1

Unification in an algebra with choice and action prefix.

N.J.Drost
Programming Research Group
University of Amsterdam
The Netherlands

July 29, 1994

[1]This work was partially supported by the Foundation for Computer Science in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO).

Abstract

This paper contains a unification algorithm for a restricted process algebra with as only operators nondeterministic choice and unary action prefix-operators, and with infinite processes. Termination is ensured by using a modified version of a method for syntactic unification described in [Col84a] and [Col84b]. The algorithm contains a procedure to remove non-most general solutions.

Key words and phrases: unification, process algebra.

Contents

1 Introduction **2**

2 Algebra and theory **5**
2.1 Introduction 5
2.2 The theory 5
2.3 The model $\mathcal{P}$ 7

3 The algorithm UnPref **10**
3.1 Introduction and definitions 10
3.2 The rules 12
3.3 The algorithm UnPref 14

4 Properties of the algorithm UnPref **19**

5 A matching algorithm for substitution **27**
5.1 The algorithm MoreGeneral 27
5.2 Correctness of MoreGeneral 31

6 Complexity of UnPref and MoreGeneral **35**
6.1 Complexity of UnPref 35
6.2 Complexity of MoreGeneral 37

7 Conclusions and discussion **38**

Chapter 1

Introduction

In this paper I give an algorithm for equation solving in a small process algebra. Process algebras are algebras devised for the description, design, and verification of concurrent processes. A concurrent process is a process that consists of several parts, which may proceed independently or interact with each other. Standard examples of concurrent processes are computer networks and vending machines. A fundamental assumption in process algebras is that processes consist of atomic actions, so the objects in the domain of a process algebra are compositions of basic elements representing atomic actions. Examples of functions in process algebras are nondeterministic choice, action prefix, sequential and parallel composition, and communication. A process theory will contain in its signature a set of constant symbols for the atomic actions (and possibly some symbols for special constants), and a set of symbols for process operators.

The words "algebra" (or "model") and "theory" (or "equational specification") are used in their model theoretic sense, i.e. an algebra (or model) is a pair consisting of a domain and a set of functions on that domain, a theory (or equational specification) is a pair consisting of a signature and a set of axioms over that signature, an algebra $\mathcal{A}$ is a Σ-algebra if there is a mapping of constant and function symbols in Σ to resp. objects and functions in $\mathcal{A}$, and a Σ-algebra $\mathcal{A}$ is an algebra (a model) of a theory $(\Sigma, \mathcal{E})$ if for all axioms $s = t \in \mathcal{E} : \mathcal{A} \models s = t$. If $(\Sigma, \mathcal{E})$ is a theory, $\mathcal{A}$ is a Σ-algebra, each function in $\mathcal{A}$ corresponds to a function symbol in Σ, and $\mathcal{A} \models s = t$ iff $\mathcal{E} \vdash s = t$, then $(\Sigma, \mathcal{E})$ is called a **complete axiomatization** of $\mathcal{A}$. $\mathcal{A}$ is not necessarily isomorphic to the initial model of $(\Sigma, \mathcal{E})$ because the domain of $\mathcal{A}$ may contain objects that do not correspond to terms in the signature. In the case of process algebras these objects represent infinite processes.

Process algebras may be used as a description method in the development of parallel systems. In top-down modular development first an overall description of a system is given, and then the system is divided into a number of modules. The combination of the modules must conform to the overall description of the system. If the description of one or more modules is missing then this corresponds to an equation in a process algebra containing variables. Solving of this equation gives then all possible solutions for the unspecified modules.

Methods for solving equations in process algebras are for instance given by Parrow [Par89], and Larsen and Liu [LL90]. Both methods allow a wide range of process algebra operators, but a small subset of equations: only one variable occurrence in each equation. The algorithm of Parrow is not aimed at producing a complete description of all solutions. The method of Larsen and Liu is not easily extendable to other types of equations. So if an algorithm for solving wider classes of equations is desired, another approach is needed.

Equation solving in abstract algebras is used in many other fields of mathematics and

computer science, e.g. deduction systems, term rewriting, logic programming, and expert systems. Syntactic unification is unification in a theory without axioms, so the algebra used is the term algebra. A solution to a set of equations is a substitution that equalizes the left and right side of each equation. The first algorithm for syntactical unification is due to Robinson [Ro65], who was also the first to use the word unification. He defined a partial order on solutions of a unification problem such that one solution is smaller (more general) than a second solution if the second solution is derivable from the first. He also showed that the solution set of a syntactic unification problem can be described by giving one most general unifier. Semantic unification or E-unification is unification in a theory with a set E of axioms. A solution to a problem P is a substitution σ such that $E \vdash s\sigma = t\sigma$ for all equations $s = t$ in P. This is the same as equality in the algebra of open terms modulo E. In semantic unification the solution set (if it exists) of a problem cannot always be described with one most general unifier. The first algorithm for semantic unification was given by Plotkin [Plo72]. He defined the notion of a maximally general set of unifiers, also called a complete set of most general unifiers [JK91].

Siekmann [Sie90] gave a classification of equational theories based on the cardinality of solution sets of problems in the theory. An equational theory is **nullary** if there are equations that have at least one solution but for which there exists no complete set of most general unifiers. It is **infinitary** if there are equations for which there exists only an infinite set of most general unifiers. A **unitary** theory has for each equation that has at least one solution a complete set of most general unifiers of size 1. A **finitary** theory has for each equation that has at least one solution a complete set of most general unifiers of finite size.

Jouannaud and Kirchner [JK91] give an overview of unification algorithms described by rules. A rule transforms an equation set into another equation set with the same set of solutions. An algorithm based on rules should eventually transform the equation set into one or more equation sets in solved form.

E-unification differs from equation solving in a process algebra because the first starts with a given theory and the second with a given algebra. With a process algebra often also a theory is given. If the axioms are a complete axiomatization of the algebra and all objects in the domain of the algebra correspond to terms in the algebra, then equation solving in the process algebra and E-unification coincide. But most process algebras have objects that do not correspond to terms in the signature. These objects represent infinite processes and are an integral part of the process algebra. Often extra features are added to describe infinite processes, e.g. process constants and recursive sets of equations in CCS and ACP_τ, and μ-expressions in CSP. This has consequences for the definition of a solution of a unification problem. A solution is now a semantic substitution (a function that replaces variables in processes by process elements) that makes the interpretation of the equations true in the model. A solution is more general than another solution if there is a semantic substitution that makes the first solution equal to the second.

Here I give a unification algorithm for a restricted version of a process algebra with as operators nondeterministic choice and action prefix, and with infinite processes. Nondeterministic choice behaves as union in the algebra of sets and has been treated in [Dro92]. The action prefix operators are free unary operators that correspond to prefixing a process with an action. Process algebras with action prefix are CCS [Mil80,Mil89] and CSP [Hoa85].

There are methods for combining the unification algorithms of two algebras, producing an algorithm for the combination of the two algebras [Sch89], but these methods are not usable for algebras with infinite processes. Some unification algorithms are available for syntactic unification with infinite terms [Col84a], [MM82], [Jaf84], but no algorithms for semantic unification in an algebra with infinite objects are known. In the presence of infinite processes

special provisions are needed to ensure that the program terminates. In my algorithm I use a method based on the one described in [Col84a] and [Col84b].

The paper is organized as follows: In Chapter 2 the signature and axioms, and a model of the algebra are described. Chapter 3 gives some definitions and describes the algorithm, first as a set of rules, and also more detailed as a pseudo-program. Chapter 4 contains the correctness proof of the unification algorithm. Chapter 5 gives an algorithm for discarding solutions that are not most general, Chapter 6 contains a complexity analysis, and Chapter 7 gives conclusion and discussion.

Chapter 2

Algebra and theory

2.1 Introduction

It is assumed that processes consist of indivisible atomic actions. There is a finite set of constants, where each constant denotes a process that performs an atomic action and then terminates successfully. There is also a process δ that immediately terminates unsuccessfully. Operators on processes are nondeterministic choice and action prefix. An action prefix operator $a.$ prefixes a process with atomic action a. Action prefix operators occur in the process algebras CCS [Mil89] and CCP [Hoa85]. The algebra contains finite and infinite processes. An infinite process may have an infinite sequence of atomic actions, but not a choice between infinitely many atomic actions. An infinite process is denoted in the signature by a recursive specification.

2.2 The theory

The signature Σ of the theory contains:

1. a finite set $\mathcal{A}$ of constant symbols, plus a special constant δ. Notation: $a, b, c, \ldots \in \mathcal{A}$; $A, B, C, \ldots \subseteq \mathcal{A}$,
2. unary operators $a\cdot$ for each $a \in \mathcal{A}$, and binary operator $+$,
3. a countably infinite set $\mathcal{V}$ of variables. Notation: $x, y, z, u, v \in \mathcal{V}$, $X, Y, Z, U, V \subseteq \mathcal{V}$.

The set of terms over this signature is denoted by $\mathcal{T}$. Notation: $s, t, \ldots \in \mathcal{T}$; $S, T, \ldots \subseteq \mathcal{T}$. A term $t \in \mathcal{T}$ is a **constant** a, a **variable** x, a **prefix term** at', a **sum** $t_1 + t_2$, or δ. The notation δ for a process that terminates immediately is taken from [BK84]. The corresponding notation in CCS is 0 and in CSP is NIL. Alternative notations for a finite sum are ΣT for finite sets $T \subseteq \mathcal{T}$ and $\sum_{i \in I} t_i$ for finite index sets I and $t_i \in T$. $\Sigma\emptyset$ is defined as δ.

The functions *Var* and *Const* give the variables, resp. the constants of a term. The function *Smd* gives the set of summands of a sum:

Definition 2.1 *Function* $Smd : \mathcal{T} \to \mathcal{P}(\mathcal{T})$ *is defined as:*

$Smd(\delta) = \emptyset$, $Smd(a) = \{a\}$, $Smd(x) = \{x\}$, $Smd(at) = \{at\}$,
$Smd(t_1 + t_2) = Smd(t_1) \cup Smd(t_2)$.

The set of subterms of a term, including the term itself, is given by the function Sub:

Definition 2.2 *$Sub : \mathcal{T} \longrightarrow \mathcal{P}(\mathcal{T})$ is defined as:*

$Sub(\delta) = \{\delta\}$, $Sub(a) = \{a\}$, $Sub(x) = \{x\}$,
$Sub(a \cdot t) = \{a \cdot t\} \cup \{a\} \cup Sub(t)$
$Sub(t_1 + t_2) = \{t_1 + t_2\} \cup Sub(t_1) \cup Sub(t_2)$

The depth of a term will be used in induction proofs:

Definition 2.3 *The* **depth** *$d(t)$ of a term t is defined as:*

$d(\delta) = d(a) = d(x) = 1$,
$d(at) = 1 + d(t)$,
$d(t_1 + t_2) = max(d(t_1), d(t_2))$.

Variables in a term may be replaced by arbitrary terms using (syntactic) substitution.

Definition 2.4 *A* **(syntactic) substitution** *is a function from $\mathcal{V}$ to $\mathcal{T}$. Notation: $\sigma = \{x := t_x \mid x \in X\}$ with $x \not\equiv t_x$ for all $x \in X$.*
$Dom(\sigma) = X$ and $Ran(\sigma) = \bigcup\{Var(t_x) \mid x \in X\}$.

Substitutions are extended in the obvious way to functions from $\mathcal{T}$ to $\mathcal{T}$. Application of a (syntactic) substitution σ to a term t is denoted as $t\sigma$.

Terms from $\mathcal{T}$ correspond to finite processes. Infinite processes are described using recursive specifications:

Definition 2.5 *A* **recursive specification** *is a set of equations of the form $\{x = t_x \mid x \in X\}$ with one equation for each $x \in X$.*

A variable $x \in X$ from a recursive specification $E \equiv \{x = t_x \mid x \in X\}$ is denoted as $< x \mid E >$. The variables in X are called **recursion variables** of E and the variables in $Var(E) - X$ the **free variables** of E. Function *RecVar* gives the recursion variables of a recursive specification, and function *FreeVar* the free variables.

An arbitrary recursive specification may have zero, one, or more solutions in a given process model. In most models, including the model used in this paper, all recursive specifications that are guarded have exactly one solution. Therefore only guarded recursive specifications are used.

Definition 2.6 *A recursive specification E is* **guarded** *if there is no subset $\{x_0 = t_0, \ldots, x_n = t_n\}$ in E with $x_i \in Smd(t_{i+1})$ for $1 \leq i < n$ and $x_n \in Smd(t_1)$ for some $n \geq 0$.*

The axiom set $\mathcal{E}$ is the same as the axiom set of the theory of sets with union and empty set:

$(x + y) + z = x + (y + z)$	(associativity)
$x + y = y + x$	(commutativity)
$x + x = x$	(idempotence)
$x + \delta = x$	(unit element)

Notation: $\equiv$ will be used for syntactic equality between terms and $=_{\mathcal{E}}$ for equality modulo the axioms.

The algebra is in fact a combination of the algebra of sets with union and empty set and an algebra with a finite number of free unary functions, i.e. functions that do not occur in axioms.

The unification algorithm will work on terms in normal form. A normalizing function on terms is defined. First a sequence of subsets of $\mathcal{T}$ is defined based on the number of "+" symbols in a term, such that $\mathcal{T}_n$ denotes all terms with at most n "+" symbols. The function N_+ gives the number of occurrences of the operator "+" in a term:

Definition 2.7

$N_+(\delta) = N_+(a) = N_+(x) = 0$,
$N_+(a{\cdot}t) = N_+(t)$,
$N_+(t_1 + t_2) = N_+(t_1) + N_+(t_2) + 1$.

For each $\mathcal{T}_n$ a normalizing function NF_n and total order $\leq_n$ are defined. Total orders on $\mathcal{A}$ and $\mathcal{V}$ are presupposed.

Definition 2.8 *For each $n \geq 0$ a function $\mathrm{NF}_n : \mathcal{T}_n \to \mathcal{T}_n$ and a total order $\leq_n$ on $\mathcal{T}_n$ are inductively defined as follows:*

For all $n \geq 0$:
$NF_n(\delta) = \delta$, $NF_n(a) = a$, $NF_n(x) = x$, $NF_n(at) = a{\cdot}NF_n(t)$,
$\delta \leq_n s$, $s \leq_n s$, $a \leq_n x$ for all $a \in \mathcal{A}$, $x \in \mathcal{V}$, and $s \in \mathcal{T}_n$,
if $s \in \mathcal{A} \cup \mathcal{V}$, $t \equiv at'$, and $t \in \mathcal{T}_n$, then $s \leq_n t$ iff $s \leq_n a$, and $t \leq_n s$ iff $a \leq_n s$ and $a \not\equiv s$,
if $s \equiv as'$, $t \equiv bt'$, and $s, t \in \mathcal{T}_n$, then $s \leq_n t$ iff $a <_n b$, or $a = b$ and $s' \leq_n t'$.

For $n > 0$:
Suppose t is a sum and $N_+(t) \leq n$.
Define $T \stackrel{\text{def}}{=} \{NF_{n-1}(t') \mid t' \in Smd(t)\}$.
Suppose $T = \{t_1, \ldots, t_m\}$, and $t_i <_{n-1} t_{i+1}$ for all $1 \leq i < m$. Then:
$NF_n(t) = t_1 + \ldots + t_m$.
If s or t is a sum:
Define $T_s \stackrel{\text{def}}{=} \{NF_{n-1}(s') \mid s' \in Smd(s)\}$, $T_t \stackrel{\text{def}}{=} \{NF_{n-1}(t') \mid t' \in Smd(t)\}$,
$T_L = T_s - T_t$, $T_R = T_t - T_s$. Then:
$s \leq_n t$ iff $T_L = \emptyset$, or there is a $t_j \in T_R$ such that for all $s_i \in T_L : s_i \leq_{n-1} t_j$.

Proposition 2.1
i) $\leq_n$ *is a total order on $\mathcal{T}_n$ for each $n \geq 0$,*
ii) $N_+(t) \geq n \Longrightarrow \mathrm{NF}_n(t) = \mathrm{NF}_{n-1}(t)$,
iii) $s <_n t \Longrightarrow s <_{n-1} t$ $(n > 0)$.

Proof: easy.

Now the normalizing function $NF : \mathcal{T} \to \mathcal{T}$ is defined as:

Definition 2.9 $\mathrm{NF}(t) = \mathrm{NF}_n(t)$ *with* $n = N_+(t)$.

