Electronic Communications of the EASST
Volume 71 (2015)

# Graph Computation Models
# Selected Revised Papers from GCM 2014

## A Unification Algorithm for GP 2

Ivaylo Hristakiev and Detlef Plump

17 pages

Guest Editors: Rachid Echahed, Annegret Habel, Mohamed Mosbah
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# A Unification Algorithm for GP 2

**Ivaylo Hristakiev*** **and Detlef Plump**

The University of York, UK

**Abstract:** The graph programming language GP 2 allows to apply sets of rule schemata (or "attributed" rules) non-deterministically. To analyse conflicts of programs statically, graphs labelled with expressisons are overlayed 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, sound and complete. For every input equation, the algorithm generates a finite set of substitutions. Soundness means that each of these substitutions solves the input equation. Since GP labels are lists constructed by concatenation, unification modulo associativity and unit law is required. This problem, which is also known as *word unification*, is infinitary in general but becomes finitary due to GP's rule schema syntax and the assumption that rule schemata are left-linear. Our unification algorithm is complete in that every solution of an input equation is an instance of some substitution in the generated set.

**Keywords:** graph programs, word unification, critical pair analysis

## 1 Introduction

A common programming pattern in the graph programming language GP 2 [Plu12, BFPR15] 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 non-deterministic, different graphs may result from the loop. Thus, if the programmer wants the loop to implement a function, a static analysis that establishes or refutes functional behaviour would be desirable.

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 [Plu05] 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 [HKT02, EEPT06, GLEO12], 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 one cannot use syntactic unification (as in logic programming) when constructing critical pairs and checking their joinability, but has to take into account all equations valid in the attribute algebra.

For example, [HKT02] presents a method of constructing critical pairs in the case where the equational theory of the attribute algebra is represented by a confluent and terminating term

* This author's work is supported by a PhD scholarship of the UK Engineering and Physical Sciences Research Council (EPSRC)

rewriting system. The algorithm first computes normal forms of the attributes of overlayed nodes and subsequently constructs the most general (syntactic) unifier of the normal forms. This has been shown to be incomplete [EEPT06, 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 algebra's 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 `init` and `add` append distances

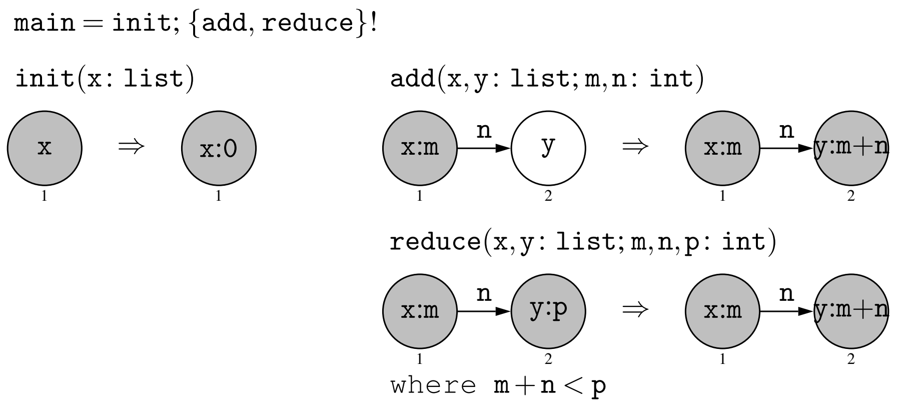
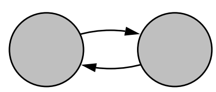



Figure 1: A program calculating shortest distances

to the labels of nodes that have not been visited before, while `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 `add` and `reduce`, their left-hand sides are overlayed. For example, the structure of the left-hand graph of `reduce` can match the following structure in two different ways:

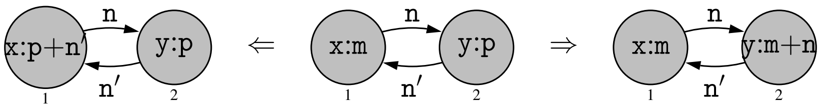


Consider a copy of `reduce` in which the variables have been renamed to $\mathtt{x}'$, $\mathtt{m}'$, etc. To match `reduce` and its copy differently requires solving the system of equations $\langle \mathtt{x}{:}\mathtt{m} =^? \mathtt{y}'{:}\mathtt{p}', \mathtt{y}{:}\mathtt{p} =^? \mathtt{x}'{:}\mathtt{m}' \rangle$. 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

$$\sigma = \{\mathtt{x}' \mapsto \mathtt{y}, \mathtt{m}' \mapsto \mathtt{p}, \mathtt{y}' \mapsto \mathtt{x}, \mathtt{p}' \mapsto \mathtt{m}\}$$

is a most general solution. It gives rise to the following critical pair:[1]

[1] For simplicity, we ignore the condition of `reduce`.




In general though, equations can arise that have several independent solutions. For example, the equation $\langle \mathtt{n{:}x} =^? \mathtt{y{:}2}\rangle$ (with n of type int and x,y of type list) has the minimal solutions

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

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

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

This problem is similar to *word unification* [BS01], which attempts to solve equations modulo associativity. (Some authors consider *AU*-unification as word unification, e.g. [Jaf90]). Solvability of word unification is decidable, albeit in PSPACE [Pla99], but there is not always a finite complete set of solutions. The same holds for AU-unification (see Section 3). Fortunately, GP's syntax for left-hand sides of rule schemata forbids labels with more than one list variable. It turns out that by additionally forbidding shared list variables between different left-hand labels of a rule, rule 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 equations between 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 equation. Moreover, the algorithm is complete in that every unifier of the input equation is an instance of some unifier in the computed set of solutions.

