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>&lt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>Orient1</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>&gt;</td>
</tr>
<tr>
<td>Orient2</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
</tr>
<tr>
<td>Orient3</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
<td>≥</td>
</tr>
</tbody>
</table>

Fig. 9. Lexicographic termination order

For most rules, the table entries are easy to check. The argument why Subst3 decreases $n_2$ is a bit more involved because the rule solves $x$ and creates two copies of the fresh variable $z$. Since the repeated variable condition of Lemma 4 is an invariant, the instance of problem $P$ in the premise of Subst3 can only contain occurrences of $x$ that appear on the right-hand side of equations $y:L$ with $y \neq x$. It follows that the $n_2$-value of $P$ is the same as that of $P\{x \mapsto b : z\}$. As a consequence, each application of Subst3 decreases $n_2$ by 2.

In order to show that the unification algorithm is sound, we need some preliminary lemmata.

**Lemma 5.** If $P = \{ x_1 =^? t_1, \ldots, x_n =^? t_n \}$ is in solved form then for all $\sigma \in \mathcal{U}(P)$, $\sigma = \overline{P}\sigma$.

**Proof.** We show that $\sigma$ and $\overline{P}\sigma$ behave the same on all variables by considering the following cases:

1. $x \in \{x_1, \ldots, x_n\}$, i.e. $x = x_k$, then $x_k\overline{P} = t_k$ which implies $x_k\overline{P}\sigma = t_k\sigma$.
   Also since $\sigma$ is a unifier of $P$, then $x_k\sigma = t_k\sigma$. Therefore $x_k\overline{P}\sigma = x_k\sigma$.  


2. If $x \notin \{x_1, \ldots, x_n\}$, then $x \overrightarrow{P} = x$ for variables outside of the domain and hence $x \overrightarrow{P} \sigma = x \sigma$.

\[ \square \]

**Lemma 6.** If $P$ is in solved form then $\overrightarrow{P}$ is an idempotent most general unifier of $P$.

**Proof.** Since none of the $x_i$ occur in any of the $t_k$’s, we get that $\text{Dom}(\overrightarrow{P}) \cap \text{V Ran}(\overrightarrow{P}) = \emptyset$. Therefore, $\overrightarrow{P}$ is idempotent. Also, we have $x_i \overrightarrow{P} = t_i = t_i \overrightarrow{P}$ for the same reason, hence $\overrightarrow{P} \in \mathcal{U}(P)$. Finally, $\overrightarrow{P}$ is most general because $\overrightarrow{P} \leq \sigma$ for all $\sigma \in \mathcal{U}(P)$ by Lemma 5.

\[ \square \]

**Lemma 7.** If $P \Rightarrow P'$, then $\mathcal{U}(P) \supseteq \mathcal{U}(P')$

**Proof.** For Remove, Decom and Orient, this is obvious.

For Subst1, let $\theta = \{x \mapsto L\}$. By applying Lemma 5 to $\{x = L\}$ which is in solved form, we get that $\sigma = \theta \sigma$ if $x \sigma = L \sigma$

\begin{align*}
\sigma \in \mathcal{U}(\{x = \text{empty}, L = M\} \cup P\theta) & \iff x \sigma = L \sigma \land \sigma \in \mathcal{U}(P\theta) \\
& \iff x \sigma = L \sigma \land \theta \sigma \in \mathcal{U}(P) \\
& \iff x \sigma = L \sigma \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(x \in \{x = \text{empty}, L = M\} \cup P) \\
& \iff \sigma = \emptyset \sigma \land \sigma \in \mathcal{U}(P)
\end{align*}

For Subst2, the argument is similar. Let $\theta = \{x \mapsto \text{empty}\}$. Then $\sigma = \theta \sigma$ if $x \sigma = \text{empty, empty} \sigma$ where $\sigma = \emptyset \sigma$ if $x \sigma = \emptyset \sigma (\text{empty})$.

For Subst3, let $\theta = \{x \mapsto b : z\}$. Then $\sigma = \theta \sigma$ if $x \sigma = \emptyset \sigma (\text{empty})$, again by Lemma 5

\begin{align*}
\sigma \in \mathcal{U}(\{x = b : z, z : L = M\} \cup P\theta) & \iff \sigma(z \in \{x = b : z, z : L = M\} \cup P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \sigma(b : z) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \emptyset \sigma (\text{empty}) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \emptyset \sigma (\text{empty}) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \emptyset \sigma (\text{empty}) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \emptyset \sigma (\text{empty}) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \emptyset \sigma (\text{empty}) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \emptyset \sigma (\text{empty}) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \emptyset \sigma (\text{empty}) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(z : L) = M \sigma \land x \sigma = \emptyset \sigma (\text{empty}) \land \sigma \in \mathcal{U}(P) \\
& \iff \sigma(x \in \{x : L = b : M\} \cup P) \\
& \iff \sigma \in \mathcal{U}(\{x : L = b : M\} \cup P)
\end{align*}

\[ \square \]

**Theorem 2 (Soundness).** If $P \Rightarrow P'$ with $P'$ in solved form, then $\overrightarrow{P'}$ is an idempotent unifier of $P$. 

\[ \square \]
Proof. Note that $\overrightarrow{P}$ unifies $P'$ because it is idempotent (by Lemma 6); a simple induction with Lemma 7 shows that $\overrightarrow{P}$ must be a unifier of $P$. \hfill \Box

4 Conclusion

This paper presents groundwork for a static confluence analysis of GP programs. We have constructed a rule-based unification algorithm for systems of equations with left-hand expressions of rule schemata, and have shown that the algorithm always terminates and is sound.

Future work includes proving that our unification algorithm always delivers a complete set of solutions, that is, that every unifier of the input problem is an instance of some unifier in the computed set of solutions. Next, to establish a Critical Pair Lemma in the sense of [6], a notion of independent rule schema applications has to be developed, as well as restriction and embedding theorems for derivations with rule schemata. In addition, since critical pairs contain graphs labelled with expressions, checking joinability of critical pairs will require sufficient conditions under which equivalence of expressions can be decided. This is because the theory of GP’s label algebra includes the undecidable theory of Peano arithmetic.

References