Proposition 2.2 $\mathrm{NF}(s) = \mathrm{NF}(t)$ *iff* $s =_{\mathcal{E}} t$.

Proof: easy.

2.3 The model $\mathcal{P}$

Because of the set-like nature of the choice operator, processes are modeled as finite and possibly non-well founded sets. Such sets are called Psets and contain pairs where the first component is an atomic action or a variable, and the second component is a Pset or a set containing a special symbol $\surd$.

Definition 2.10 *A Pset is a finite set of pairs (d, q) with $d \in \mathcal{A} \cup \mathcal{V}$ and q a Pset or $q = \{\surd\}$.*

The idea of modeling processes as sets of pairs comes from [BMOZ88].
A Pset q is a **state** of a Pset p if $p = q$, or $(a, p') \in p$ for some $a \in \mathcal{A}$ and q is a state of p'. A Pset with a finite number of states is called **regular**. The domain $\mathcal{D}$ of $\mathcal{P}$ consists of all regular Psets in which variables $x \in \mathcal{V}$ only occur in pairs $(x, \{\surd\})$. The operators of $\mathcal{P}$ are $\oplus$ (nondeterministic choice) and action prefix $\underline{a\cdot}$ for each $a \in \mathcal{A}$.

The model $\mathcal{P}$:

Domain $\mathcal{D}$: regular Psets in which variables $x \in \mathcal{V}$ only occur in pairs $(x, \{\surd\})$.
Operators: $\oplus$ and $\underline{a\cdot}$ for each $a \in \mathcal{A}$ with:

$p_1 \oplus p_2 = p_1 \cup p_2$,
$\underline{a\cdot}p = \{(a, p)\}$.

Terms from $\mathcal{T}$ are mapped to regular Psets as follows:

$$[\![\delta]\!] = \emptyset,\ \ [\![a]\!] = \{\,(a, \{\surd\})\,\},\ \ [\![x]\!] = \{\,(x, \{\surd\})\,\},\ \ [\![a{\cdot}t]\!] = \underline{a\cdot}[\![t]\!],\ \ [\![t_1 + t_2]\!] = [\![t_1]\!] \oplus [\![t_2]\!].$$

Sets of terms from $\mathcal{T}$ are mapped to sets of Psets from $\mathcal{D}$:

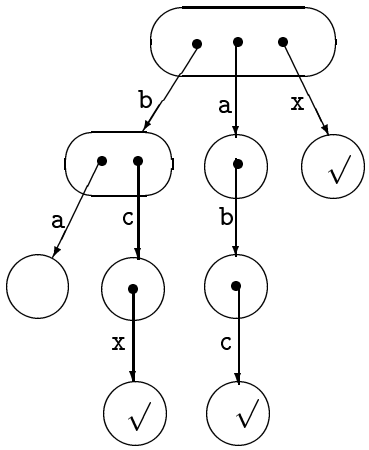
$$[\![T]\!] = \{\,[\![t]\!] \mid t \in T\,\}.$$

Recursion variables from finite guarded recursive specifications are also mapped to Psets from $\mathcal{D}$. Suppose $E \equiv \{x = t_x \mid x \in X\}$ is a finite guarded recursive specification. Then:

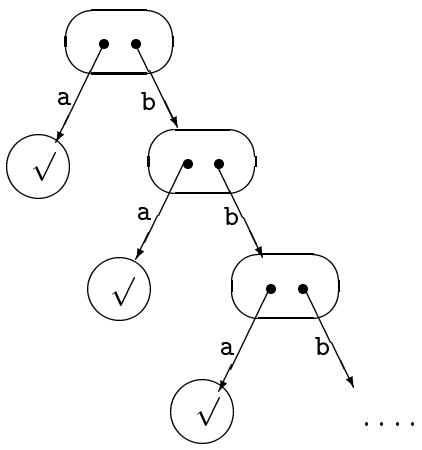
$$[\![\langle x|E\rangle]\!] = [\![\, t_x\{x := \langle x|E\rangle \mid x \in X\}\,]\!].$$

A Pset may be depicted as a kind of tree, where each set is represented as an oval and each pair as a labeled arrow to an oval:

$[\![\, b(a\delta + cx) + abc + x\,]\!]$:



$[\![\langle x|E\rangle]\!]$ with $E \equiv \{x = a + bx\}$:



Definition 2.11 *A* **path** *in a Pset is a sequence $(a_0, \ldots, a_n)$ or $(a_0, \ldots, a_i, \ldots$ with p_0 a state of p and there are p_i such that $(a_{i+1}, p_{i+1}) \in p_i$ for all $i > 1$ (and $i < n$).*
A **cycle** *is a path $(a_0, \ldots, a_n)$ from some p_0 to p_n with $n > 0$ and $p_0 = p_n$.*

Definition 2.12 *The* **projection function** *π_n on processes is defined for all $n \geq 1$ as:*

$$\begin{array}{lll}
\pi_n(\emptyset) & = \emptyset & \text{for all } n \geq 1, \\
\pi_n(\{\,(a, \{\surd\})\,\}) & = \{\,(a, \{\surd\})\,\} & \text{for all } n \geq 1, \\
\pi_1(\underline{a\cdot}p) & = \{\,(a, \{\surd\})\,\}, & \\
\pi_n(\underline{a\cdot}p) & = \{\,(a, \pi_{n-1}(p))\,\} & \text{for all } n > 1, \\
\pi_n(p \oplus p') & = \pi_n(p) \oplus \pi_n(p') & \text{for all } n \geq 1.
\end{array}$$

An **assignment** is a function from $\mathcal{V}$ to $\mathcal{D}$. Notation: $\psi = \{ x := q_x \mid x \in X \}$, with $\psi(x) \neq x$ iff $x \in X$. Assignments are used to define semantic substitutions. A semantic substitution is a function from $\mathcal{D}$ to $\mathcal{D}$, that replaces each occurrence of a variable of a given set of variables by a number of Pset-elements. Application of a semantic substitution α to a process p is denoted as $p\alpha$. The notation for a semantic substitution is equal to the notation of the assignment that is used as a basis: $\alpha = \{ x := q_x \mid x \in X \}$, with $[\![x]\!]\alpha \neq [\![x]\!]$ iff $x \in X$.

Definition 2.13 *Suppose p is a Pset and $\alpha = \{ x := q_x \mid x \in X \}$. Application of the semantic substitution α to process p is defined as:*

$$\begin{aligned} p\alpha \;\; = \;\; & \{ (d, p'\alpha) \mid (d, p') \in p \text{ and } d \in \mathcal{A} \cup \mathcal{V} - X \} \\ & \cup \bigcup \{ q_x \mid (x, \{\surd\}) \in p \text{ and } x := q_x \in \alpha \}, \end{aligned}$$

where $\{\surd\}\alpha = \{\surd\}$.

Example:
Suppose $p = \{ (a, \emptyset), (x, \{\surd\}), (y, \{\surd\}) \}$, and $\alpha = \{ x := \{ (b, \{\surd\}) \} \}$.
Then $p\alpha = \{ (a, \emptyset), (b, \{\surd\}), (y, \{\surd\}) \}$.
The set X of a semantic substitution $\alpha = \{ x := q_x \mid x \in X \}$ is called the domain of α. The function Dom gives the domain of a semantic substitution. The set $\bigcup \{ y \mid y \text{ occurs in } q_x \text{ for some } x \in X \}$ is called the range of α and is produced by the function Ran. A semantic substitution will often be denoted as $\alpha = \{ x := [\![t_x]\!] \mid x \in X \}$ with the t_x terms from $\mathcal{T}$. Suppose $\sigma = \{ x := t_x \mid x \in X \}$ is a (syntactic) substitution. Then $[\![t\sigma]\!] = [\![t]\!]\alpha$.

A semantic substitution may be restricted to a subset X of $\mathcal{V}$:

Definition 2.14 *The* **restriction to** *$X \subseteq \mathcal{V}$ of a semantic substitution α is defined as:*

$$\alpha_{|X} \stackrel{\text{def}}{=} \{ x := q_x \mid x \in X \text{ and } x := q_x \in \alpha \}.$$

Chapter 3

The algorithm UnPref

3.1 Introduction and definitions

The unification algorithm presented in this paper is based on rules. A rule transforms a unification problem consisting of a set of equations (possibly existentially quantified) into a number of alternative unification problems. The algorithm has as input a set of equations over $\mathcal{T}$(not quantified). Special provisions are needed to ensure that the algorithm terminates in the presence of infinite processes. The method used here is based on a method described by Colmerauer [Col84a] for unification of syntactic trees that may be infinite. First some general unification definitions are given. These are based on [JK91], except the definition of a solution as a semantic substitution, and a solved form as a set of guarded equations. The notation follows the proposal in [DJ91].

Definition 3.1 *A* **(unification) problem (equation solving problem)** *is a possibly existentially quantified set of equations*

$$\exists\, \vec{u}\{s_i =^?_{\mathcal{E}} t_i \mid i \in I\}$$

with $\vec{u}$ *a sequence of variables from* $\mathcal{V}$ *and all* s_i *and* t_i *from* $\mathcal{T}$.

A solution of a unification problem is a semantic substitution:

Definition 3.2 *Suppose* $\exists\, \vec{u}P$ *is a problem with* $P = \{s_i =^?_{\mathcal{E}} t_i \mid i \in I\}$ *and* $V = Var(P) - \vec{u}$. *A* **solution (unifier)** *of* $\exists\, \vec{u}P$ *is a semantic substitution* α *such that there is a semantic substitution* β *with* $Dom(\beta) = \vec{u}$, $Ran(\beta) \cap \vec{u} = \emptyset$, *and*

$$\mathcal{P} \models [\![s_i]\!]\, \alpha_{|V}\, \beta = [\![t_i]\!]\, \alpha_{|V}\, \beta \text{ for all } i \in I.$$

The set of all solutions of a problem $\exists\, \vec{u}P$ *is denoted as* $Sol(\exists\, \vec{u}P)$.

Definition 3.3 *A problem* $\exists \vec{u}_1 P_1$ *is* **equivalent** *with a problem* $\exists \vec{u}_2 P_2$ *iff* $Sol(\exists \vec{u}_1 P_1) = Sol(\exists \vec{u}_2 P_2)$.

Definition 3.4 *A* **rule** *is a transition*

$$\exists\, \vec{u}P \Longrightarrow \{\exists \vec{u}_1 P_1, \ldots, \exists\, \vec{u}_n P_n\}$$

with $\exists\, \vec{u}P$, $\exists \vec{u}_1 P_1, \ldots,$ $\exists\, \vec{u}_n P_n$ *unification problems.*

The algorithm should eventually produce unification problems that are easily transformed to solutions. Such unification problems are called solved forms.

Definition 3.5 *A* **solved form** *is a unification problem* $\exists \vec{u} P$ *with* $P = \{x_1 =^?_{\mathcal{E}} t_1, \ldots, x_n =^?_{\mathcal{E}} t_n\}$, *where*

1. *all* x_i *are different,*
2. *the set* $\{x_1 = t_1, \ldots, x_n = t_n\}$ *is guarded.*

For unification in algebras without infinite objects there is an extra condition, the occur check, in the definition of a solved form (see for instance [JK91]): occurrences of the variables x_1 to x_n are not allowed in the terms t_1 to t_n. This condition is omitted here because these equations correspond to infinite processes via a guarded recursive specification.

Solutions to a unification problem can be partially ordered from more general to more specific. More specific solutions can be derived from a more general one by combining the general solution with a substitution.

Definition 3.6 *A semantic substitution* α *is* **more general** *with respect to a set* $X \subseteq \mathcal{V}$ *than a semantic substitution* β, *notation* $\alpha \leq_X \beta$, *if there is a semantic substitution* γ *such that for all* $x \in X : x\alpha\gamma = x\beta$. *This order is called the* **subsumption order**.

Definition 3.7 *A semantic substitution* α *is* **most general** *with respect to a set* $X \subseteq \mathcal{V}$ *in a set of semantic substitutions if it is minimal in this set with respect to the subsumption order* $\leq_X$.

A unification algorithm should produce a set of solutions that is as small as possible and from which all solutions are derivable.
Suppose $\exists \vec{u} P$ is a unification problem and $V = Var(P) - \vec{u}$.

Definition 3.8 *A set SU of unifiers of* $\exists \vec{u} P$ *is* **complete** *if for all* $\alpha \in Sol(\exists \vec{u} P)$ *there is a* $\beta \in SU$ *such that* $\beta \leq_V \alpha$.

Definition 3.9 *A complete set SU of unifiers of* $\exists \vec{u} P$ *is* **minimal** *if for all* $\alpha, \beta \in SU$: *if* $\alpha \leq_V \beta$, *then* $\alpha \equiv \beta$.

The rules that are used in the unification algorithm described in this paper maintain two sets of equations (possibly quantified). Notation: $\exists \vec{u} \langle S, P \rangle$. At the start of the algorithm the first set S is empty, and the second set P consists of the problem to be solved. Rules may put equations of the form $x =^?_{\mathcal{E}} t_x$ and $as =^?_{\mathcal{E}} at$ into S. The set S must have some special properties to ensure that its equations may be used as rewrite rules.

Definition 3.10 *A set of equations is* **circular** *if each right hand side also occurs as a left hand side.*

Definition 3.11 *A set S of equations is* **reduced** *if*

1. *S contains only equations of the form* $x =^?_{\mathcal{E}} t_x$ *and* $as =^?_{\mathcal{E}} at$,
2. *all left hand sides of equations in S are different,*
3. *S has no circular subset.*

The set S is used to compute **representatives** of terms of equations in P. The equations $s =^?_{\mathcal{E}} t$ in S are used as rewrite rules $s \longrightarrow t$. The rewrite rules are only used to rewrite whole summands of a term, not smaller subterms of summands. This rewriting system is confluent and terminating because S is reduced. So S defines a function on terms called *rep*. Before a rule is applied to an equation of P the equation is first transformed to an equation of representatives. If this new equation is of the form $x =^?_{\mathcal{E}} t_x$ or $at_1 =^?_{\mathcal{E}} at_2$ it is placed into S. This method ensures termination of the algorithm. The function *rep* is first described in [Hue76]. Colmerauer [Col84a] uses a simpler version of this function in an algorithm for the unification of syntactic trees that may be infinite.

Definition 3.12 *The* **representative** $rep(t, S)$ *of a term t in normal form with respect to a reduced set S of equations is defined as:*

$$rep(t,S) = Nf(rep(t_1,S) + \ldots + rep(t_n,S)) \quad \text{if } Smd(t) = \{t_1 \ldots, t_n\} \text{ for some } n > 1 \text{ and there is an equation } t_i =^?_{\mathcal{E}} t_i' \text{ in } S,$$
$$rep(t,S) = rep(t',S) \quad \text{if } t =^?_{\mathcal{E}} t' \in S,$$
$$rep(t,S) = t \quad \text{if for no summand } t' \text{ of } t \text{ there is an equation } t' =^?_{\mathcal{E}} t'' \text{ in } S.$$

The algorithm produces at termination a problem $\exists\, \vec{u}\, \langle\, S, P\, \rangle$ with $P = \emptyset$ and S a reduced set. A solved form S' is extracted from S by leaving out all equations of the form $as =^?_{\mathcal{E}} at$. It will be proved that S' has the same solutions as S by proving that S is decomposed, and that a decomposed set of equations has the same set of solutions as the maximal set in solved form contained in S. The proof uses the fact that the set of variables occurring as a left hand side in S is an eliminable set of S.

Definition 3.13 *A set S of equations is* **decomposed** *if S is reduced and for each $as =^?_{\mathcal{E}} at \in S$: $Smd(rep(s,S)) = Smd(rep(t,S))$.*

A set W of variables of a problem $\exists\, \vec{u} P$ is eliminable if there is an equivalent problem of the form $\exists\, \vec{u}\{\, w =^?_{\mathcal{E}} t_w \mid w \in W\,\}$ that has only variables from $\vec{u}$ and V in the terms t_w, and is in solved form. Then the value of each $w \in W$ in a solution of $\exists\, \vec{u}\, P$ is a function of the values of the variables in $V - W$.