## 2 GP Rule Schemata

We refer to [Plu12, BFPR15] 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 $\mathscr{C}$ is a system $G = (V,E,s,t,l,m)$, where $V$ and $E$ are finite sets of *nodes* (or *vertices*) and *edges*, $s,t\colon E \to V$ are the *source* and *target* functions for edges, $l\colon V \to \mathscr{C}$ is the node labelling function and $m\colon E \to \mathscr{C}$ is the edge labelling function. We write $\mathscr{G}(\mathscr{C})$ for the class of all graphs over $\mathscr{C}$.

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. For example, all labels in Figure 2 are list expressions.

Figure 4 gives a grammar in Extended Backus-Naur Form defining the abstract syntax of labels. The function length return the length of a variable, while indeg and outdeg access the indegree resp. outdegree of a left-hand node in the host graph.

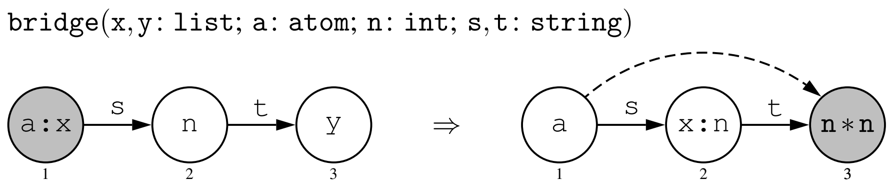



Figure 2: Declaration of a GP rule schema

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 $\mathrm{Var} = \mathrm{IVar} \cup \mathrm{SVar} \cup \mathrm{AVar} \cup \mathrm{LVar}$. The mark components of labels are represented graphically rather than textually.

Each expression $l$ has a unique smallest type, denoted by $\mathrm{type}(l)$, which can be read off the

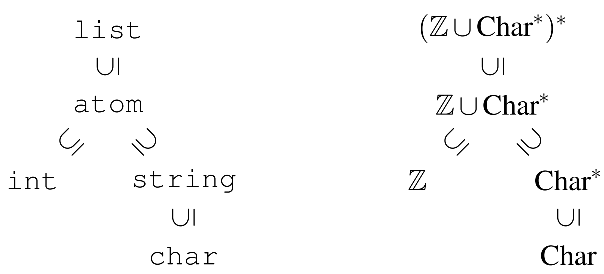



Figure 3: Subtype hierarchy for labels

| | | |
|---|---|---|
| Integer | ::= | Digit {Digit} \| IVar |
| | | \| '−' Integer \| Integer ArithOp Integer |
| | | \| `length` '(' LVar \| AVar \| SVar ')' |
| | | \| (`indeg` \| `outdeg`) '(' Node ')' |
| ArithOp | ::= | '+' \| '−' \| '∗' \| '/' |
| String | ::= | ' " ' {Char} ' " ' \| SVar \| String '.' String |
| Atom | ::= | Integer \| String \| AVar |
| List | ::= | `empty` \| Atom \| LVar \| List ':' List |
| Label | ::= | List [Mark] |
| Mark | ::= | `red` \| `green` \| `blue` \| `grey` \| `dashed` \| `any` |

Figure 4: Abstract syntax of rule schema labels

hierarchy in Figure 3 after $l$ has been normalised with the rewrite rules shown at the beginning of Subsection 3.2. We write $\mathrm{type}(l_1) < \mathrm{type}(l_2)$ or $\mathrm{type}(l_1) \leq \mathrm{type}(l_2)$ to compare types according to the subtype hierarchy. If the types of $l_1$ and $l_2$ are incomparable, we write $\mathrm{type}(l_1) \parallel \mathrm{type}(l_2)$.

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, no length or degree operators, no string concatenation, and at most one occurrence of a list variable.

In other words, simple expressions contain no unary or binary operators except list concatenation, and at most one occurrence of a list variable. For example, given the variable declarations of Figure 2, $\mathtt{a:x}$ and $\mathtt{y:n:n}$ are simple expressions whereas $\mathtt{n} * 2$ or $\mathtt{x:y}$ are not simple.

Our definition of simple expressions is more restrictive than that in [Plu12] because we exclude string concatenation and the unary minus. These operations (especially string concatenation) would considerably inflate the unification algorithm and its completeness proof, without posing a substantial challenge.

**Definition 2** (Rule schema) A *rule schema* $\langle L, R, I\rangle$ consists of graphs $L,R$ in $\mathscr{G}(\mathrm{Label})$ and a set $I$, the *interface*, such that $I = V_L \cap V_R$. 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.

**Assumption 1** (Left-linearity)**.** We assume that rule schemata $\langle L, R, I\rangle$ are *left-linear*, that is, the labels of different nodes or edges in $L$ do not contain the same list variable.

This assumption is necessary to ensure that the solutions of the equations resulting from overlaying two rule schemata can be represented by a finite set of unifiers. For example, without this assumption it is easy to construct two rule schemata that induce the system of equations $\langle \mathtt{x} : \mathtt{1} =^? \mathtt{y},\ \mathtt{y} =^? \mathtt{1} : \mathtt{x}\rangle$. This system has solutions $\{\mathtt{x} \mapsto \mathtt{empty}, \mathtt{y} \mapsto \mathtt{1}\}$, $\{\mathtt{x} \mapsto \mathtt{1}, \mathtt{y} \mapsto \mathtt{1} : \mathtt{1}\}$, $\{\mathtt{x} \mapsto \mathtt{1} : \mathtt{1}, \mathtt{y} \mapsto \mathtt{1} : \mathtt{1} : \mathtt{1}\}, \ldots$ which form a infnite, minimal and compete set of solutions (See Definition 5 below).