Definition 3.14 *Suppose $\exists\, \vec{u} P$ is a problem and $Var(P) - \vec{u} = V$.*
A set $W \subseteq V$ is an **eliminable set** *of $\exists\, \vec{u} P$ if for each substitution $\alpha = \{x := p_x \mid x \in V - W\}$ there is exactly one substitution $\beta = \{y := p_y \mid y \in W\}$ such that $\alpha\beta$ is a solution of $\exists\, \vec{u} P$.*

3.2 The rules

Suppose s and t are terms from $\mathcal{T}$ with $\mathrm{Nf}(s) \equiv s$ and $\mathrm{Nf}(t) \equiv t$, S is a completed set of equations, and $rep(s,S) = s'$, $rep(t,S) = t'$.
Rule 1 (Eq) removes equations where the representatives of the left and right hand side are equal.

1 (Eq)	Suppose $s' \equiv t'$.
	$\exists\, \vec{u}\, \langle\, S, P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle \Longrightarrow \{\ \exists\, \vec{u}\, \langle\, S, P\, \rangle\ \}.$

Suppose in the following that $s' \not\equiv t'$.
Rule 2 (Var) is applied if s' or t' is a variable that is not a summand of the other term. It

removes the equation $s =^?_{\mathcal{E}} t$ from P and places $s' =^?_{\mathcal{E}} t'$ or $t' =^?_{\mathcal{E}} s'$ into S.

2 (Var) Suppose $(s' \equiv x,\ t' \equiv t_x,\ x \notin Smd(t'))$, or $(t' \equiv x,\ s' \equiv t_x,\ x \notin Smd(s'))$.

$$\exists \vec{u}\, \langle\, S,\, P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle \Longrightarrow \{\ \exists \vec{u}\, \langle\, S \cup \{x =^?_{\mathcal{E}} t_x\},\ P\, \rangle\ \}.$$

Rule 3 detects cases that have no solution:

3 (NoSol) Define $T_{s'} = Smd(s')$ en $T_{t'} = Smd(t')$. Suppose
$[T_{t'} \cap \mathcal{V} = \emptyset$ and $\{\mathcal{A} \cap (T_{s'} - T_{t'}) \neq \emptyset$ or $\exists as'' \in T_{s'}$ and no $at'' \in T_{t'}\}]$ or
$[T_{s'} \cap \mathcal{V} = \emptyset$ and $\{\mathcal{A} \cap (T_{t'} - T_{s'}) \neq \emptyset$ or $\exists at'' \in T_{s'}$ and no $as'' \in T_{t'}\}]$. Then:

$$\exists \vec{u}\, \langle\, S,\, P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle \Longrightarrow \{F\}.$$

Rule 4 (Prefix) is applied if s' and t' are prefix terms starting with the same action prefix. The equation $s' =^?_{\mathcal{E}} t'$ is moved from P to S, and a new equation is placed into P.

4 (Prefix) Suppose $s' \equiv as_1,\ t' \equiv at_1$. Then:

$$\exists \vec{u}\, \langle\, S,\, P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle \Longrightarrow \{\ \exists \vec{u}\, \langle\, S \cup \{as_1 =^?_{\mathcal{E}} at_1\},\, P \cup \{s_1 =^?_{\mathcal{E}} t_1\}\, \rangle\ \}.$$

Rule 5 (Sum) is used if rule NoSol is not applicable and at least one side of the equation $s' = t'$ is a sum. It is a complicated rule that incorporates an adapted version of a unification algorithm for sets, based on [BB86] and [Dro92]. Here only a global description is given. The full description is presented in the next section. The unification algorithm makes equations $as_i =^?_{\mathcal{E}} at_j$ with $as_i \in Smd(s')$ and $at_j \in Smd(t')$, and makes alternatives with different subsets of these equations. It also chooses fresh variables for the variables x in $Smd(s')$ and $Smd(t')$, and distributes constants and prefix terms from $Smd(t')$ in different ways to variables in $Smd(s')$, and from $Smd(s')$ to variables in $Smd(t')$. This results in equations $x =^?_{\mathcal{E}} t_x$ for all $x \in Smd(s') \cup Smd(t')$ with t_x a sum of fresh variables and subsets of constants and prefix terms from $Smd(s')$ or $Smd(t')$. Different alternatives are created for different subsets.

5 (Sum) Suppose s' or t' is a sum, no other rule is applicable, and $\{s' =^?_{\mathcal{E}} t'\}$ is not a guarded specification. Then:

$$\exists \vec{u}\, \langle\, S,\ P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle \Longrightarrow$$
$$\{\quad \exists \vec{u}, \vec{v}\, \langle\, S \cup \{\, x =^?_{\mathcal{E}} t_x \mid x \in Smd(s') \cup Smd(t')\,\},\ P \cup W \rangle \mid\ t_x \text{ is a sum of fresh variables plus constants and prefix terms of } Smd(s') \text{ or } Smd(t'),\ W \subseteq \{\, as_i =^?_{\mathcal{E}} at_j \mid a \in \mathcal{A},\ as_i \in Smd(s'),\ at_j \in Smd(t')\,\}\quad \}.$$

Some examples of applications of rule 5 (Sum):

$$\langle\, \emptyset,\ \{as =^?_{\mathcal{E}} at_1 + at_2\}\, \rangle \overset{Sum}{\Longrightarrow} \{\, \langle\, \emptyset,\ \{as =^?_{\mathcal{E}} at_1,\ as =^?_{\mathcal{E}} at_2\}\, \rangle\, \},$$

$$\langle\, \emptyset,\ \{x + y =^?_{\mathcal{E}} a\}\, \rangle \overset{Sum}{\Longrightarrow} \{\, \langle\, \emptyset,\ \{x =^?_{\mathcal{E}} a,\ y =^?_{\mathcal{E}} a\}\, \rangle,$$

$$\langle\ \emptyset,\ \{x =^?_{\mathcal{E}} a,\ y =^?_{\mathcal{E}} \delta\}\ \rangle,$$
$$\langle\ \emptyset,\ \{x =^?_{\mathcal{E}} \delta,\ y =^?_{\mathcal{E}} a\}\ \rangle\ \},$$

$$\langle\ \emptyset,\ \{x_1 + x_2 =^?_{\mathcal{E}} y_1 + y_2\}\ \rangle \stackrel{Sum}{\Longrightarrow} \{\ \exists u_1, u_2, u_3, u_4\ \langle\ \{x_1 =^?_{\mathcal{E}} u_1 + u_2,\ x_2 =^?_{\mathcal{E}} u_3 + u_4,\ y_1 =^?_{\mathcal{E}} u_1 + u_3,\ y_2 =^?_{\mathcal{E}} u_2 + u_4\},\ \emptyset\ \rangle\ \}.$$

3.3 The algorithm UnPref

The algorithm reads a set P of equations, puts them into normal form, calls recursively procedures for applications of rules, and produces a set of pre-solutions, which may be empty. A **pre-solution** is a set of equations that are all of the form $x =^?_{\mathcal{E}} t_x$ and $as =^?_{\mathcal{E}} at$. A solution is made from a pre-solution by leaving out all equations $as =^?_{\mathcal{E}} at$ and replacing each $x =^?_{\mathcal{E}} t_x$ by $x := t_x$.

```
prog UnPref;
var ∃u⃗ < S, P > :  Problem;
  Sol :  SetOfSolutions;
begin " read equations and put into normal form ";
  " put normalized equations into P ";
  u⃗ := ∅;
  S := ∅;
  Sol := ∅;
  CallRules(∃u⃗ < S, P >,Sol)
end.
```

Procedure CallRules chooses an equation from P and calls the procedure corresponding to the applicable rule. The **case** statement chooses the first clause with a condition that is true. If P is empty then the new solution present in S is placed into Sol.

```
proc CallRules(∃u⃗ < S, P >,Sol);
  if P = ∅ then " put S into Sol "
  else
    case
      " rule 3 (NoSol) is applicable ":  skip;
      " rule 1 (Eq) is applicable to some s =?_E t in P ":
        EqRule(∃u⃗ < S, P >, s =?_E t, Sol);
      " rule 2 (Var) is applicable to some s =?_E t in P ":
        VarRule(∃u⃗ < S, P >, s =?_E t, Sol);
      " rule 4 (Prefix) is applicable to some s =?_E t in P ":
        PrefixRule(∃u⃗ < S, P >, s =?_E t, Sol);
      " rule 5 (Sum) is applicable to some s =?_E t in P ":
        SumRule(∃u⃗ < S, P >, s =?_E t, Sol);
    esac
  fi
end.
```

Procedure EqRule removes the equation under consideration and continues the recursion.

```
proc EqRule(∃u⃗ < S, P >, s =?_E t, Sol);
```

```
  P := P − {s =?_E t};
  CallRules(∃u⃗ < S, P >, Sol)
end.
```

Procedure VarRule places an equation $x =^?_{\mathcal{E}} t_x$ into S and continues the recursion.

```
proc VarRule(∃u⃗ < S, P >, s =?_E t, Sol);
  P := P − {s =?_E t};
  if " s is a variable x " then t_x:=t else t_x:=s fi;
  " put x =?_E t_x into S ";
  CallRules(∃u⃗ < S, P >, Sol)
end.
```

Procedure PrefixRule computes the representatives of the input equation, and makes a new equation in such a way that the right hand side has not a greater depth than the left hand side. This equation is placed into S. The input equation is removed from P and replaced by the new equation where the first action prefix is stripped from both terms.

```
proc PrefixRule(∃u⃗ < S, P >, s =?_E t, Sol);
  P := P - {s =?_E t};
  " determine as_1 = rep(s, S) and at_1 = rep(t, S) ";
  if d(as_1) < d(at_1) then " add at_1 =?_E as_1 to S " else " add as_1 =?_E at_1 to S " fi
  " add s_1 =?_E t_1 to P ";
  CallRules(∃u<S,P>, Sol)
end.
```

Procedure SumRule makes a number of alternatives. In each alternative a different set P′ of equations $s_i =^?_{\mathcal{E}} t_j$ is added to P, with each (s_i,t_j) a pair of matching prefix terms. Each s_i is a summand of the representative of the left hand side of the input equation, and each t_j is a summand of the right hand side. Procedure MakeMatchSet determines all possible sets W of pairs of matching terms from the input equation. For each set W an alternative is created in which procedure SetsAlg is called.

```
proc SumRule(∃u⃗ < S, P >, s =?_E t, Sol);
  P := P - {s =?_E t};
  s′ := rep(s, S);
  t′ := rep(t, S);
  MatchSet := MakeMatchSet(s′, t′);
  foreach W ∈ MatchSet do
    " make copy S_W of S ";
    " make copy P_W of P ";
    foreach (s_i, t_j) ∈ W do
      " put s_i =?_E t_j into P_W"
    od;
    SetsAlg(∃u⃗ < S_W, P_W >, W, s′ =?_E t′);
  od
end.
```

Procedure SetsAlg is an adaptation of a unification algorithm for sets with union and empty set [BB86], [Dro92]. Procedure MakeNewVars selects new variables and attributes these to the variables in $Smd(s)$ and $Smd(t)$. Procedure DistrConstants makes a number of alternatives by distributing constants and prefix terms in $Smd(s =^?_{\mathcal{E}} t)$ to variables in $Smd(s =^?_{\mathcal{E}} t)$.

The resulting equations $\{x =^?_{\mathcal{E}} t_x \mid x \in Smd(s =^?_{\mathcal{E}} t) \cap \mathcal{V}\}$ are added to S, and the recursion is continued by calling procedure CallRules.

proc SetsAlg($\exists\, \vec{u} < S, P >$, W, $s =^?_{\mathcal{E}} t$, Sol);
  NewEqs := $\emptyset$;
  MakeNewVars($\vec{u}$, $s =^?_{\mathcal{E}} t$, NewEqs);
  DistrConstants($\exists\, \vec{u} < S, P >$, $s =^?_{\mathcal{E}} t$, NewEqs, Sol)
end.

proc MakeNewVars($\vec{u}$, $s =^?_{\mathcal{E}} t$, NewEqs);
  X := " the set of single left variables of $s =^?_{\mathcal{E}} t$ ";
  Y := " the set of single right variables of $s =^?_{\mathcal{E}} t$ ";
  Z := " the set of double variables of $s =^?_{\mathcal{E}} t$ ";

  foreach $(x, y) \in (X \times Y) \cup (X \times Z) \cup (Y \times Z)$ **do**
    " choose fresh variables $v_{\{x,y\}}$ and add them to $\vec{u}$";
  od;
  foreach $z \in Z$ **do**
    " choose fresh variables $v_{\{z\}}$ and add them to $\vec{u}$";
  od;

  foreach $x \in X$ **do**
    " put $x =^?_{\mathcal{E}}$ Nf$(\sum_{y \in Y} v_{\{x,y\}} + \sum_{z \in Z} v_{\{x,z\}})$ into NewEqs "
  od;
  foreach $y \in Y$ **do**
    " put $y =^?_{\mathcal{E}}$ Nf$(\sum_{x \in X} v_{\{x,y\}} + \sum_{z \in Z} v_{\{y,z\}})$ into NewEqs "
  od;
  foreach $z \in Z$ **do**
    " put $z =^?_{\mathcal{E}}$ Nf$(v_{\{z\}} + \sum_{x \in X} v_{\{x,z\}} + \sum_{y \in Y} v_{\{y,z\}})$ into NewEqs "
  od
end.

proc DistrConstants($\exists\, \vec{u} < S, P >$, W, $s =^?_{\mathcal{E}} t$, NewEqs, Sol);
  X := " the set of single left variables of $s =^?_{\mathcal{E}} t$ ";
  Y := " the set of single right variables of $s =^?_{\mathcal{E}} t$ ";
  Z := " the set of double variables of $s =^?_{\mathcal{E}} t$ ";
  C_D := $Smd(s) \cap Smd(t) \cap \mathcal{A}$;
  T_{D_L} := $first(W) \cup C_D$;
  T_{D_R} := $second(W) \cup C_D$;
  T_L := $Smd(s) - (\mathcal{V} \cup T_{D_L})$;
  T_R := $Smd(t) - (\mathcal{V} \cup T_{D_R})$;
  Sets$_L$:= $(\mathcal{P}(Y) - \emptyset) \cup \{\ \{z\} \mid z \in Z\}$;
  Sets$_R$:= $(\mathcal{P}(X) - \emptyset) \cup \{\ \{z\} \mid z \in Z\}$;
  Sets$_D$:= $\mathcal{P}(X) \cup \mathcal{P}(Y)$;

Suppose $T_L = \{t_1, \ldots, t_k\}$, $T_R = \{t_{k+1}, \ldots, t_m\}$, $T_{D_L} = \{t_{m+1}, \ldots, t_n\}$.
  foreach sequence $(S_1, \ldots, S_k, S_{k+1}, \ldots, S_m, S_{m+1}, \ldots, S_n)$
    with $S_1, \ldots, S_k \in$ Sets$_L$, $S_{k+1}, \ldots, S_m \in$ Sets$_R$, $S_{m+1}, \ldots, S_n \in$ Sets$_D$
  do
    " make a copy S' of S ";

```
    " make a copy P' of P ";
    for i:=1 to n do
      foreach x ∈ S_i do
        " replace x =?_E t_x in NewEqs by x =?_E Nf(t_x + t_i) "
      od
    od;
    S' := S' ∪ NewEqs;
    CallRules(∃ u⃗ < S', P' >, Sol)
  od
end.
```

Function MakeMatchSet looks at all pairs (s_i, t_j) with s_i a prefix term from $Smd(s)$ and t_j a prefix term from $Smd(t)$ and determines via function Match which pairs match. Then the power set of the set of matching pairs is constructed. If $Smd(s)$ contains no variables, then each prefix term in $Smd(t)$ must match with some prefix term in $Smd(s)$, so only those set of pairs are kept in which each prefix term of $Smd(t)$ occurs at least once. And analogous if $Smd(t)$ contains no variables.

```
function MakeMatchSet(s, t):  SetOfMatchSets;
  Pairs := ∅;
  T_s := Smd(s) − (A ∪ V);
  T_t := Smd(t) − (A ∪ V);
  foreach (s_i, t_j) ∈ T_s × T_t do
    if Match(s_i, t_j) then " put (s_i, t_j) into Pairs " fi
  od;
  Sets := P(Pairs);
  if Smd(s) ∩ V = ∅
  then " remove all sets W from Sets with second(W) ⊉ T_t − T_s "
  fi;
  if Smd(t) ∩ V = ∅
  then " remove all sets W from Sets with first(W) ⊉ T_s − T_t "
  fi;
  MakeMatchSet := Sets
end.
```

Function Match tests whether two terms may be unifiable. If Match(s, t) is false then s and t are not unifiable, but if Match(s, t) is true it is not sure whether s and t are unifiable or not.

```
function Match(s, t) :  boolean;
  case
    s = as' and t = at' :  Match := true
    s = as' and t = bt' (a ≢ b) :  Match := false
  esac
end.
```

The algorithm UnPref produces pre-solutions. These are in the following way transformed into solutions:
Suppose $P \stackrel{\text{def}}{=} \{s_i =^?_{\mathcal{E}} t_i \mid i \in I\}$ is a problem and S is a pre-solution produced by UnPref with

input P, and suppose $V = Var(P)$. First remove all equations of the form $as =^?_{\mathcal{E}} at$. Now define the recursive specification E_α as:

$$E_\alpha \stackrel{\text{def}}{=} \{v = t_v \mid v =^?_{\mathcal{E}} t_v \in S\}.$$

Then define the solution α as:

$$\alpha \stackrel{\text{def}}{=} \{v := [\![\langle v \mid E_\alpha \rangle]\!] \mid v \in V\}.$$

Chapter 4

Properties of the algorithm UnPref

The algorithm must be correct, i.e. each rule must be sound and the set of rules must be complete.