# 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 $\sigma_I\colon \mathrm{IVar} \to \mathrm{Integer}$, $\sigma_S\colon \mathrm{SVar} \to \mathrm{String}$, $\sigma_A\colon \mathrm{AVar} \to \mathrm{Atom}$, $\sigma_L\colon \mathrm{LVar} \to \mathrm{List}$. Here Integer, String, Atom and List are the sets of expressions defined by the GP label grammar of Figure 4. For example, if $\mathtt{z} \in \mathrm{LVar}$, $\mathtt{x} \in \mathrm{IVar}$ and $\mathtt{y} \in \mathrm{SVar}$, then we write $\sigma = \{\mathtt{x} \mapsto \mathtt{x}+\mathtt{1}, \mathtt{z} \mapsto \mathtt{y} : -\mathtt{x} : \mathtt{y}\}$ for the substitution that maps $\mathtt{x}$ to $\mathtt{x}+\mathtt{1}$, $\mathtt{z}$ to $\mathtt{y} : -\mathtt{x} : \mathtt{y}$ 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, $(\mathtt{z} : -\mathtt{x})\sigma = \mathtt{y} : -\mathtt{x} : \mathtt{y} : -(\mathtt{x}+\mathtt{1})$.

By $\mathrm{Dom}(\sigma)$ we denote the set $\{x \in \mathrm{Var} \mid \sigma(x) \neq x\}$ and by $\mathrm{VRan}(\sigma)$ the set of variables occurring in the expressions $\{\sigma(x) \mid x \in \mathrm{Var}\}$. A substitution $\sigma$ is *idempotent* if $\mathrm{Dom}(\sigma) \cap \mathrm{VRan}(\sigma) = \emptyset$. The *composition* of two substitutions $\sigma$ and $\theta$, is defined as $x(\sigma \circ \theta) = \begin{cases} (x\sigma)\theta & \text{if } x \in \mathrm{Dom}(\sigma) \\ x\theta & \text{otherwise} \end{cases}$ and is an associative operation.

**Definition 3** (Unification problem) A *unification problem* is a pair of an equation and a substitution

$$P = \langle s =^? t, \sigma_P\rangle$$

where $s$ and $t$ are simple list expressions without common variables.

The symbol $=^?$ signifies that the equation must be *solved* rather than having to hold for all values of variables. The purpose of $\sigma_P$ is for the unification algorithm (Section 3.2) to record a partial solution. An illustration of this concept will be seen in Figure 8.

In Section 2, we already assumed that GP rule schemata need to be left-linear. Now, the problem of solving a system of equations $\{s_1 = t_1, s_2 = t_2\}$ can be broken down to solving individual equations and combining the answers - if $\sigma_1$ and $\sigma_2$ are solutions to each individual equation, then $\sigma_1 \cup \sigma_2$ is a solution to the combined problem as $\sigma_1$ and $\sigma_2$ do not share variables.

Consider the equational axioms for associativity and unity,

$$\mathrm{AU} = \{\mathtt{x} : (\mathtt{y} : \mathtt{z}) = (\mathtt{x} : \mathtt{y}) : \mathtt{z},\ \mathtt{empty} : \mathtt{x} = \mathtt{x},\ \mathtt{x} : \mathtt{empty} = \mathtt{x}\}$$

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

**Definition 4** (Unifier) Given a unification problem $P = \langle s =^? t, \sigma_P\rangle$ a *unifier* of $P$ a is a substitution $\theta$ such that

$$s\theta =_{\mathrm{AU}} t\theta \text{ and } x_i\theta =_{\mathrm{AU}} t_i\theta$$

for each binding $\{\mathtt{x_i} \mapsto \mathtt{t_i}\}$ in $\sigma_P$ .

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

The special unification problem `fail` represents failure and has no unifiers. A problem $P = \langle s =^? t, \varnothing\rangle$ is *initial* and $P = \langle \varnothing, \sigma_P\rangle$ is *solved*.

A substitution $\sigma$ is *more general* on a set of variables $X$ than a substitution $\theta$ if there exists a substitution $\lambda$ such that $x\theta =_{\mathrm{AU}} x\sigma\lambda$ for all $x \in X$. In this case we write $\sigma \leqq_X \theta$ 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 $\sigma \leqq_X \theta$ and $\theta \leqq_X \sigma$.

**Definition 5** (Complete set of unifiers [Plo72]) A set $\mathscr{C}$ of substitutions is a *complete set of unifiers* of a unification problem $P$ if

1. (Soundness) $\mathscr{C} \subseteq \mathscr{U}(P)$, that is, each substitution in $\mathscr{C}$ is a unifier of $P$, and
2. (Completeness) for each $\theta \in \mathscr{U}(P)$ there exists $\sigma \in \mathscr{C}$ such that $\sigma \leqq_X \theta$, where $X = \mathrm{Var}(P)$ is the set of variables occurring in $P$.

Set $\mathscr{C}$ is also *minimal* if each pair of distinct unifiers in $\mathscr{C}$ are incomparable with respect to $\leqq_X$.

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

For simplicity, we replace $=^?$ with $=$ in unification problems from now on.