Definition 4.1 *A rule* $\exists\,\vec{u}\,\langle\, S, P\,\rangle \Longrightarrow \{\ \exists\,\vec{u}_i\,\langle\, S_i, P_i\,\rangle \mid i \in I\ \}$ *is* **sound** *iff* $Sol(\exists\,\vec{u}\,\langle\, S,\, P\,\rangle) = \bigcup_{i\in I} Sol(\exists\,\vec{u}_i\ \langle\, S_i,\, P_i\,\rangle)$.

Definition 4.2 *A set SR of rules is* **complete** *iff for all problems* $\exists\,\vec{u}\,\langle\, S,\, P\,\rangle$*: if* $P \neq \emptyset$ *then at least one rule from SR is applicable.*

Also the algorithm must terminate, and the produced pre-solutions must be equivalent to their greatest subsets in solved form. And the algorithm should preferably produce a minimal and complete set of most general unifiers.

An execution of algorithm UnPref may be seen as a tree, where each node contains an intermediate result $\exists\,\vec{u}_i\ \langle\, S_i, P_i\,\rangle$, and branching nodes correspond to applications of rule Sum, where the branches point to all produced alternatives.

First it is proved that the set of rules is complete, and if $P = \emptyset$ then the subset of all equations $x =^?_{\mathcal{E}} t_x$ in S is in solved form.

Proposition 4.1 *If the algorithm UnPref produces an intermediate result* $\exists\,\vec{u}\,\langle\, S,\, P\,\rangle$ *with* $P \neq \emptyset$ *along an execution path, then at least one rule is applicable to* $\exists\,\vec{u}\,\langle\, S,\, P\,\rangle$.

Proof:
Take an equation $s =^?_{\mathcal{E}} t$ from P with $rep(s, S) = s'$ and $rep(t, S) = t'$ and check all possibilities for s' and t'. □

Theorem 4.2 *If the algorithm UnPref produces on termination a pre-solution* S*, then the set* $S' \subseteq S$*, defined as* $S' = \{\, x =^?_{\mathcal{E}} t_x \mid x \in \mathcal{V},\ t_x \in \mathcal{T},\ x =^?_{\mathcal{E}} t_x \in S\,\}$*, is in solved form.*

Proof:
All left sides of equations in S are different, so it remains to be proved that S' is guarded. Suppose S' is not guarded. Then there are equations $x_0 =^?_{\mathcal{E}} x_1 + t_0,\ x_1 =^?_{\mathcal{E}} x_2 + t_1,\ \ldots,$ $x_{n-1} =^?_{\mathcal{E}} x_n + t_{n-1},\ x_n =^?_{\mathcal{E}} x_0 + t_n$ in S' for some $n \geq 0$. Take an equation that was added last to S, say $x_i =^?_{\mathcal{E}} x_{i+1} + t_i$. This equation was added by rule 2 (Var) or 5 (Sum). But rule 2 (Var) adds only one equation, and rule 5 (Sum) adds at most one equation having variables as summands of its right hand side that appear also in equations already present in S. So the equations $x_0 =^?_{\mathcal{E}} x_1 + t_0, \ldots, x_{i-1} =^?_{\mathcal{E}} x_i + t_{i-1}, x_{i+1} =^?_{\mathcal{E}} x_{i+2} + t_{i+1}, \ldots, x_n =^?_{\mathcal{E}} x_0 + t_n$ were already present in S. But then $rep(x_{i+1} + t_i) = x_i + t_i'$ for some t_i', and rule 2 (Var) is not

applicable, and rule 5 (Sum) would not have added this equation to S. □

In the next part it is proved that the algorithm always terminates. First I prove that all intermediate terms consist of subterms of the original equations and new variables.

Proposition 4.3 *Suppose $\exists \vec{u} \langle S, P \rangle$ is an intermediate result of the algorithm UnPref with input P_0. If $s =^?_{\mathcal{E}} t \in (S \cup P)$, $s' \in Smd(s) \cup Smd(t)$, then s' is either a variable or a subterm of the terms in P_0.*

Proof:
Induction on the number n of rule applications.
$n = 0$: trivial.
$n > 0$:
Suppose the last rule applied is:
1 (Eq) or 3 (NoSol): trivial.
2 (Var) or 4 (Prefix): easy.
5 (Sum):
Suppose $s' = rep(s, S)$ and $t' = rep(t, S)$. Then (IH) all $s_i \in Smd(s')$ are subterms of terms in P_0 or a variable, and also all $t_j \in Smd(t')$. Procedure SumRule puts equations $s_i =^?_{\mathcal{E}} t_j$ into P with $as_i, at_j \in Smd(s_i =^?_{\mathcal{E}} t_j)$ and calls procedure SetsAlg with input $s' =^?_{\mathcal{E}} t'$. This procedure puts equations $x =^?_{\mathcal{E}} t_x$ into S with $Smd(t_x) \subseteq Smd(s' =^?_{\mathcal{E}} t') \cup \mathcal{V}$. □

Theorem 4.4 *The algorithm UnPref terminates on all inputs P consisting of equations over $\mathcal{T}$.*

Proof:
Consider an execution tree of UnPref. All rules create a finite number of alternatives, so an infinite execution tree must contain an infinite path of consecutive rule applications.
The algorithm can only proceed if $P \neq \emptyset$. On each equation in P only one rule is applicable. So P may be divided into groups Gi according to the applicable rule.

If there is an equation in G3 (NoSol) then the path terminates when this rule is executed. Application of rule 1 (Eq) decreases the number of equations in P by one and has no side effects. Side effects may arise when an equation is placed in S, because this may change representatives of other equations. So applications of rules Eq and NoSol are not relevant for determining whether an infinite path exists or not. That leaves rule 2 (Var), 4 (Prefix), and 5 (Sum). These may have the following effects:

Rule	New equations in P:	Possible side effects:
2 (Var)		$n \geq 0$ eqs. from G2 to other groups.
4 (Prefix)	1 eq. in arbitrary group	$n \geq 0$ eqs. from G4 to G1,
5 (Sum)	$n \geq 0$ eqs. in G4	$m \geq 0$ eqs. from G2 to other groups.

Application of rule 2 (Var) or 5 (Sum) decreases the number of equations in G2 ∪ G5. So a sequence of rule applications without use of rule 4 (Prefix) is always finite. And application of rule 4 adds only a finite number of equations to $P-$G4. So an infinite chain of rule applications must contain infinitely many applications of rule 4. But this is impossible because every application of rule 4 adds a different equation $s =^?_{\mathcal{E}} t$ to S with s and t subterms of the original equation set (Lemma 4.3). □

Now I prove that the algorithm is sound. First a proposition is proved concerning procedure SetsAlg.

Definition 4.3 *Suppose* $W = \{ (s_i, t_i) \mid i \in I) \}$ *is a set of pairs of terms. Then* $first(W) = \{ s_i \mid i \in I \}$ *and* $second(W) = \{ t_i \mid i \in I \}$.

Proposition 4.5 *Suppose* $\exists \vec{u} \langle S, P \rangle$, W, *and* $s =^?_{\mathcal{E}} t$ *are input to SetsAlg and* $[\![first(W)]\!]\alpha = [\![second(W)]\!]\alpha$. *Suppose* X, Y, Z, T_{D_L}, T_{D_R}, T_L, *and* T_R *correspond with the internal variables with the same names in SetsAlg. Then after execution of SetsAlg it holds for each produced alternative* $\exists \vec{u}, \vec{v} \langle S \cup S', P \cup P' \rangle$:

$$\bigcup\{ [\![t_x]\!]\alpha \mid x \in X \cup Z \text{ and } x =^?_{\mathcal{E}} t_x \in S' \} \cup \bigcup [\![T_L]\!]\alpha \cup \bigcup [\![T_{D_L}]\!]\alpha =$$

$$\bigcup\{ [\![t_y]\!]\alpha \mid y \in Y \cup Z \text{ and } y =^?_{\mathcal{E}} t_y \in S' \} \cup \bigcup [\![T_R]\!]\alpha \cup \bigcup [\![T_{D_R}]\!]\alpha.$$

Proof:
Procedure MakeNewVars distributes the same new variables to the variables in $X \cup Z$ and to $Y \cup Z$. Procedure DistrConstants distributes each term from T_L to a nonempty set of variables from $Y \cup Z$, each term from T_R to a nonempty set of variables from $X \cup Z$, and each term from T_{D_L} to a (possibly empty) set of variables from X or from Y. And $[\![first(W)]\!]\alpha = [\![second(W)]\!]\alpha$, so $[\![T_{D_L}]\!]\alpha = [\![T_{D_R}]\!]\alpha$. Then:

$$\bigcup\{ [\![t_x]\!]\alpha \mid x \in X \cup Z \text{ and } x =^?_{\mathcal{E}} t_x \in S' \} \subseteq$$
$$\bigcup\{ [\![t_y]\!]\alpha \mid y \in Y \cup Z \text{ and } y =^?_{\mathcal{E}} t_y \in S' \} \cup \bigcup [\![T_R]\!]\alpha \cup \bigcup [\![T_{D_R}]\!]\alpha,$$

and

$$\bigcup\{ [\![t_x]\!]\alpha \mid x \in X \cup Z \text{ and } x =^?_{\mathcal{E}} t_x \in S' \} \supseteq \bigcup [\![T_R]\!]\alpha.$$

Also

$$\bigcup\{ [\![t_y]\!]\alpha \mid y \in Y \cup Z \text{ and } y =^?_{\mathcal{E}} t_y \in S' \} \subseteq$$
$$\bigcup\{ [\![t_x]\!]\alpha \mid x \in X \cup Z \text{ and } x =^?_{\mathcal{E}} t_x \in S' \} \cup \bigcup [\![T_L]\!]\alpha \cup \bigcup [\![T_{D_L}]\!]\alpha,$$

and

$$\bigcup\{ [\![t_y]\!]\alpha \mid y \in Y \cup Z \text{ and } y =^?_{\mathcal{E}} t_y \in S' \} \supseteq \bigcup [\![T_L]\!]\alpha.$$

So

$$\bigcup\{ [\![t_x]\!]\alpha \mid x \in X \cup Z \text{ and } x =^?_{\mathcal{E}} t_x \in S' \} \cup \bigcup [\![T_L]\!]\alpha \cup \bigcup [\![T_{D_L}]\!]\alpha =$$
$$\bigcup\{ [\![t_x]\!]\alpha \mid x \in X \cup Z \text{ and } x =^?_{\mathcal{E}} t_x \in S' \} \cup \bigcup\{ [\![t_y]\!]\alpha \mid y \in Y \cup Z \text{ and } y =^?_{\mathcal{E}} t_y \in S' \} \cup$$
$$\cup \bigcup [\![T_L]\!]\alpha \cup [\![T_{D_R}]\!]\alpha \cup \bigcup [\![T_R]\!]\alpha =$$
$$\bigcup\{ [\![t_y]\!]\alpha \mid y \in Y \cup Z \text{ and } y =^?_{\mathcal{E}} t_y \in S' \} \cup \bigcup [\![T_R]\!]\alpha \cup \bigcup [\![T_{D_R}]\!]\alpha.$$

□

Theorem 4.6 *The rules 1 (Eq), 2 (Var), 3 (NoSol), 4 (Prefix), and 5 (Sum) are sound.*

Proof:
Rule 1 (Eq): easy.
Rules 2 (Var) and 3 (NoSol): trivial.
Rule 4 (Prefix):
Suppose $rep(s, S) \equiv as'$, $rep(t, S) \equiv at'$, and rule 4 is applied to $\exists \vec{u} \langle S, P \cup \{ s =^?_{\mathcal{E}} t \} \rangle$ producing $\{ \exists \vec{u} \langle S \cup \{ as' =^?_{\mathcal{E}} at' \}, P \cup \{ s' =^?_{\mathcal{E}} t' \} \rangle \}$.
From $rep(s, S) \equiv as'$ and $rep(t, S) \equiv at'$ it follows that there are equations $s_0 =^?_{\mathcal{E}} s_1$, $s_1 =^?_{\mathcal{E}} s_2$,

$\ldots$, $s_{k-1} =^?_{\mathcal{E}} s_k$ for some $k \geq 0$ in S with $s \equiv s_0,\ s_k \equiv as'$, and $t_0 =^?_{\mathcal{E}} t_1$, $t_1 =^?_{\mathcal{E}} t_2$, $\ldots$, $t_{l-1} =^?_{\mathcal{E}} t_l$ for some $l \geq 0$ with $t \equiv t_0$ and $t_l \equiv at'$. Now if $s =_{\mathcal{E}} t$ and $s_i =_{\mathcal{E}} s_i'$ for all $s_i =^?_{\mathcal{E}} s_i' \in S$ then $as' =_{\mathcal{E}} at'$, and if $as' =_{\mathcal{E}} at'$ and $s_i =_{\mathcal{E}} s_i'$ for all $s_i =^?_{\mathcal{E}} s_i' \in S$ then $s =^?_{\mathcal{E}} t$. So α is a solution of $\exists\, \vec{u}\, \langle\, S,\ P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle$ iff α is a solution of $\exists\, \vec{u}\, \langle\, S \cup \{as' =^?_{\mathcal{E}} at'\},\ P\, \rangle$. Then if α is a solution of $\exists\, \vec{u}\, \langle\, S \cup \{as' =^?_{\mathcal{E}} at'\},\ P \cup \{s' =^?_{\mathcal{E}} t'\}\rangle$, α is also a solution of $\exists\, \vec{u}\, \langle\, S,\ P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle$.
Now suppose α is a solution of $\exists\, \vec{u}\, \langle\, S,\ P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle$.
Define $V = Var(S \cup P \cup \{s =^?_{\mathcal{E}} t\}) - \vec{u}$, and $\alpha' = \alpha_{|V}$. Then there is a semantic substitution β with $\mathrm{Dom}(\beta) \subseteq \vec{u}$ such that $[\![s]\!]\alpha'\beta = [\![t]\!]\alpha'\beta$ and $[\![s_i]\!]\alpha'\beta = [\![t_i]\!]\alpha'\beta$ for all $s_i =^?_{\mathcal{E}} t_i \in S \cup P$. Also $[\![as']\!]\alpha'\beta = [\![at']\!]\alpha'\beta$ (see above). So $\{(a, [\![s']\!]\alpha'\beta)\} = [\![as']\!]\alpha'\beta = [\![at']\!]\alpha'\beta = \{(a, [\![t']\!]\alpha'\beta)\}$. Then $[\![s']\!]\alpha'\beta = [\![t']\!]\alpha'\beta$.
So α is a solution of $\exists\, \vec{u}\, \langle\, S \cup \{as' =^?_{\mathcal{E}} at'\},\ P \cup \{s' =^?_{\mathcal{E}} t'\}\, \rangle$.
Rule 5 (Sum):
Suppose $rep(s, S) = s'$ and $rep(t, S) = t'$.
Define $V_s = Smd(s') \cap \mathcal{V},\ V_t = Smd(t') \cap \mathcal{V},\ T_s = Smd(s') - \mathcal{V},\ T_t = Smd(t') - \mathcal{V}$. Suppose rule 5 is applied to $\exists\, \vec{u}\, \langle\, S,\ P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle$ producing a number of alternatives $\exists\, \vec{u},\ \vec{v}\, \langle\, S \cup S',\ P \cup P'\, \rangle$ with
S' of the form $\{x =^?_{\mathcal{E}} t_x \mid x \in X \cup Y \cup Z\}$, and
P' of the form $\{s_i =^?_{\mathcal{E}} t_j \mid (s_i, t_j) \in W\}$ for a $W \subseteq T_s \times T_t$.
Define $V = Var(S \cup P \cup \{s =^?_{\mathcal{E}} t\}) - \vec{u}$.
1) Suppose α is a solution of an alternative $\exists\, \vec{u},\ \vec{v}\, \langle S \cup S',\ P \cup P'\rangle$, $P' = \{s_i =^?_{\mathcal{E}} t_j \mid (s_i, t_j) \in W\}$. Define $\alpha' = \alpha_{|V}$. Then there is a β with $\mathrm{Dom}(\beta) \subseteq \vec{u} \cup \vec{v}$ s.t. $[\![first(W)]\!]\alpha'\beta = [\![second((W)]\!]\alpha'\beta$. Then $[\![x]\!]\alpha'\beta = [\![t_x]\!]\alpha'\beta$ for all $x =^?_{\mathcal{E}} t_x \in S'$. Define $\beta' = \beta_{|\vec{u}}$. Suppose T_L, T_R, T_{D_L}, and T_{D_R} have the same values as the parameters with the same names in procedure DistrConstants. Then:

$$
\begin{aligned}
[\![s']\!]\alpha'\beta' = [\![s']\!]\alpha'\beta = [\![\sum_{x \in X \cup Z} t_x + \sum_{t_L \in T_L} t_L + \sum_{t_D \in T_{D_L}} t_{D_L}\ ]\!]\alpha'\beta = \\
= \bigcup_{x \in X \cup Z} [\![t_x]\!]\alpha'\beta \cup \bigcup [\![T_L]\!]\alpha'\beta \cup \bigcup [\![T_{D_L}]\!]\alpha'\beta = \text{ (Proposition 4.5)} \\
= \bigcup_{y \in Y \cup Z} [\![t_y]\!]\alpha'\beta \cup \bigcup [\![T_R]\!]\alpha'\beta \cup \bigcup [\![T_{D_R}]\!]\alpha'\beta \\
= [\![t']\!]\alpha'\beta = [\![t']\!]\alpha'\beta'.
\end{aligned}
$$