*Example* 1 The minimal complete set of unifiers of the problem $\langle \mathtt{a} : \mathtt{x} = \mathtt{y} : 2\rangle$ (where a is an atom variable and x,y are list variables) is $\{\sigma_1, \sigma_2\}$ with

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

We have $\sigma_1(\mathtt{a} : \mathtt{x}) = 2 : \mathtt{empty} =_{\mathrm{AU}} 2 =_{\mathrm{AU}} \mathtt{empty} : 2 = \sigma_1(\mathtt{y} : 2)$ and $\sigma_2(\mathtt{a} : \mathtt{x}) = \mathtt{a} : \mathtt{z} : 2 = \sigma_2(\mathtt{y} : 2)$. Other unifiers such as $\sigma_3 = \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto \mathtt{a}\}$ 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,
- $\mathtt{x}, \mathtt{y}, \mathtt{z}$ stand for variables of any type (unless otherwise specified),
- $\mathtt{a}, \mathtt{b}$ stand for
  - (i) simple string or integer expressions, or
  - (ii) string, integer or atom variables
- $\mathtt{s}, \mathtt{t}$ stand for
  - (i) simple string or integer expressions, or
  - (ii) variables of any type

**Preprocessing.** Given a unification problem $P = \langle s =^? t, \sigma \rangle$, we rewrite the terms in $s$ and $t$ using the reduction rules

$$L : \texttt{empty} \to L \quad \text{and} \quad \texttt{empty} : L \to L$$

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

$$\texttt{x} : \texttt{empty} : \texttt{1} : \texttt{empty} \to \texttt{x} : \texttt{1} : \texttt{empty} \to \texttt{x} : \texttt{1}.$$

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 |
| Decomp1: | syntactically equates a list variable with an atom expression or list variable |
| Decomp1': | syntactically equates an atom variable with an expression of lesser type |
| Decomp2/2': | assigns a list variable to start with another list variable |
| Decomp3: | removes identical symbols from the head |
| Decomp4: | solves an atom variable |
| Subst1: | solves a variable |
| Subst2: | assigns `empty` to a list variable |
| Subst3: | assigns an atom prefix to a list variable |
| Orient1/2: | moves variables to left-hand side |
| Orient3: | moves variables of larger type to left-hand side |
| Orient4: | moves a list variable to the 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 $P = \langle \texttt{a} : 2 = \texttt{m},\ \sigma \rangle$ (where a and m are variables of type `atom` and `list`, respectively) by setting $\texttt{y} \mapsto \texttt{a}$, $\texttt{x} \mapsto \texttt{m}$, $M \mapsto 2$, and $L \mapsto \texttt{empty}$. The rule instance and its normal form are then

$$\frac{\langle \texttt{a} : 2 = \texttt{m} : \texttt{empty}, \sigma \rangle}{\langle \texttt{m} : \texttt{empty} = \texttt{a} : 2, \sigma \rangle} \quad \text{and} \quad \frac{\langle \texttt{a} : 2 = \texttt{m}, \sigma \rangle}{\langle \texttt{m} = \texttt{a} : 2, \sigma \rangle}$$

where the conclusion of the normal form is the result of applying Orient3 to $P$.

$$\frac{\langle L = L, \sigma\rangle}{\langle \varnothing, \sigma\rangle}\ \textsf{Remove}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{s} : M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\langle L = M,\ \sigma \circ \{\mathtt{x} \mapsto \mathtt{s}\}\rangle}\ \textsf{Decomp1}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{a} : M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{a}) \leq \mathrm{type}(\mathtt{x}) \leq \mathtt{atom}}{\langle (L = M)\{\mathtt{x} \mapsto \mathtt{a}\},\ \sigma \circ \{\mathtt{x} \mapsto \mathtt{a}\}\rangle}\ \textsf{Decomp1}'$$

$$\frac{\langle \mathtt{x} : L = \mathtt{y} : M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathtt{list} \quad \mathrm{type}(\mathtt{y}) = \mathtt{list} \quad \mathtt{x}' \text{ is a fresh list variable}}{\langle \mathtt{x}' : L = M, \sigma \circ \{\mathtt{x} \mapsto \mathtt{y} : \mathtt{x}'\}\rangle}\ \textsf{Decomp2}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{y} : M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathtt{list} \quad \mathrm{type}(\mathtt{y}) = \mathtt{list} \quad \mathtt{y}' \text{ is a fresh list variable}}{\langle L = \mathtt{y}' : M, \sigma \circ \{\mathtt{y} \mapsto \mathtt{x} : \mathtt{y}'\}\rangle}\ \textsf{Decomp2}'$$

$$\frac{\langle \mathtt{s} : L = \mathtt{s} : M, \sigma\rangle}{\langle L = M, \sigma\rangle}\ \textsf{Decomp3} \qquad \frac{\langle \mathtt{x} = \mathtt{a} : \mathtt{y}, \sigma\rangle \quad \mathrm{type}(\mathtt{x}) = \mathtt{atom} \quad \mathrm{type}(\mathtt{y}) = \mathtt{list}}{\langle \mathtt{empty} = \mathtt{y}, \sigma \circ \{\mathtt{x} \mapsto \mathtt{a}\}\rangle}\ \textsf{Decomp4}$$

$$\frac{\langle \mathtt{x} = L, \sigma\rangle \quad \mathtt{x} \notin \mathrm{Var}(L) \quad \mathrm{type}(\mathtt{x}) \geq \mathrm{type}(L)}{\langle \varnothing, \sigma \circ \{\mathtt{x} \mapsto L\}\rangle}\ \textsf{Subst1}$$

$$\frac{\langle \mathtt{x} : L = M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\langle L = M, \sigma \circ \{\mathtt{x} \mapsto \mathtt{empty}\}\rangle}\ \textsf{Subst2}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{a} : M\}, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathtt{x}' \text{ is a fresh list variable} \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\langle \mathtt{x}' : L = M, \sigma \circ \{\mathtt{x} \mapsto \mathtt{a} : \mathtt{x}'\}\rangle}\ \textsf{Subst3}$$