So α is a solution of $\exists\, \vec{u}\, \langle\, S,\ P \cup \{s' =^?_{\mathcal{E}} t'\}\, \rangle$. Then α is also a solution of $\exists\, \vec{u}\, \langle\, S,\ P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle$ (see correctness proof of rule 4).
2) Suppose α is a solution of $\exists\, \vec{u}\, \langle S,\ P \cup \{s =^?_{\mathcal{E}} t\}\, \rangle$.
Then α is also a solution of $\exists\, \vec{u}\, \langle\, S,\ P \cup \{s' =^?_{\mathcal{E}} t'\}\rangle$ (see correctness proof of rule 4). Define $\alpha' = \alpha_{|V}$. Then there is a β with $\mathrm{Dom}(\beta) \subseteq \vec{u}$ such that $[\![s_*]\!]\alpha'\beta = [\![t_*]\!]\alpha'\beta$ for all $s_* =^?_{\mathcal{E}} t_* \in S \cup P$. Then:

$$\bigcup [\![V_s]\!]\alpha'\beta \cup \bigcup [\![T_s]\!]\alpha'\beta = \bigcup [\![V_t]\!]\alpha'\beta \cup \bigcup [\![T_t]\!]\alpha'\beta.$$

Now an alternative $\exists\, \vec{u},\ \vec{v}\, \langle\, S \cup S',\ P \cup P'\rangle$ is constructed such that α is a solution of this alternative.
Suppose $s_i \in T_s$ and $[\![s_i]\!]\alpha'\beta \notin \bigcup_{y \in V_t} [\![y]\!]\alpha'\beta$. Then there must be a $t_j \in T_t$ such that $[\![s_i]\!]\alpha'\beta = [\![t_j]\!]\alpha'\beta$. Now define sets of pairs (s_i, t_j) based on this property:

$$W_s = \{\ (s_i, t_j) \mid s_i \in T_s - \mathcal{A},\ [\![s_i]\!]\alpha'\beta \notin \bigcup [\![V_t]\!]\alpha'\beta,\ t_j \in T_t - \mathcal{A},\ [\![s_i]\!]\alpha'\beta = [\![t_j]\!]\alpha'\beta\ \},$$

$$W_t = \{\ (s_i, t_j) \mid t_j \in T_t - \mathcal{A},\ [\![t_j]\!]\alpha'\beta \notin \bigcup [\![V_s]\!]\alpha'\beta,\ s_i \in T_s - \mathcal{A},\ [\![s_i]\!]\alpha'\beta = [\![t_j]\!]\alpha'\beta\ \},$$

and $W = W_s \cup W_t$.
Define $P' \stackrel{\text{def}}{=} \{\ s_i =^?_{\mathcal{E}} t_j \mid (s_i, t_j) \in W\ \}$, $X = V_s - V_t$, $Y = V_t - V_s$, $Z = V_s \cap V_t$, and $C_D = T_s \cap T_t \cap \mathcal{A}$. Choose a set $\vec{v}$ of fresh variables with

$$\vec{v} = \{\ v_{\{x,y\}} \mid x \in X \text{ and } y \in Y \cup Z\ \} \cup \{\ v_{\{y,z\}} \mid y \in Y \text{ and } z \in Z\ \} \cup \{\ v_{\{z\}} \mid z \in Z\ \}.$$

Now define S' as:

$$\begin{aligned} S' \quad = \quad & \{\ x =^?_{\mathcal{E}} \sum_{y \in Y} v_{\{x,y\}} + \sum_{z \in Z} v_{\{x,z\}} + \sum T_x \mid x \in X\ \} \cup \\ & \{\ y =^?_{\mathcal{E}} \sum_{x \in X} v_{\{x,y\}} + \sum_{z \in Z} v_{\{y,z\}} + \sum T_y \mid y \in Y\ \} \cup \\ & \{\ z =^?_{\mathcal{E}} v_{\{z\}} + \sum_{x \in X} v_{\{x,z\}} + \sum_{y \in Y} v_{\{y,z\}} + \sum T_z \mid z \in Z\ \}. \end{aligned}$$

where

$$T_x = \{\ t \mid t \in T_t \text{ and } [\![t]\!]\alpha'\beta \in ([\![x]\!]\alpha'\beta - \bigcup [\![Z]\!]\alpha'\beta)\ \} \text{ for all } x \in X,$$

$$T_y = \{\ t \mid t \in T_s \text{ and } [\![t]\!]\alpha'\beta \in ([\![y]\!]\alpha'\beta - \bigcup [\![Z]\!]\alpha'\beta)\ \} \text{ for all } y \in Y.$$

Compute all T_z as follows:
$T_z := \emptyset$ for all $z \in Z$; for each $t \in T_s - (\mathit{first}(W) \cup C_D)$ with $[\![t]\!]\alpha'\beta \subseteq (\bigcup [\![Z]\!]\alpha'\beta - [\![Y]\!]\alpha'\beta)$ choose a z with $[\![t]\!]\alpha'\beta \subseteq [\![z]\!]\alpha'\beta$, and add t to T_z. Analogous for each $t \in T_t - (\mathit{second}(W) \cup C_D)$.
The alternative $\exists\, \vec{u},\ \vec{v}\, \langle\, S \cup S',\ P \cup P' \,\rangle$ exists because:
- the right sets of new variables are created,
- each constant $a \in T_s - T_t$ is attributed to at least one variable from V_t, and analogous for $a \in T_t - T_s$,
- each constant $a \in T_s \cap T_t$ is attributed to either a subset of X or a subset of Y,
- each prefix term $as_i \in T_s$ is attributed to at least one variable from V_t, or is present in an equation $as_i =^?_{\mathcal{E}} at_j$ with $at_j \in T_t$, and analogous for $at_j \in T_t$,
- if $s_i \in T_s$ is attributed to at least one variable from V_t and there is no equation $s_i =^?_{\mathcal{E}} t_j$, then s_i is either attributed to a nonempty subset of Y or to one variable from Z. Analogous for $t_j \in T_t$,
- if $s_i \in T_s$ is attributed to at least one variable from $V_s \cup V_t$ and there is an equation $s_i =^?_{\mathcal{E}} t_j$, then s_i is either attributed to a subset of X or of Y. The same holds for $t_j \in T_t$.

Now it must still be proved that α is a solution of $\exists\, \vec{u}\, \langle\, S \cup S',\ P \cup P' \,\rangle$. Therefore the semantic substitution γ is defined:

$$\begin{aligned} \gamma \quad = \quad & \{\ v_{\{x,y\}} := [\![x]\!]\alpha'\beta \cap [\![y]\!]\alpha'\beta \mid x \in X, y \in Y\ \} \cup \\ & \{\ v_{\{x,z\}} := [\![x]\!]\alpha'\beta \cap [\![z]\!]\alpha'\beta \mid x \in X \cup Y, z \in Z\ \} \cup \\ & \{\ v_{\{z\}} := [\![z]\!]\alpha'\beta \mid z \in Z\ \} \end{aligned}$$

Suppose $x =^?_{\mathcal{E}} t_x \in S'$ and $x \in X$. Then:

$$[\![t_x]\!]\alpha'\beta\gamma = [\![\sum_{y \in Y} v_{\{x,y\}} + \sum_{z \in Z} v_{\{x,z\}} + \Sigma T_x\ ]\!]\alpha'\beta\gamma \ = \bigcup_{y \in Y \cup Z} [\![v_{\{x,y\}}]\!]\gamma \cup \bigcup [\![T_x]\!]\alpha'\beta =$$

$$= \bigcup_{y \in Y \cup Z} ([\![x]\!]\alpha'\beta \cap [\![y]\!]\alpha'\beta) \cup ([\![x]\!]\alpha'\beta - \bigcup_{z \in Z} [\![z]\!]\alpha'\beta) = [\![x]\!]\alpha'\beta = [\![x]\!]\alpha'\beta\gamma.$$

In the same way: $[\![y]\!]\alpha'\beta\gamma = [\![t_y]\!]\alpha'\beta\gamma$ for all $y \in Y$, and

$$[\![t_z]\!]\alpha'\beta\gamma = [\![v_{\{z\}} + \sum_{x\in X} v_{\{x,z\}} + \sum_{y\in Y} v_{\{y,z\}} + T_z]\!]\alpha'\beta\gamma = [\![z]\!]\alpha'\beta \cup ([\![z]\!]\alpha'\beta \cap \ldots) = [\![z]\!]\alpha'\beta.$$

Also $[\![s_i]\!]\alpha'\beta\gamma = [\![t_j]\!]\alpha'\beta\gamma$ for all $s_i =^?_{\mathcal{E}} t_j$ in $S \cup P \cup P'$.
So α is a solution of $\exists\, \vec{u},\, \vec{v}\, \langle\, S \cup S',\, P \cup P' \,\rangle$.

□

In the next part it will be proved that at termination of UnPref with result $\exists\, \vec{u}\, \langle\, S, P \,\rangle$ the set S is decomposed. It will also be proved that a decomposed set of equations is equivalent to its greatest subset in normal form.

Proposition 4.7 *Suppose $\exists\, \vec{u}\, \langle\, S,\, P \,\rangle$ is an intermediate result of the algorithm UnPref. Then:*

1. *The left hand sides of all equations in S are different,*
2. *S does not contain a circular subproblem,*
3. *S contains only equations of the form $x =^?_{\mathcal{E}} t$ $(x \in \mathcal{V}, t \in \mathcal{T})$ and $as =^?_{\mathcal{E}} at$ $(a \in \mathcal{A}, s, t \in \mathcal{T})$.*

Proof:
Induction on the number n of rule applications.
$n = 0$: $S = \emptyset$.
$n > 0$: Check all rules.
Rules 1 (Eq) and 3 (NoSol): trivial.
Rules 2 (Var), 4 (Prefix), and 5 (Sum): easy. □

Proposition 4.8 *Suppose rule Sum is applied in an execution of UnPref, and procedure SetsAlg is called with input $\exists\, \vec{u}\, \langle\, S,\, P \,\rangle$, W, and $s =^?_{\mathcal{E}} t$ during the execution of algorithm UnPref, and $\exists\, \vec{u},\, \vec{v}\, \langle\, S \cup S',\ P \rangle$ is one of the produced alternatives.*
Suppose $s' = rep(s, S \cup S')$ and $t' = rep(t, S \cup S')$.
Then there is a set W' of pairs (s_i, t_j) such that

i) $first(W') = Smd(s')$,

ii) $second(W') = Smd(t')$,

iii) if $(s_i, t_j) \in W'$ then $s_i \equiv t_j$ or $s_i =^?_{\mathcal{E}} t_j \in P$.

Proof:
Define $V_s = Smd(s) \cap \mathcal{V},\ V_t = Smd(t) \cap \mathcal{V},\ T_s = Smd(s) - \mathcal{V},\ T_t = Smd(t) - \mathcal{V}$.
Define $W' \stackrel{\text{def}}{=} W \cup \{\ (v, v) \mid v \in \vec{v}\ \} \cup \{\ (s_i, s_i) \mid s_i \in (T_s - first(W)) \cup (T_t - second(W))\ \}$.
Claim: W' has the desired properties.
Proof of claim:
i) $Smd(s') = first(W')$.
i1) $Smd(s') \subseteq first(W')$.
Suppose $s_i \in Smd(s')$. Then s_i is a nonvariable summand of $Smd(s)$, or $x \in Smd(s)$, $x =^?_{\mathcal{E}} t_x \in S'$, and $s_i \in Smd(t_x)$. If $s_i{}'$ is a nonvariable summand of $Smd(s)$, then $(s_i, s_i) \in W'$, so $s_i \in first(W')$. If $x =^?_{\mathcal{E}} t_x \in S'$ and $s_i \in Smd(t_x)$, then $s_i \in \vec{v}$ or $s_i \in T_s \cup T_t$ (see procedure MakeNewVars and DistrConstants), so $s_i \in first(W')$. So $Smd(s') \subseteq first(W')$.
i2) $first(W') \subseteq Smd(s')$.
It holds that $first(W) \subseteq Smd(s')$ (see function MakeMatchSet), $\{\ v \mid v \in \vec{v}\ \} \subseteq Smd(s')$ (see

procedure MakeNewVars), and $T_s \subseteq Smd(s')$, because SetsAlg adds only equations to S' of the form $x =^?_{\mathcal{E}} t_x$. So consider a term $s_i \in first(W')$ with $s_i \in T_t - second(W)$.
If $s_i \in T_s$ then $s_i \in Smd(s')$. So suppose $s_i \notin T_s$. Then there is an $x \in V_s$ such that $x =^?_{\mathcal{E}} t_x \in S'$ and $s_i \in Smd(t_x)$ (see procedure DistrConstants). So again $s_i \in Smd(s')$.
It follows that $first(W') \subseteq Smd(s')$.
ii) $Smd(t') = second(W')$: analogous.
iii) If $(s_i, t_j) \in W'$, then $s_i \equiv t_j$ or $s_i =^?_{\mathcal{E}} t_j \in P$.
Trivial for $(s_i, t_j) \in \{\ (v, v) \mid v \in \vec{v}\ \} \cup \{\ (s_i, s_i) \mid s_i \in (T_s - first(W)) \cup (T_t - second(W))\ \}$.
Suppose $(s_i, t_j) \in W$. Then $s_i =^?_{\mathcal{E}} t_j \in P$ (see procedure SumRule). □

Lemma 4.9 *Consider a path in an execution tree of UnPref. Number the nodes, and call the intermediate results $\exists \vec{u}_0 \langle\ S_0,\ P_0\ \rangle$, $\exists \vec{u}_1 \langle\ S_1,\ P_1\ \rangle, \cdots$. Suppose $\exists\, \vec{u}_n \langle\ S_n, P_n\ \rangle$ is an intermediate result . Then for each $as =^?_{\mathcal{E}} at$ in S_n the condition $C(n, as =^?_{\mathcal{E}} at)$ holds, where $C(n, as =^?_{\mathcal{E}} at)$ is defined as:*
There is a set W of pairs of terms s.t. $Smd(rep(s, S_n)) = first(W)$, $Smd(rep(t, S_n)) = second(W)$, and for all $(s_i, t_j) \in W$: $rep(s_i, S_n) \equiv rep(t_j, S_n)$ or there are s_i', t_j' with $(s_i' =^?_{\mathcal{E}} t_j' \in P_n$, $rep(s_i') \equiv s_i$, and $rep(t_j') \equiv t_j)$.

Proof:
Induction on the number n of rule applications.
$n = 0$: $S = \emptyset$.
$n > 0$:
IH: The condition C$(n-1, as =^?_{\mathcal{E}} at)$ holds for all $as =^?_{\mathcal{E}} at \in S_{n-1}$.
The condition can be made invalid in three ways:
a) by removing an equation from P_{n-1} that corresponds to a pair in a set W,
b) by adding an equation $as =^?_{\mathcal{E}} at$ to S_n that has no set W,
c) by adding an equation to S_{n-1} that alters representatives of terms of equations in P_n.
Suppose the nth rule application used rule:
1 (Eq):
Suppose $s =^?_{\mathcal{E}} t$ is removed from P_{n-1}, and $s =^?_{\mathcal{E}} t \in P_{n-1}$ was needed for the truth of C$(n-1, as_1 =^?_{\mathcal{E}} at_1)$ because $(rep(s, S_{n-1}), rep(t, S_{n-1})) \in W$. But $rep(s, S_{n-1}) \equiv rep(t, S_{n-1})$, so also $rep(s, S_n) \equiv rep(t, S_n)$, and C$(n, as_1 =^?_{\mathcal{E}} at_1)$ holds.
2 (Var):
Suppose $s =^?_{\mathcal{E}} t$ is removed from P_{n-1}, $rep(s, S_{n-1}) \equiv x$, $rep(t, S_{n-1}) \equiv t_x$, and $x =^?_{\mathcal{E}} t_x$ is added to S_{n-1}. Suppose $s =^?_{\mathcal{E}} t$ was essential for the truth of C$(n-1, as_1 =^?_{\mathcal{E}} at_1)$ because $(x, t_x) \in W$. Replace (x, t_x) in W by (t_x, t_x). Then C$(n, as_1 =^?_{\mathcal{E}} at_1)$ holds because $x =^?_{\mathcal{E}} t_x \in S_n$ and so $rep(s, S_n) \equiv rep(t, S_n)$.
Suppose $s' =^?_{\mathcal{E}} t' \in P$ is essential for the truth of C$(n-1, bs_2 =^?_{\mathcal{E}} bt_2)$ and $x \in Smd(rep(s', S_{n-1}))$. Then replace (s', t') in the corresponding W by $(rep(s', S_n), t')$.
3 (NoSol): trivial.
4 (Prefix):
Suppose $s =^?_{\mathcal{E}} t$ is removed from P_{n-1}, $rep(s, S_{n-1}) = as'$, $rep(t, S_{n-1}) = at'$, $as' =^?_{\mathcal{E}} at'$ is added to S_{n-1}, and $s' =^?_{\mathcal{E}} t'$ to P. Then C$(n, as' =^?_{\mathcal{E}} at')$ holds.
Suppose $s =^?_{\mathcal{E}} t$ was essential for the truth of C$(n-1, bs_1 =^?_{\mathcal{E}} bt_1)$ because $(as', at') \in W$. Replace (as', at') in W by (at', at'). Then C$(n, bs_1 =^?_{\mathcal{E}} bt_1)$ holds because $as' =^?_{\mathcal{E}} at' \in S_n$ and so $rep(s, S_n) \equiv rep(t, S_n)$.
Suppose $s_2 =^?_{\mathcal{E}} t_2 \in P$ is essential for the truth of C$(n-1, bs_3 =^?_{\mathcal{E}} bt_3)$, and $as' \in Smd(rep(s_2)) \cup Smd(rep(t_2))$. Then replace $(rep(s_2, S_{n-1}), rep(t_2, S_{n-1}))$ in the corresponding W by $(rep(s_2, S_n), rep(t_2, S_n))$.
5 (Sum):

Suppose $s =^?_{\mathcal{E}} t$ is removed from P_{n-1}, $rep(s, S_{n-1}) = s'$, $rep(t, S_{n-1}) = t'$. Suppose $s =^?_{\mathcal{E}} t$ was needed for the truth of $\mathrm{C}(n-1, as_1 =^?_{\mathcal{E}} at_1)$ because $(s', t') \in W$. Procedure SumRule puts equations $x =^?_{\mathcal{E}} t_x$ in S_n and $bs_i =^?_{\mathcal{E}} bt_j$ in P_n in such a way that there is a set W_1 with:
$first(W_1) = Smd(s')$, $second(W_1) = Smd(t')$, and if $(s_i, t_j) \in W_1$ then $s_i \equiv t_j$ or there are s_i', t_j' with $rep(s_i', S_n) \equiv s_i$, $rep(t_j', S_n) \equiv t_j$ and $s_i =^?_{\mathcal{E}} t_j \in P_n$ (Proposition 4.8).
Define $W' \stackrel{\text{def}}{=} \{\ (rep(s_i, S_n),\ rep(t_j, S_n)) \mid (s_i, t_j) \in W - \{(s', t')\}\ \} \cup W_1$.
Then $Smd(rep(s, S_n)) = first(W')$ and $Smd(rep(t, S_n)) = second(W')$, so $\mathrm{C}(n, as_1 =^?_{\mathcal{E}} at_1)$ holds.
Suppose $x =^?_{\mathcal{E}} t_x$ is added to S_{n-1}, $s_1 =^?_{\mathcal{E}} t_1 \in P$ is essential for the truth of $\mathrm{C}(n-1, as_2 =^?_{\mathcal{E}} at_2)$, and $x \in Smd(rep(s_1, S_{n-1}) \cup Smd(rep(t_1, S_{n-1}))$. Then replace $(rep(s_1, S_{n-1}), rep(t_1, S_{n-1}))$ in W by $(rep(s_1, S_n), rep(t_1, S_n))$. □

If S is a pre-solution produced by UnPref on some input P, then S is decomposed (Propositions 4.7 and 4.8). Now it is proved that a decomposed set is equivalent to its greatest subset in solved form. The proof of the theorem follows the same course as the proof of a similar theorem for (differently defined) decomposed problems in syntactic unification [Col84a].

Theorem 4.10 *A decomposed set S is equivalent to its greatest subset in solved form.*

Proof:
First the theorem is proved for decomposed sets S with a special property. Call a decomposed set S **simple** if for each equation $s =^?_{\mathcal{E}} t \in S$: $rep(s, S) = t$, and if s is not a variable then $d(t) \leq d(s)$.
Suppose S is a simple decomposed set. Define S' as the greatest subset of S that is in solved form. Suppose α is a solution of S'. Then it is proved by induction on the depth $d(s)$ of s that α is a solution of each $s =^?_{\mathcal{E}} t$ in S.
$d(s) = 1$:
Then s is a variable, so $s =^?_{\mathcal{E}} t$ is in S' and α is a solution of S'.
$d(s) > 1$:
Suppose $s = as'$ and $t = at'$.
S is decomposed, so $Smd(rep(s')) = Smd(rep(t'))$. Suppose $s' \equiv s_1 + \ldots + s_m$, $rep(s_1) = s_1', \ldots, rep(s_m) = s_m'$ $(m \geq 1)$. Then for all i with $1 \leq i \leq m$: $s_i \equiv s_i'$, or $s_i =^?_{\mathcal{E}} s_i' \in S'$ with $\mathrm{d}(s_i) \geq \mathrm{d}(s_i')$. And $d(s_i) < d(s)$, so (IH) α is a solution of all equations $s_i =^?_{\mathcal{E}} s_i'$ in S. Then α is a solution of $s' =^?_{\mathcal{E}} rep(s')$. Analogous: α is a solution of $t' =^?_{\mathcal{E}} rep(t')$. So α is a solution of $s' =^?_{\mathcal{E}} t'$, and also of $as' =^?_{\mathcal{E}} at'$.

Now suppose that S is an arbitrary decomposed set, not necessarily simple. Consider all terms t that occur as a right hand side of an equation in S but not as a left hand side. Each term t defines an equivalence class R_t of terms in S that have t as their representative. A semantic substitution α is a solution of S iff α is a solution of all equations $t_1 =^?_{\mathcal{E}} t_2$ with $t_1, t_2 \in R_t$ for each set R_t in S. Now choose a term t_* in each R_t: if R_t contains only variables then $t_* = t$, else t_* becomes a non-variable term in R_t with minimal depth. Now α is a solution of S iff α is a solution of all equations $t' =^?_{\mathcal{E}} t_*$ with $t' \in R_t$ for each R_t in S. So S is equivalent to the problem $T = \{\ t' =^?_{\mathcal{E}} t_* \mid t' \in R_t - \{t_*\},\ R_t$ an equivalence class in $S\ \}$. T is decomposed because S is decomposed. Suppose S' and T' are the maximal sets in solved form in S and T respectively. T is a simple decomposed problem, so T' is equivalent to T. So $\mathrm{Sol}(T') = \mathrm{Sol}(T) = \mathrm{Sol}(S)$. Also $\mathrm{Sol}(S) \subseteq \mathrm{Sol}(S')$ because $S \supseteq S'$. But S' and T' have the same elimination set, so if $\mathrm{Sol}(T') \subseteq \mathrm{Sol}(S')$ then $\mathrm{Sol}(T') = \mathrm{Sol}(S')$ [Col84a]. So $\mathrm{Sol}(T') = \mathrm{Sol}(S')$, and also $\mathrm{Sol}(S) = \mathrm{Sol}(S')$. □

Chapter 5

A matching algorithm for substitution

The algorithm UnPref does not always produce a minimal set of solutions. One example is the problem $\{ax + aax =^?_{\mathcal{E}} ay + aay\}$. This problem leads to more solutions, two of which are given below:

$$
\begin{array}{ll}
 & < \emptyset,\ \{ax + aax =^?_{\mathcal{E}} ay + aay\} > \stackrel{Sum}{\Longrightarrow}(1)(2) \\
(1) & < \emptyset,\ \{ax =^?_{\mathcal{E}} ay,\ aax =^?_{\mathcal{E}} aay\} > \stackrel{Prefix}{\Longrightarrow} \\
 & < \{ax =^?_{\mathcal{E}} ay\},\ \{x =^?_{\mathcal{E}} y,\ aax =^?_{\mathcal{E}} aay\} > \stackrel{Var}{\Longrightarrow} \\
 & < \{ax =^?_{\mathcal{E}} ay,\ x =^?_{\mathcal{E}} y\},\ \{aax =^?_{\mathcal{E}} aay\} > \stackrel{Prefix}{\Longrightarrow} \\
 & < \{ax =^?_{\mathcal{E}} ay,\ x =^?_{\mathcal{E}} y,\ aax = aay\},\ \{ax =^?_{\mathcal{E}} ay\} > \stackrel{Eq}{\Longrightarrow} \\
 & < \{ax =^?_{\mathcal{E}} ay,\ x =^?_{\mathcal{E}} y,\ aax = aay\},\ \emptyset > . \\
(2) & < \emptyset,\ \{ax =^?_{\mathcal{E}} aay,\ aax =^?_{\mathcal{E}} ay\} > \stackrel{Prefix}{\Longrightarrow} \\
 & < \{ax =^?_{\mathcal{E}} aay\},\ \{x =^?_{\mathcal{E}} ay,\ aax =^?_{\mathcal{E}} ay\} > \stackrel{Var}{\Longrightarrow} \\
 & < \{ax =^?_{\mathcal{E}} aay,\ x =^?_{\mathcal{E}} ay\},\ \{aax =^?_{\mathcal{E}} ay\} > \stackrel{Prefix}{\Longrightarrow} \\
 & < \{ax =^?_{\mathcal{E}} aay,\ x =^?_{\mathcal{E}} ay,\ aax =^?_{\mathcal{E}} ay\},\ \{ax =^?_{\mathcal{E}} y\} > \stackrel{Var}{\Longrightarrow} \\
 & < \{ax =^?_{\mathcal{E}} aay,\ x =^?_{\mathcal{E}} ay,\ aax =^?_{\mathcal{E}} ay,\ y =^?_{\mathcal{E}} aay\},\ \emptyset > .
\end{array}
$$

Now define $E_\alpha \stackrel{\text{def}}{=} \{x = y\}$, $\alpha \stackrel{\text{def}}{=} \{x := [\![\langle x|E_\alpha\rangle]\!],\ y := [\![\langle y|E_\alpha\rangle]\!]\}$, $E_\beta \stackrel{\text{def}}{=} \{x = ay,\ y = aay\}$, $\beta \stackrel{\text{def}}{=} \{x := [\![\langle x|E_\beta\rangle]\!],\ y := [\![\langle y|E_\beta\rangle]\!]\}$. Then $\alpha \leq_V \beta$, for take $\gamma \stackrel{\text{def}}{=} \{x := [\![\langle x|E_\gamma\rangle]\!],\ y := [\![\langle x|E_\gamma\rangle]\!]\}$, $E_\gamma \stackrel{\text{def}}{=} \{x = ax\}$). So in the rest of this section an algorithm is given to determine whether a semantic substitution is more general than another semantic substitution.

5.1 The algorithm MoreGeneral

This algorithm determines whether a semantic substitution α, given as a recursive specification E_α, is more general w.r.t. some set $V \subseteq \mathcal{V}$ than a semantic substitution β, given as E_β. It is in fact a matching algorithm. It tries to find out whether there exists a substitution on the variables of E_α that makes E_α semantically equal to E_β. The recursive specifications E_α and E_β must first be modified in such a way that their variables do not overlap, they are in linear form, and E_β has a minimal number of recursion variables.

Definition 5.1 *A term t is* **linear** *if $t =_{\mathcal{E}} \sum_{i=1}^{m} a_i x_i + \sum_{j=1}^{n} b_j$ for some m, n with $m \geq 0, n \geq 0$, $a_i, b_j \in \mathcal{A}, x_i \in \mathcal{V}$. A recursive specification $E = \{x = t_x \mid x \in X\}$ is* **linear** *if all t_x are linear.*

Choose two substitutions σ_α and σ_β such that each variable in E_α and E_β is substituted by a fresh variable, $Var(E_\alpha\sigma_\alpha) \cap Var(E_\beta\sigma_\beta) = \emptyset$, $E_\alpha\sigma_\alpha$ and $E_\beta\sigma_\beta$ are linear, and $E_\beta\sigma_\beta$ has a minimal number of variables. To determine which recursion variables in E_β produce the same process the algorithm of Groote and Vaandrager [GV90] may be used. Define ${E_\alpha}' = E_\alpha\sigma_\alpha$ and ${E_\beta}' = E_\beta\sigma_\beta$. Then all equations in ${E_\alpha}'$ are of the form $x = t_x$ with all summands of t_x constants, variables from $Free Var({E_\alpha}')$, and terms ay with $a \in \mathcal{A}$ and $y \in Rec Var({E_\alpha}')$, and the same for ${E_\beta}'$. Now $(\sigma_\alpha, {E_\alpha}')$ and $(\sigma_\beta, {E_\beta}')$ are input to function MoreGeneral.

The algorithm tests recursively whether pairs (x, y) with x a recursion variable from ${E_\alpha}'$ and y from ${E_\beta}'$ describe two processes $[\![\langle x|{E_\alpha}'\rangle]\!]$ and $[\![\langle y|{E_\beta}'\rangle]\!]$ such that $[\![\langle x|{E_\alpha}'\rangle]\!]$ may be more general than $[\![\langle y|{E_\beta}'\rangle]\!]$. Pairs (x, y) to be tested are placed in the set ToTest. As soon as a pair is tested it is moved to the set Tested. In this way each pair (x, y) is tested at most once.

The array Max keeps for each free variable u in ${E_\alpha}'$ the set of terms that can maximally be attributed to u. The set Min is filled with pairs (U, T) where T contains the set of terms such that each term must be attributed to at least one u in U.

Function MoreGeneral initializes Min, Max, ToTest, and Tested using function Initialize, and places all pairs (x, y) of recursion variables corresponding to a variable v in the original problem into ToTest. Then MatchPair is called for each of these pairs (x, y).

```
function MoreGeneral( (σα,Eα), (σβ,Eβ) ):  boolean;
  if Eα = ∅ then Result := true else Result := false;
  if Dom(σα) ⊆ Dom(σβ) then
    V  := Dom(σα);
    FVα := FreeVar(Eα);
    Initialize(Min, Max, ToTest, Tested);
    foreach v ∈ V do
      var1 := vσα;
      var2 := vσβ;
      "put (var1,var2) into ToTest"
    od;
    foreach v ∈ V do
      var1 := vσα;
      var2 := vσβ;
      MatchPair(var1, var2, Max, Min, ToTest, Tested, Result)
    od
  fi;
  MoreGeneral := Result
end.

proc Initialize(Max :  array[FVα] of TermSet ∪ {"undefd"},
                Min :  Set of (VarSet,TermSet),
                ToTest, Tested :  Set of Var);
  foreach u ∈ FVα do Max[u] := "undefd" od;
  Min := ∅;
  ToTest := ∅;
  Tested := ∅
```

end.