$$\frac{\langle \mathtt{a} : M = \mathtt{x} : L, \sigma\rangle \quad \mathtt{a} \text{ is not a variable}}{\langle \mathtt{x} : L = \mathtt{a} : M, \sigma\rangle}\ \textsf{Orient1}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{y}, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathrm{type}(\mathtt{y})}{\langle \mathtt{y} = \mathtt{x} : L, \sigma\rangle}\ \textsf{Orient2}$$

$$\frac{\langle \mathtt{y} : M = \mathtt{x} : L, \sigma\rangle \quad \mathrm{type}(\mathtt{y}) < \mathrm{type}(\mathtt{x})}{\langle \mathtt{x} : L = \mathtt{y} : M, \sigma\rangle}\ \textsf{Orient3} \qquad \frac{\langle \mathtt{empty} = \mathtt{x} : L, \sigma\rangle \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\langle \mathtt{x} : L = \mathtt{empty}, \sigma\rangle}\ \textsf{Orient4}$$

Figure 5: Transformation rules

$$\frac{\langle \mathtt{x} = L, \sigma\rangle \quad \mathtt{x} \in \mathrm{Var}(L) \quad \mathtt{x} \neq L \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\mathrm{fail}}\ \mathsf{Occur} \qquad \frac{\langle \mathtt{a} : L = \mathtt{empty}, \sigma\rangle}{\mathrm{fail}}\ \mathsf{Clash2}$$

$$\frac{\langle \mathtt{a} : L = \mathtt{b} : M, \sigma\rangle \quad \mathtt{a} \neq \mathtt{b} \quad \mathrm{Var}(\mathtt{a}) = \emptyset = \mathrm{Var}(\mathtt{b})}{\mathrm{fail}}\ \mathsf{Clash1} \qquad \frac{\langle \mathtt{empty} = \mathtt{a} : L, \sigma\rangle}{\mathrm{fail}}\ \mathsf{Clash3}$$

$$\frac{\langle \mathtt{x} = L, \sigma\rangle \quad \mathrm{type}(\mathtt{x}) \parallel \mathrm{type}(L)}{\mathrm{fail}}\ \mathsf{Clash4}$$

Figure 6: Failure rules

Showing that a unification problem has no solution can be a lengthy affair because we need to compute all normal forms with respect to $\Rightarrow$. Instead, the rules Occur and Clash1 to Clash4, shown in Figure 6, introduce *failure*. Failure cuts off parts of the search tree for a given problem $P$. This is because if $P \Rightarrow \mathrm{fail}$, then $P$ has no unifiers and it is not necessary to compute another normal form. Effectively, the failure rules have precedence over the other rules. They are justified by the following lemmata.

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

*Proof.* Since $\mathtt{x} \in Var(L)$ and $\mathtt{x} \neq L$, $L$ is of the form $s_1 : s_2 : \ldots : s_n$ with $n \geq 2$ and $\mathtt{x} \in Var(s_i)$ for some $1 \leq i \leq n$. As $L$ is normalised, none of the terms $s_i$ contains the constant $\mathtt{empty}$. Also, since $L$ is simple, it contains no list variables other than $\mathtt{x}$ and $\mathtt{x}$ is not repeated. It follows $\sigma(\mathtt{x}) \neq_{\mathrm{AU}} \sigma(L)$ for every substitution $\sigma$. $\square$

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

**Lemma 3** *An equation* $\mathtt{a} : L = \mathtt{b} : M$ *with* $\mathtt{a} \neq \mathtt{b}$ *has no solution if* $\mathtt{a}$ *and* $\mathtt{b}$ *are atom expressions without variables.*

**The algorithm.** The unification algorithm in Figure 7 starts by normalizing the input equation, 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 the form $\langle \varnothing, \sigma\rangle$, then $\sigma$ 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 that `next` $\Rightarrow P'$. If $P'$ is `fail`, then the derivation tree below `next` is ignored, otherwise $P'$ gets normalized and enqueued.

$$
\begin{array}{ll}
\mathsf{Unify}(\mathtt{P}): & \mathtt{U} := \emptyset \\
& \text{create empty queue } \mathtt{Q} \text{ of unification problems} \\
& \text{normalize } \mathtt{P} \\
& \mathtt{Q.enqueue}(\langle \mathtt{P}, \varnothing \rangle) \\
& \mathtt{while}\ \ \mathtt{Q} \text{ is not empty} \\
& \quad \mathtt{next} := \mathtt{Q.dequeue}() \\
& \quad \mathtt{if}\ \ \mathtt{next} \text{ is in the form } \langle \varnothing, \sigma \rangle \\
& \quad\quad \mathtt{U} := \mathtt{U} \cup \{\sigma\} \\
& \quad \mathtt{else\ if}\ \mathtt{next} \not\Rightarrow \mathtt{fail} \\
& \quad\quad \mathtt{foreach}\ \mathtt{P}' \text{ such that } \mathtt{next} \Rightarrow \mathtt{P}' \\
& \quad\quad\quad \text{normalize } \mathtt{P}' \\
& \quad\quad\quad \mathtt{Q.enqueue}(\mathtt{P}') \\
& \quad\quad \mathtt{end\ foreach} \\
& \quad\quad \mathtt{end\ if} \\
& \quad \mathtt{end\ if} \\
& \mathtt{end\ \ while} \\
& \mathtt{return}\ \mathtt{U}
\end{array}
$$