Procedure MatchPair first checks if (x, y) is in ToTest. If this is the case, it fetches the terms t_α and t_β corresponding with x and y in E_α and E_β. It then tests via function Possible if the first term is possibly more general than the second. If this is the case the recursion is continued by calling procedure MatchTerms.

```
proc MatchPair(x, y, Max, Min, ToTest, Tested, Result);
  if (x, y) ∈ ToTest
  then
    "remove (x, y) from ToTest;
    "look up x = t_α in E_α";
    T_α := Smd(t_α);
    "look up y = t_β in E_β";
    T_β := Smd(t_β);
    if Possible(T_α,T_β)
    then
      "put (x, y) into Tested";
      MatchTerms(T_α, T_β, Min, Max, ToTest, Tested, Result)
    fi
  fi
end.
```

Procedure MatchTerms computes all alternatives for equating prefix terms of T_α to prefix terms of T_β by calling function MakePairSets. Each alternative uses the original versions of parameters Min, Max, ToTest, and Tested. In each alternative new values of Max and Min are computed. If the minima are still smaller than the maxima (tested by procedure TestMinMax) the recursion is continued by calling procedure MatchRest.

```
proc MatchTerms(T_α, T_β, Max, Min, ToTest, Tested, Result);
  PairSets := MakePairSets(T_α, T_β);
  Cmax := Max;
  Cmin := Min;
  CToTest := ToTest;
  CTested := Tested;
  foreach W ∈ PairSets with not Result do
    Max := Cmax;
    Min := Cmin;
    ToTest := CToTest;
    Tested := CTested;
    ComputeMax(T_α, T_β, Max);
    ComputeMin(T_α, T_β, Min, W);
    if TestMinMax(Min, Max)
    then MatchRest(W, Max, Min, ToTest, Tested, Result)
    fi
  od
end.
```

Procedure MatchRest strips the prefixes from the pairs in W, puts the resulting pairs of variables into ToTest, and continues the recursion by calling MatchPair for each pair in W.

```
proc MatchRest(W, Max, Min, ToTest, Tested, Result);
  foreach (ax, ay) ∈ W with (x, y) ∉ ToTest ∪ Tested do
    "put (x, y) into ToTest"
  od;
  if ToTest = ∅ then Result := true
  else
    foreach (ax, ay) ∈ W do
      MatchPair(x, y, Max, Min, ToTest, Tested, Result)
    od
  fi
end.
```

Function Possible tests if it is possible that T_α is more general than T_β. This is the case if the constants in T_α are a subset of the constants in T_β, and also the prefixes of terms in T_α are a subset of the prefixes of terms in T_β.

```
function Possible(T_α, T_β);
  CT_α := T_α ∩ A;
  PT_α := {a | ∃t : at ∈ T_α};
  PT_β := {a | ∃t : at ∈ T_β};
  if CT_α ⊆ T_β and PT_α ⊆ PT_β
  then Possible := true
  else Possible := false
  fi
end.
```

Function MakePairSets first computes all pairs of a prefix term from T_α and a prefix term from T_β such that the first term could be more general than the second. It then makes all subsets of this set of pairs that contain each prefix term in T_α at least once.

```
function MakePairSets(T_α, T_β);
  Pairs := ∅;
  PT_α := {at | a ∈ A, at ∈ T_α};
  foreach (t_α, t_β) ∈ T_α × T_β do
    if ∃a : t_α ≡ at_α' and t_β ≡ at_β'
    then "put (t_α, t_β) into Pairs"
    fi
  od;
  PairSets := P(Pairs);
  foreach S ∈ PairSets do
    if first(S) ≠ PT_α
    then "remove S from PairSets"
    fi
  od;
  MakePairSets := PairSets
end.
```

Procedure ComputeMax sets for each variable u in T_α the value of Max$[u]$ to T_β if Max$[u]$ was undefined, or $T_\beta \cap T$ if Max$[u]$ already had value T.

```
proc ComputeMax(T_α, T_β, Max);
```

```
  U := T_α ∩ V;
  foreach u ∈ U do
    if Max[u] = "undefd" then Max[u] := T_β
    else Max[u] := Max[u] ∩ T_β
    fi
  od
end.
```

Procedure ComputeMin computes the set of variables in T_α and the set of terms that must minimally be attributed to these variables. This pair of sets is then added to Min. Here it is essential that if y_1 and y_2 are different recursion variables of E_β, then $[\![\langle y_1|E_\beta\rangle]\!] \neq [\![\langle y_2|E_\beta\rangle]\!]$

```
proc ComputeMin(T_α, T_β, Min, W);
  U := T_α ∩ V;
  T := T_β - (second(W) ∪ T_α);
  if (U,T) ≠ (∅,∅) then "put (U,T) into Min" fi
end.
```

Function TestMinMax tests if the minima in Min are smaller than or equal to the union of the maxima in Max for each set of variables in Min.

```
function TestMinMax(Min, Max): boolean;
  test := true;
  foreach (U,T) ∈ Min with test do
    if T ⊈ ⋃_{u∈U} Max[u] then test := false fi
  od;
  TestMinMax := test
end.
```

5.2 Correctness of MoreGeneral

Suppose S_1 and S_2 are solved forms produced by UnPref on input P with $Var(P) = V$. Suppose E_α and E_β are the recursive specifications corresponding to S_1 and S_2, α and β are the corresponding semantic substitutions, σ_α and σ_β are the syntactic substitutions such that $E_\alpha{}' = E_\alpha\sigma_\alpha$ and $E_\beta{}' = E_\beta\sigma_\beta$, $Var(E_\alpha{}') \cap Var(E_\beta{}') = \emptyset$, $E_\alpha{}'$ and $E_\beta{}'$ are linear, and if $y_1, y_2 \in Rec\,Var(E_\beta{}')$ and $y_1 \not\equiv y_2$ then $[\![\,\langle y_1|E_\beta{}'\rangle\,]\!] \neq [\![\,\langle y_2|E_\beta{}'\rangle\,]\!]$. Define $\alpha' = \{\, x := \langle x|E_\alpha{}'\rangle \mid x \in Rec\,Var(E_\alpha{}')\,\}$ and $\beta' = \{\, y := \langle y|E_\beta{}'\rangle \mid y \in Rec\,Var(E_\beta{}')\,\}$. Then $[\![v]\!]\alpha = [\![v\sigma_\alpha]\!]\alpha'$, and $[\![v]\!]\beta = [\![v\sigma_\beta]\!]\beta'$ for all $v \in V$.

The execution of algorithm MoreGeneral can be seen as a tree. In each node there is a call to procedure MatchPair. If the parameters x, y are in ToTest and it is possible that $[\![x]\!]\alpha' \leq [\![y]\!]\beta'$, then MatchPair calls MatchTerms. The for-loop in MatchTerms corresponds to a branching in the execution tree. Each set W of pairs of prefix terms stands for a different alternative in which different pairs of prefix terms are matched. If Possible = true then the for-loop has at least one branch (see function MakePairSets).

The following proofs will look at one path in the execution tree. The first part of each path is a sequence of nodes each containing a successful call to MatchPair, where $(x, y) \in$ ToTest, Possible = true, and TestMinMax = true. The last node contains a call to MatchPair in which one of these conditions does not hold, or Result is made true. The calls to MatchPair along one path will be numbered, and the parameters of call i will be denoted as x_i, y_i, Max_i, Min_i,

ToTest_i, and Tested_i. Internal parameters used inside this call to MatchPair are denoted by T_{α_i}, T_{β_i}, and W_i (MatchTerms), U_i (ComputeMax and ComputeMin), and T_i (ComputeMin).

Consider a path ending in Result := true. Call the final values of Max, Min, and Tested: Max_f, Min_f, and Tested_f. Then each $(x, y) \in \text{Tested}_f$ corresponds to one call to MatchPair with x and y as parameters. Conversely, each (x, y) that is in some ToTest_i is also in Tested_f. So also each (x, y) that is in some W_i is also in Tested_f.

Theorem 5.1 $\alpha \leq_V \beta \Longrightarrow \mathit{MoreGeneral}((\sigma_\alpha, {E_\alpha}'), (\sigma_\beta, {E_\beta}')) = \mathit{true}.$

Proof:
Suppose $\alpha \leq_V \beta$. Then there is a γ such that $[\![v]\!]\alpha\gamma = [\![v]\!]\beta$ for all $v \in V$ and $\text{Dom}(\gamma) = \mathit{FreeVar}({E_\alpha}')$. Then also $[\![v\sigma_\alpha]\!]\alpha'\gamma = [\![v\sigma_\beta]\!]\beta'$, and there is a γ' such that for each $x \in \mathit{RecVar}({E_\alpha}')$ there is a $y \in \mathit{Var}({E_\beta}')$ with $[\![x]\!]\alpha'\gamma' = [\![y]\!]\beta'$.
Now it will be proved that for each $n > 0$ there is a complete execution path of MoreGeneral with $\leq n$ calls to MatchPair and Result = true, or an incomplete execution path (truncated after the nth call to MatchPair) such that at every call i of procedure MatchPair in this path:

1. $[\![x_i]\!]\alpha'\gamma' = [\![y_i]\!]\beta'$,
2. if $\text{Max}_i[u] \neq$ `"undefd"`, then $[\![u]\!]\gamma' \subseteq [\![\text{Max}_i[u]]\!]\beta'$ for all $u \in \mathit{FreeVar}({E_\alpha}')$,
3. if $(U, T) \in \text{Min}_i$, then $[\![T]\!]\beta' \subseteq \bigcup_{u \in U} [\![u]\!]\gamma'$,
4. $\text{ToTest}_i = \emptyset$ or Result = true.

Proof: by induction on the number n of calls to MatchPair.
$n = 1$:
If $E_\alpha = \emptyset$ then MoreGeneral produces Result = true, so there is a complete execution path without calls to MatchPair. Else there is an incomplete execution path ending in a call to MatchPair with:
1) $x_1 \equiv v\sigma_\alpha$ and $y_1 \equiv v\sigma_\beta$ for some $v \in V$, so $[\![x_1]\!]\alpha'\gamma' = [\![y_1]\!]\beta'$,
2) $\text{Max}_1[u] =$ `"undefd"` for all $u \in \mathit{FreeVar}({E_\alpha}')$,
3) $\text{Min}_1 = \emptyset$.
$n > 1$:
If there is a complete execution path with length $\leq n$ for which the conditions hold, then there is a complete path with length $\leq n + 1$. So suppose there is no such path. Then (IH) there is an incomplete execution path with length n having the required properties.
If $\text{ToTest}_n = \emptyset$, then the values of Min_n and Max_n are not changed. If no further call to MatchPair follows then Result = true holds. If a $(n+1)th$ call to MatchPair is made, then $(x_{n+1}, y_{n+1}) \notin \text{ToTest}_{n+1}$, so $(x_{n+1}, y_{n+1}) \in \text{Tested}_{n+1}$, and there is an i such that $x_i \equiv x_{n+1}$ and $y_i \equiv y_{n+1}$, and so (IH) $[\![x_{n+1}]\!]\alpha'\gamma' = [\![y_{n+1}]\!]\beta'$.
If $\text{ToTest}_n \neq \emptyset$, but $(x_n, y_n) \notin \text{ToTest}_n$, then the same argument holds as for the case where $\text{ToTest}_n = \emptyset$, except that there must be a $(n+1)th$ call to MatchPair.
If $(x_n, y_n) \in \text{ToTest}_n$, then $[\![T_{\alpha_n}]\!]\alpha'\gamma' = [\![T_{\beta_n}]\!]\beta'$. Then there is a set W in PairSets_n with the following properties:

1. $[\![ax']\!]\alpha'\gamma' = [\![ay']\!]\beta'$ for all $(ax', ay') \in W$,
2. $[\![U_n]\!]\gamma' \supseteq [\![T_{\beta_n} - (\mathit{second}(W) \cup T_{\alpha_n})]\!]\beta'$.

The branch of the execution path using this W is chosen. Now $\text{Max}_{n+1}[u]$ is computed for all $u \in U_n$. If $\text{Max}_n[u] =$ `"undefd"`, then $\text{Max}_{n+1}[u] = T_\beta$, and $[\![u]\!]\gamma' \subseteq [\![T_{\alpha_n}]\!]\alpha'\gamma' = [\![T_{\beta_n}]\!]\beta'$. If

$\mathrm{Max}_n[u] \neq$ `"undefd"` then $\mathrm{Max}_{n+1}[u] = \mathrm{Max}_n[u] \cap T_{\beta_n}$. And (IH) $[\![u]\!]\gamma' \subseteq [\![\mathrm{Max}_n[u]]\!]\beta'$. Also $[\![u]\!]\gamma' \subseteq [\![T_{\alpha_n}]\!]\alpha'\gamma' = [\![T_{\beta_n}]\!]\beta'$. So $[\![u]\!]\gamma' \subseteq [\![\mathrm{Max}_n[u]]\!]\beta' \cap [\![T_{\beta_n}]\!]\beta'$. And $[\![\mathrm{Max}_n[u]]\!]\beta' \cap [\![T_{\beta_n}]\!]\beta' = [\![\mathrm{Max}_n[u] \cap T_{\beta_n}]\!]\beta'$ because Max_n[u] and T_{β_n} only contain terms of the form $a \in \mathcal{A}$, $w \in \mathit{FreeVar}(E_\beta{}')$, and bx with $x \in \mathit{RecVar}(E_\beta{}')$, $\mathrm{Dom}(\beta') = \mathit{RecVar}(E_\beta{}')$, and if $[\![x]\!]\beta' = [\![y]\!]\beta'$ then $x \equiv y$.
Also $\mathrm{Min}_{n+1} = \mathrm{Min}_n \cup \{U_n, T_n\}$, where $T_n = T_{\beta_n} - (\mathit{second}(W_n) - T_{\alpha_n})$. So $[\![T_n]\!]\beta' \subseteq [\![U_n]\!]\gamma'$.
If this W does not lead to a $(n+1)$th call of MatchPair, then this call to MatchPair produces Result := true, and $\mathrm{ToTest}_{n+1} = \emptyset$.
If the $(n+1)$th call to MatchPair is from procedure MatchRest with parameters x' and y', then $(ax', ay') \in W_n$ for some a and $[\![ax']\!]\alpha'\gamma' = [\![ay']\!]\beta'$, so $[\![x']\!]\alpha'\gamma' = [\![y']\!]\beta'$. If this call is from procedure MoreGeneral with parameters x' and y' then there is a $v \in V$ such that $v\sigma_\alpha = x'$ and $v\sigma_\beta = y'$. And $[\![v\sigma_\alpha]\!]\alpha'\gamma' = [\![v\sigma_\beta]\!]\beta'$ (assumption). So conditions 1), 2), 3), and 4) also hold at the $(n+1)$th call to MatchPair.
And all execution paths contain a finite number of calls to MatchPair, so there is a finite complete execution path that produces Result = true. □

Theorem 5.2 *Suppose S_1, S_2, E_α, E_β, α, β, $E_\alpha{}'$, $E_\beta{}'$, α', and β' are as before.*
If MoreGeneral$((\sigma_\alpha, E_\alpha{}'), (\sigma_\beta, E_\beta{}')) =$ true, then $\alpha \leq_V \beta$.