Figure 7: Unification algorithm

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 $\langle \mathtt{y} : 2 = \mathtt{a} : \mathtt{x} \rangle$ to which the rules Decomp1, Subst2 and Subst3 can be applied. The three resulting problems form the second level of the search tree and are processed in turn. Eventually, the unifiers

$$
\begin{array}{lcl}
\sigma_1 & = & \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto \mathtt{a}\} \\
\sigma_2 & = & \{\mathtt{x} \mapsto \mathtt{y}' : 2, \mathtt{y} \mapsto \mathtt{a} : \mathtt{y}'\} \\
\sigma_3 & = & \{\mathtt{a} \mapsto 2, \mathtt{x} \mapsto \mathtt{empty}, \mathtt{y} \mapsto \mathtt{empty}\}
\end{array}
$$

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 [Sch92] which looks at the head of the problem 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 [BS01]. In addition, our algorithm deals with GP's subtype system.

### 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 $\Rightarrow$.

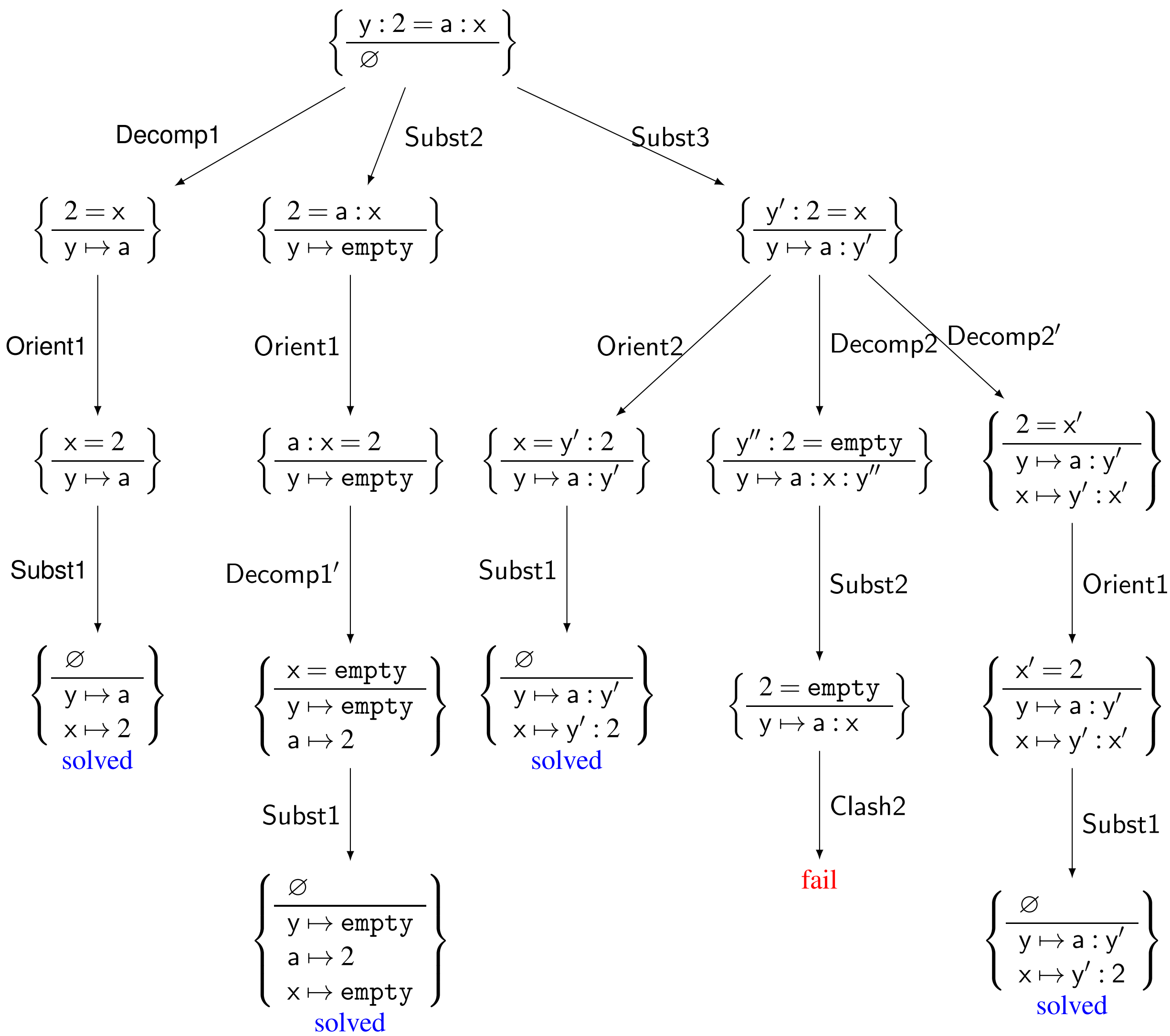



Figure 8: Unification example

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

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

- 0 if $L = \texttt{empty}$,
- 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 $P = \langle L = M, \sigma \rangle$ the tuple $(n_1, n_2, n_3, n_4)$, where

- $n_1$ is the size of $P$, that is, $n_1 = |L| + |M|$;
- $n_2 = \begin{cases} 0 & \text{if } L \text{ starts with a variable} \\ 1 & \text{otherwise} \end{cases}$
- $n_3 = \begin{cases} 1 & \text{if type}(\mathtt{x}) > \text{type}(\mathtt{y}) \\ 0 & \text{otherwise} \end{cases}$

  where x and y are the starting symbols of $L$ and $M$
- $n_4 = |L|$

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 $n_4$.