Proof:
Consider the path in the execution of MoreGeneral that produces Result = true. Number the calls to MatchPair in this path. Call the final values of Max, Min, and Tested of this path Max_f, Min_f, and Tested_f, respectively. Suppose a semantic substitution γ is given. Now it is proved that if for all $(T, U) \in \mathrm{Min}_f$: $[\![T]\!]\beta' \subseteq [\![U]\!]\gamma \subseteq \bigcup_{u \in U} [\![\mathrm{Max}_f[u]]\!]\beta'$, then for all $n \in \mathbb{N}$ and all $(x, y) \in \mathrm{Tested}_f$: $\pi_n([\![x]\!]\alpha'\gamma) = \pi_n([\![y]\!]\beta')$. This is proved by induction on n.
If $\mathrm{Tested}_f = \emptyset$ then $\alpha = \mathrm{Id}$ (the identity function on processes), and $\alpha \leq_V \beta$. So suppose $\mathrm{Tested}_f \neq \emptyset$. Take a pair (x, y) from Tested_f, and suppose the call to MatchPair having x and y as parameters is number i. Then:
$n = 1$:
$\pi_1([\![\mathit{first}(W_i)]\!]\alpha'\gamma) = \pi_1([\![\mathit{first}(W_i)]\!]) = \pi_1([\![\mathit{second}(W_i)]\!]) = \pi_1([\![\mathit{second}(W_i)]\!]\beta')$ (see function MakePairSets). And:

$$\begin{aligned}
&\pi_1([\![x]\!]\alpha'\gamma) = \pi_1([\![T_{\alpha_i}]\!]\alpha'\gamma) \\
&= \pi_1([\![U_i]\!]\gamma) \cup \pi_1([\![T_{\alpha_i} \cap T_{\beta_i}]\!]) \cup \pi_1([\![\mathit{first}(W_i)]\!]\alpha'\gamma) \\
&\supseteq \pi_1([\![T_i]\!]\beta') \cup \pi_1([\![T_{\alpha_i} \cap T_{\beta_i}]\!]) \cup \pi_1([\![\mathit{second}(W_i)]\!]\beta') \\
&= \pi_1([\![T_{\beta_i} - (\mathit{second}(W_i) \cup T_{\alpha_i})]\!]\beta') \cup \pi_1([\![T_{\alpha_i} \cap T_{\beta_i}]\!]) \cup \pi_1([\![\mathit{second}(W_i)]\!]\beta') \\
&\supseteq \pi_1([\![T_{\beta_i}]\!]\beta') - (\pi_1([\![\mathit{second}(W_i)]\!]\beta') \cup \pi_1([\![T_{\alpha_i}]\!])) \cup \pi_1([\![T_{\alpha_i} \cap T_{\beta_i}]\!]) \cup \pi_1([\![\mathit{second}(W_i)]\!]\beta') \\
&\supseteq \pi_1([\![T_{\beta_i}]\!]\beta') = \pi_1([\![y]\!]\beta').
\end{aligned}$$

Also:

$$\begin{aligned}
&\pi_1([\![x]\!]\alpha'\gamma) = \\
&= \pi_1([\![U_i]\!]\gamma) \cup \pi_1([\![T_{\alpha_i} \cap T_{\beta_i}]\!]) \cup \pi_1([\![\mathit{first}(W_i)]\!]\alpha'\gamma) \\
&\subseteq \pi_1(\textstyle\bigcup_{u \in U_i} [\![\mathrm{Max}_f[u]]\!]\beta') \cup \pi_1([\![T_{\alpha_i} \cap T_{\beta_i}]\!]) \cup \pi_1([\![\mathit{second}(W_i)]\!]\beta') \\
&\subseteq \pi_1([\![T_{\beta_i}]\!]\beta') = \pi_1([\![y]\!]\beta').
\end{aligned}$$

$n > 1$:

$$\begin{aligned}
&\pi_n([\![\mathit{first}(W_i)]\!]\alpha'\gamma) = \{\pi_n([\![ax']\!]\alpha'\gamma) \mid (ax', ay') \in W_i\} = \\
&= \{[\![a]\!]\pi_{n-1}([\![x']\!]\alpha'\gamma) \mid (ax', ay') \in W_i\} = (IH) \\
&= \{[\![a]\!]\pi_{n-1}([\![y']\!]\beta') \mid (ax', ay') \in W_i\} \\
&= \pi_n([\![\mathit{second}(W_i)]\!]\beta'),
\end{aligned}$$

Also $\pi_n([\![x]\!]\alpha'\gamma) = \pi_n([\![y]\!]\beta'$ (proof analogous to proof of $\pi_1([\![x]\!]\alpha'\gamma) = \pi_1([\![y]\!]\beta')$).
So if $[\![T]\!] \subseteq [\![U]\!]\gamma \subseteq \bigcup_{u\in U}[\![\mathrm{Max}_f[u]]\!]\beta'$ for all $(T < U) \in \mathrm{Min}_f$, then for all (x, y) in Tested_f and for all $n \in \mathbb{N}$: $\pi_n([\![x]\!]\alpha'\gamma) = \pi_n([\![y]\!]\beta')$. Then $[\![x]\!]\alpha'\gamma = [\![y]\!]\beta'$, because all processes in $\mathcal{P}$ have a finite number of states. And for all $v \in V$: $(v\sigma_\alpha, v\sigma_\beta)$ is in Tested_f (see procedure MoreGeneral). Take a semantic substitution γ with $\mathrm{Dom}(\gamma) = \mathit{FreeVar}(E_\alpha{}')$ and $[\![T]\!]\beta' \subseteq [\![U]\!]\gamma \subseteq \bigcup_{u\in U}[\![\mathrm{Max}_f[u]]\!]\beta'$ for all $(T, U) \in \mathrm{Min}_f$. This is possible because $T \subseteq \bigcup_{u\in U}\mathrm{Max}_f[u]$ for all $(T, U) \in \mathrm{Min}_f$. So also $[\![T]\!]\beta' \subseteq \bigcup_{u\in U}[\![\mathrm{Max}_f[u]]\!]\beta'$. Then for all $v \in V$: $[\![v\sigma_\alpha]\!]\alpha'\gamma = [\![v\sigma_\beta]\!]\beta'$. It follows that for all $v \in V : [\![v]\!]\alpha\gamma = [\![v]\!]\beta$. So $\alpha \leq_V \beta$. □

Chapter 6

Complexity of UnPref and MoreGeneral

6.1 Complexity of UnPref

An execution of algorithm UnPref has the form of a tree, where each node corresponds to the application of a rule, and a branching node corresponds to an application of rule Sum, producing a number of alternatives. If the maximal path length is p and the branching degree is b, then the complexity of UnPref is $O(b^p)$.

Suppose P is a set of equations that is input to UnPref. Define a number of measures on P:

N_s : the number of different prefix subterms in P,
N_{sv} : the number of different subterms in P having a variable as a summand,
N_{eq} : the maximal number of prefix subterms in P starting with the same action prefix.

N_s, N_{sv}, and N_{eq} are dependent on the size of P, measured as the number of equations and the length of the terms in P. The following measures reflect properties of sum subterms in P, and do not necessarily increase with the size of P.

n_v : the maximum number of variable summands in one subterm of P,
n_{nv} : the maximum number of non-variable summands in one subterm of P,
n_{eq} : the maximum number of prefix summands in one sum subterm of P starting with the same action prefix,
n_{da} : the maximum number of different action prefixes with which prefix terms start in one sum subterm of P.

On each original equation at least once a rule is applied. Only rules 4 (Prefix) and 5 (Sum) produce new equations. Sum produces zero or more equations for Prefix, and Prefix produces an equation for an arbitrary rule. If rule Prefix is used in more nodes in one path, it is in each node applied to a different equation. These equations are of the form $at_1 =^?_{\mathcal{E}} at_2$ with at_1 and at_2 subterms of P. So the number of applications of Prefix in one path is at most $N_s N_{eq}$. Each application produces one new equation, on which any rule may be applicable. If rule Eq can be applied then the equation will be discarded. Rule Var moves the equation to the special set S, and NoSol terminates the path. If Sum is applicable, then a number of equations of the form $at_1 =^?_{\mathcal{E}} at_2$ is added to P. So for each application of Prefix there is at most one application of another rule. It follows that the path length has complexity

$$O(N_s \cdot N_{eq}).$$

The number of branches in a node with an application of Sum is the product of the number of sets produced by MakePairSets and the number of alternatives produced by SetsAlg. These are dependent on the number of summands that are a variable and the number of summands that are not a variable in the equation under consideration.

Every application of Sum may create a new set of fresh variables. The function *rep* may substitute these new variables into the equations. In the first call to Sum at most ${n_v}^2$ new variables are created (see procedure MakeNewVars), in the second at most $({n_v}^2)^2$, and in the nth at most ${n_v}^{2^n}$. The number of calls to Sum producing new variables in one path is at most ${N_{sv}}^2$, so a term in an equation produced during the execution of UnPref contains at most

$${n_v}^{2^{N_{sv}^2}}$$

variables.

If a term in an equation has summands that are variables, then the number of summands may increase by applying the function *rep*. So the maximum number of prefix summands in a term of an intermediate equation is at most N_s. Procedure MakePairSets first determines a set of matching pairs. The maximal size of this set is $N_s \cdot N_{eq}$, so the complexity of the power set of this set is

$$O(2^{N_s \cdot N_{eq}}).$$

Suppose procedure SetsAlg has input $s =^?_{\mathcal{E}} t$ and W. Define $k_X = \#X$, $k_Y = \#Y$, $k_Z = \#Z$, $l_D = \#T_{D_L}$, $l_L = \#T_L$, and $l_R = \#T_R$, with X, Y, Z, T_{D_L}, T_L, and T_R as in procedure DistrConstants. Then SetsAlg produces

$$(2^{k_Y} - 1 + k_Z)^{l_L} \cdot (2^{k_X} - 1 + k_Z)^{l_R} \cdot (2^{k_X} + 2^{k_Y} - 1)^{l_D}$$

alternatives [BB86], [Dro92].
And $k_X + k_Y + k_Z \leq {n_v}^{2^{N_{sv}^2}}$ and $l_L + l_R + l_D \leq N_s$,
so SetsAlg produces at most

$$O((2^{n_v^{2^{N_{sv}^2}}})^{l_L+l_R+l_D} \leq O(2^{N_s \cdot n_v^{2^{N_{sv}^2}}})$$

alternatives. This gives a worst case branching complexity of

$$O(2^{N_s \cdot N_{eq}} \cdot 2^{N_s \cdot n_v^{2^{N_{sv}^2}}}) = O(2^{N_s^2 \cdot N_{eq} \cdot n_v^{2^{N_{sv}^2}}})$$

and a worst case complexity for UnPref of :

$$O(2^{N_s^2 \cdot N_{eq}^2 \cdot n_v^{2^{N_{sv}^2}}})$$

This extremely high complexity is caused by the extremely high number of possible solutions for certain sets of equations, especially when sums contain variables as summands. If a sum subterm contains more than one variable summand, this represents a choice from a set of processes with more than one completely unknown process, and this is a not very useful application in top-down system development. A process with unspecified components that must conform to a given specification corresponds to an equation $s =^?_{\mathcal{E}} t$ without variables in t. This suggests some reasonable restrictions on the allowed input:

1. At most one summand of each sum subterm of P is a variable.
   Then the number of variables in an equation does not increase during execution. If SetsAlg is applied to an equation $s =^?_{\mathcal{E}} t$ with $x \in Smd(s)$, $y \in Smd(t)$, and $x \equiv y$, then it produces one branch, if $x \not\equiv y$ then it produces $3^{l_D} \leq 3^{N_s}$ alternatives. So the complexity of UnPref in this case is:

   $$O((2^{N_s \cdot N_{eq}} \cdot 3^{N_s})^{N_s \cdot N_{eq}}) = O(2^{N_{eq}} \cdot 6^{N_s^2 \cdot N_{eq}}).$$

2. In each equation of P one of both terms contains no variables.
   In this case SetsAlg produces at most $(2^{n_v} - 1)^{n_{nv}}$ alternatives. Function *rep* may substitute terms for variables, but each of these terms contains at most n_{nv} summands, none of which is a variable. So procedure MakePairSets produces at most $2^{n_{nv}^2 \cdot n_v}$ alternatives, and the complexity of UnPref becomes:

   $$O((2^{n_{nv}^2 \cdot n_v} \cdot 2^{n_{nv} \cdot n_v})^{N_s \cdot N_{eq}}) = O(2^{n_v \cdot (n_{nv}^2 + n_{nv}) \cdot N_s \cdot N_{eq}}).$$

3. No summands of sum subterms of P are variables.
   Then SetsAlg produces only one alternative, the number of subterms in sums does not change by substitutions, and MakePairSets produces at most $2^{n_{da} \cdot n_{eq}^2}$ sets. So the complexity of UnPref becomes:

   $$O(2^{n_{da} \cdot n_{eq}^2 \cdot N_s \cdot N_{eq}}).$$

The complexity decreases further if the amount of nondeterminism (number of terms in a sum that start with the same prefix) is reduced.

6.2 Complexity of MoreGeneral

Suppose $(\sigma_\alpha, E_\alpha)$ and (σ_β, E_β) are input to function MoreGeneral. Define $N_\alpha = \#\, Var(E_\alpha), N_\beta = \#\, Var(E_\beta)$. Then procedure MatchPair is executed at most $N_\alpha \cdot N_\beta$ times, so the worst case complexity of MoreGeneral is $O(N_\alpha \cdot N_\beta)$. And to remove all non-most-general solutions from a set of N solutions, MoreGeneral is executed at most $N \cdot (N-1)$ times.

Chapter 7

Conclusions and discussion

In this paper a rule-based unification algorithm is given for a restricted process algebra with as operators nondeterministic choice and action prefixes. The algebra contains infinite processes, that are denoted by recursive specifications. The algorithm is proved sound, complete and terminating. The produced solution set is not always minimal, so a matching algorithm is given to remove non-minimal solutions. This algorithm is also proved correct.

The worst case complexity of UnPref is exponential in the number of subterms of the input equation set, and quadruply exponential in the number of subterms of P having a variable as a summand. This extremely high complexity is reduced to singly exponential if restrictions are placed on the input equations. These restrictions are: at most one summand of each subterm is a variable, or at least one side of each equation contains no variables, or no summands of sum subterms of the equations are variables.

Bibliography

[BW90] Baeten, J.C.M., and W.P. Weijland: Process Algebra. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, Cambridge. 1990.

[BMOZ88] Bakker, J.W. de, J.-J. Meyer, E.-R. Olderog, and J.I. Zucker: Transition Semantics, Metric Spaces and Ready Sets in the Semantics of Uniform Concurrency. J. of Comp. and Syst. Sci. 36, pp. 158-224. 1988.

[BK84] Bergstra, J.A., and J.W. Klop: Process Algebra for Synchronous Communication. Inf. and Control 60, pp. 109-137. 1984.

[Col84a] Colmerauer, A.: Equations et inequations sur les arbres finis et infinis. Internal Report, Groupe Intelligence Artificielle, Université Aix-Marseille II. 1984.

[Col84b] Colmerauer, A.: Equations and inequations on finite and infinite trees. Proc. of the Int. Conf. on Fifth Gen. Comp. Syst. 1984.

[DJ91] Dershowitz, N., and J.-P. Jouannaud: Notations for rewriting. Bull. of the Eur. Ass. for Theor. Comp. Sci. 43, pp. 162-173. 1991.

[Dro92] Drost, N.J.: Unification in the algebra of sets with union and empty set. Report P9213, Programming Research Group, University of Amsterdam. 1992.

[GV90] Groote, J.F., and F. Vaandrager: An efficient algorithm for branching bisimulation and stuttering equivalence. Proc. ICALP, Warwick, LNCS 443, pp. 626-638. 1990.

[Hoa85] Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, London. 1985.

[Hue76] Huet, G.: Resolution d'equations dans les langages d'ordre 1,2,...,Ω. Thèse de Doctorat d'État, Université Paris VI. 1976.

[Jaf84] Jaffar, J.: Efficient Unification over Infinite Terms. New Generation Computing 2, pp. 207-219. 1982.

[JK91] Jouannaud, J.-P., and C. Kirchner: Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In: Lassez, J.-L. and G. Plotkin (eds): Computational Logic. Essays in honor of Alan Robinson. pp. 257-321. MIT Press, Cambridge, MA. 1991.

[MM82] Martelli, A. and U. Montanari: An Efficient Unification Algorithm. ACM Trans. on Prog. Lang. and Syst. 4(2), pp. 258-282. 1982.

[Mil80] Milner, R.: A Calculus of Communicating Systems. LNCS 22, Springer Verlag, Berlin. 1980.

[Mil89] Milner, R.: Communication and Concurrency. Prentice Hall, London. 1989.

[Plo72] Plotkin, G.: Building-in equational theories. In: Meltzer, B., and D. Michie (eds): Machine Intelligence 7, pp. 73-90. University Press Edinburgh. 1972.

[Rob65] Robinson, J.A.: A Machine Oriented Logic Based on the Resolution Principle. JACM 12, pp. 23-41. 1965.

[Sch89] Schmidt-Schauss, M.: Unification in a Combination of Arbitrary Disjoint Equational Theories. J. Symbolic Computation 8, pp. 51-99. 1989.