| | $n_1$ | $n_2$ | $n_3$ | $n_4$ |
|---|---|---|---|---|
| Subst1 − 3 | $>$ | | | |
| Decomp1 − 4 | $>$ | | | |
| Remove | $>$ | | | |
| Orient1 | $=$ | $>$ | | |
| Orient4 | $=$ | $>$ | | |
| Orient3 | $=$ | $=$ | $>$ | |
| Orient2 | $=$ | $=$ | $=$ | $>$ |

Figure 9: Lexicographic termination order

For most rules, the table entries are easy to check. All the rules except for Orient decrease the size of the input equation $s =^? t$. The Orient rules keep the size equal, but move variables and expressions of a bigger type to the left-hand side. □

**Lemma 4** *If $P \Rightarrow P'$, then $\mathscr{U}(P) \supseteq \mathscr{U}(P')$ .*

*Proof.* We show that for each transformation rule, a unifier $\theta$ of the conclusion unifies the premise.

Note that for Remove, Decomp3 and Orient1-4, we have that $\sigma_{P'} = \sigma_P$ .

- Remove

  $$\begin{aligned} \theta \in \mathscr{U}(\langle \varnothing, \sigma_{P'} \rangle) &\iff \theta \in \mathscr{U}(\langle \varnothing, \sigma_P \rangle) \\ &\iff \theta \in \mathscr{U}(\langle \varnothing, \sigma_P \rangle) \wedge L\theta = L\theta \\ &\iff \theta \in \mathscr{U}(\langle L = L, \sigma_P \rangle) \end{aligned}$$

- Decomp3

  $$\begin{aligned} \theta \in \mathscr{U}(\langle L = M, \sigma_{P'} \rangle) &\iff \theta \in \mathscr{U}(\langle L = M, \sigma_P \rangle) \\ &\iff \theta \in \mathscr{U}(\langle L = M, \sigma_P \rangle) \wedge \mathtt{s}\theta = \mathtt{s}\theta \\ &\iff \theta \in \mathscr{U}(\langle \mathtt{s} : L = \mathtt{s} : M, \sigma_P \rangle) \end{aligned}$$

- Decomp1 - We have $\mathtt{x}\theta = \mathtt{s}\theta$ and $L\theta = M\theta$.

  Then $(\mathtt{x} : L)\theta = (\mathtt{s} : L)\theta = (\mathtt{s} : M)\theta$ as required.

- Decomp1' - We have $\mathtt{x}\theta = \mathtt{a}\theta$ and $L\theta = M\theta$.

  Then $(\mathtt{x} : L)\theta = (\mathtt{a} : L)\theta = (\mathtt{a} : M)\theta$ as required.

- Decomp2 - We have $\mathtt{x}\theta = (\mathtt{y} : \mathtt{x}')\theta$ and $(\mathtt{x}' : L)\theta = M\theta$.

  Then $(\mathtt{x} : L)\theta = (\mathtt{y} : \mathtt{x}' : L)\theta = (\mathtt{y} : M)\theta$ as required.

- Decomp2' - We have $\mathtt{y}\theta = (\mathtt{x} : \mathtt{y}')\theta$ and $L\theta = (\mathtt{y}' : M)\theta$.

  Then $(\mathtt{y} : M)\theta = (\mathtt{x} : \mathtt{y}' : M)\theta = (\mathtt{x} : L)\theta$ as required.

- Decomp4 - We have $\mathtt{x}\theta = \mathtt{a}\theta$ and $\mathtt{y}\theta = (\mathtt{empty})\theta = \mathtt{empty}$.

  Then $(\mathtt{a} : \mathtt{y})\theta = (\mathtt{x} : \mathtt{y})\theta = (\mathtt{x} : \mathtt{empty})\theta = \mathtt{x}\theta$ as required.

- Subst1 - We have $\mathtt{x}\theta = L\theta$

  $$\begin{aligned} \theta \in \mathscr{U}(\langle \varnothing, \sigma_{S'} \rangle) &\iff \theta \in \mathscr{U}(\langle \varnothing, \sigma_S \circ (\mathtt{x} \mapsto L) \rangle) \\ &\iff \theta \in \mathscr{U}(\langle \varnothing, \sigma_S \circ (\mathtt{x} \mapsto L) \rangle) \wedge \mathtt{x}\theta = L\theta \\ &\iff \theta \in \mathscr{U}(\langle \mathtt{x} = L, \sigma_S \rangle) \end{aligned}$$

- Subst2 - We have $\mathtt{x}\theta = (\mathtt{empty})\theta$ and $L\theta = M\theta$.

  Then $(\mathtt{x} : L)\theta = (\mathtt{empty} : L)\theta = L\theta = M\theta$ as required.

- Subst3 - We have $\mathtt{x}\theta = (\mathtt{a} : \mathtt{x}')\theta$ and $(\mathtt{x}' : L)\theta = M\theta$.

  Then $(\mathtt{x} : L)\theta = (\mathtt{a} : \mathtt{x}' : L)\theta = (\mathtt{a} : M)\theta$ as required.

- Orient1-4 - Since $=_{\mathrm{AU}}$ is an equivalence relation and hence symmetric, a unifier of the conclusion is also a unifier of the premise.

□

**Theorem 2** (Soundness) *If $P \Rightarrow^{+} P'$ with $P' = \langle \varnothing, \sigma_{P'} \rangle$ in solved form, then $\sigma_{P'}$ is a unifier of $P$.*

*Proof.* We have that $\sigma_{P'}$ is a unifier of $P'$ by definition. A simple induction with Lemma 4 shows that $\sigma_{P'}$ must be a unifier of $P$. □

### 3.4 Completeness

In order to prove that our algorithm is complete, by Definition 5 we have to show that for any unifier $\delta$, there is a unifier in our solution set that is more general.

Our proof involves using a *selector* algorithm that takes a unification problem $\langle s =^? t \rangle$ together with an arbitrary unifier $\delta$, and outputs a *path* of the unification tree associated with a more general unifier than $\delta$. This is very similar to [Sie78] where completeness of a A-unification algorithm is shown via such selector.

Due to space restrictions the following lemma is given without proof. For the selector algorithm and proof (comprising of an extra 9 pages) see the long version of this paper [HP14].

**Lemma 5** (Selector Lemma) *There exists an algorithm* $\mathsf{Select}(\delta, s =^? t)$ *that takes an equation $s =^? t$ and a unifier $\delta$ as input and produces a sequence of* selections $B = (b_1, \ldots, b_k)$ *such that:*

- *$\mathsf{Unify}(s =^? t)$ has a path specified by $B$ .*
- *For all selections $b \in B$: if $\sigma$ is the substitution corresponding to $b$, then there exists an instantiation $\lambda$ such that $\sigma \circ \lambda \leq \delta$ .*

For example, consider the unification problem $\langle \mathtt{y} : 2 = \mathtt{a} : \mathtt{x} \rangle$ and $\delta = (\mathtt{x} \mapsto 1 : 2, \mathtt{y} \mapsto \mathtt{a} : 1)$ as a unifier. The unification tree was shown in Figure 8. $\mathsf{Select}(\delta, \mathtt{y} : 2 = \mathtt{a} : \mathtt{x})$ would produce selections (Subst3, Decomp2', Orient1, Subst1), which corresponds to the right-most path in the tree. The unifier at the end of this path is $\sigma = (\mathtt{x} \mapsto \mathtt{y}' : 2, \mathtt{y} \mapsto \mathtt{a} : \mathtt{y}')$ which is more general than $\delta$ by instantiation $\lambda = (\mathtt{y}' \mapsto 1)$.

Now we are able to state our completeness result, which follows directly from Lemma 5.

**Theorem 3** (Completeness) *For every unifier $\delta$ of a unification problem $\langle s =^? t \rangle$, there exist a unifier $\sigma$ generated by* $\mathsf{Unify}$ *such that $\sigma \leq \delta$ .*

## 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. Moreover, we have proved completeness in that every unifier of the input problem is an instance of some unifier in the computed set of solutions.

Future work includes establishing a Critical Pair Lemma in the sense of [Plu05]; this entails developing a notion of *independent* rule schema applications, as well as restriction and embedding theorems for derivations with rule schemata. Another topic is to consider critical pairs of conditional rule schemata (see [EGH$^+$12]).

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.

**Acknowledgements:** We thank the anonymous referees of GCM 2014 for their comments on previous versions of this paper.

# Bibliography

[BFPR15] C. Bak, G. Faulkner, D. Plump, C. Runciman. A Reference Interpreter for the Graph Programming Language GP 2. In *Proc. Graphs as Models (GaM 2015)*. Electronic Proceedings in Theoretical Computer Science 181, pp. 48–64. 2015.

[BS01] F. Baader, W. Snyder. Unification Theory. In Robinson and Voronkov (eds.), *Handbook of Automated Reasoning*. Pp. 445–532. Elsevier and MIT Press, 2001.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer-Verlag, 2006.

[EGH$^+$12] H. Ehrig, U. Golas, A. Habel, L. Lambers, F. Orejas. M-Adhesive Transformation Systems with Nested Application Conditions. Part 2: Embedding, Critical Pairs and Local Confluence. *Fundamenta Informaticae* 118(1):35–63, 2012.

[GLEO12] U. Golas, L. Lambers, H. Ehrig, F. Orejas. Attributed Graph Transformation with Inheritance: Efficient Conflict Detection and Local Confluence Analysis Using Abstract Critical Pairs. *Theoretical Computer Science* 424:46–68, 2012.

[HKT02] R. Heckel, J. M. Küster, G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *Proc. International Conference on Graph Transformation (ICGT 2002)*. Lecture Notes in Computer Science 2505, pp. 161–176. Springer-Verlag, 2002.

[HP14] I. Hristakiev, D. Plump. A Unification Algorithm for GP (Long version). http://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.Full.GCM14.pdf, 2014.

[Jaf90] J. Jaffar. Minimal and Complete Word Unification. *Journal of the ACM* 37(1):47–85, 1990.

[Pla99] W. Plandowski. Satisfiability of Word Equations with Constants is in PSPACE. In *Symposium on Foundations of Computer Science (FOCS 1999)*. Pp. 495–500. IEEE Computer Society, 1999.

[Plo72] G. Plotkin. Building-in Equational Theories. *Machine intelligence* 7(4):73–90, 1972.

[Plu05] D. Plump. Confluence of Graph Transformation Revisited. In Middeldorp et al. (eds.), *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science 3838, pp. 280–308. Springer-Verlag, 2005.

[Plu12] D. Plump. The Design of GP 2. In *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*. Electronic Proceedings in Theoretical Computer Science 82, pp. 1–16. 2012.

[Sch92] K. U. Schulz. Makanin's Algorithm for Word Equations: Two Improvements and a Generalization. In *Proc. Word Equations and Related Topics (IWWERT '90)*. Lecture Notes in Computer Science 572, pp. 85–150. Springer-Verlag, 1992.

[Sie78] J. Siekmann. *Unification and Matching Problems*. PhD thesis, University of Essex, 1978.