TERM REWRITING IN ASSOCIATIVE COMMUTATIVE THEORIES WITH IDENTITIES

A THESIS PRESENTED

BY

Martin Joachim Henz

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE IN

COMPUTER SCIENCE

STATE UNIVERSITY OF NEW YORK

AT STONY BROOK

December 1991

State University of New York

at Stony Brook

The Graduate School

Martin Joachim Henz

We, the thesis committee for the above candidate for

the degree of Master of Science,

hereby recommend acceptance of this thesis.

Professor Leo Bachmair, Advisor
Computer Science

Professor Jieh Hsiang,
Computer Science

Professor I.V. Ramakrishnan,
Computer Science

This thesis is accepted by the Graduate School.

Graduate School

Abstract of the Thesis

Term Rewriting in Associative Commutative Theories with Identities

by

Martin Joachim Henz

Master of Science

in

Computer Science

State University of New York

at Stony Brook

1991

Versions of constraint rewriting for completion of rewrite systems in the presence of associative commutative operators with identities have been proposed, in which constraints are used to limit the applicability of rewrite rules. We extend these approaches such that the initially given equations can contain constraints, and such that a suitable version of unification modulo associativity, commutativity and identity can be interleaved with the process of completion.

To

my parents Albert and Klara Henz

and

my wife Kelly Reedy.

Contents

List of Tables

List of Figures

Acknowledgements

I take this opportunity to express my gratitude to my thesis advisor, Professor Leo Bachmair, for his continuous support and invaluable guidance he offered me in many fruitful discussions.

I thank the members of my thesis committee, Professor Jieh Hsiang and Professor I.V. Ramakrishnan, for having read and discussed this thesis.

I wish to extend my thanks to Professor Jacques Loeckx and Stefan Uhrig who supported and encouraged me thoughout my studies in Saarbrücken.

I am indebted to my friends in Stony Brook, especially my office mates, for their hospitality and help, and for having turned my stay in Stony Brook into the most enjoyable experience.

I thank the Fulbright Commission for providing a scholarship that made my stay in Stony Brook possible and the Siemens AG for financial support throughout my studies.

Finally, I wish to give my most special thanks to my wife, Kelly Reedy, for "being with me in all my struggles". I also thank her for proof-reading this thesis.

Chapter 1

Introduction

In mathematics and the sciences, terms and equations are extensively used. Computers provide tools to mechanize the handling of terms and equations. In many computer applications, including symbolic algebraic computation, automated theorem proving, program specification and verification, automated reasoning about terms and equations plays an important role.

Rewrite systems are sets of directed equations used to compute by replacing subterms in a given term until a simplest form possible, called a normal form, is obtained. A rewrite system is called canonical, if normal forms are unique and always exist. Knuth and Bendix [KB 70] proposed a procedure, called standard completion procedure, to build canonical systems by systematically generating and orienting valid equational consequences from a given set of equations.

An extension of the framework of rewrite systems is rewriting in equational theories where the replacement of subterms is generalized using equations that express properties of the used operators. Extended completion leads to the construction of canonical systems in this framework.

In many applications, operators with the properties of associativity and commutativity occur (examples are conjunction in logic and addition of integers). A procedure for completion in the presence of associative commutative operators has been presented in [PS 81]. Here, equational consequences from equations are computed using *associative-commutative unification*, a process that is in many cases particularly time-consuming.

Often, associative commutative operators are associated with an identity (for example `true` for conjunction and 0 for addition). Unfortunately, associative commutative completion often fails in the presence of identities, since severe restrictions on the orientation of equations must be imposed.

In [BPW 89], this problem is overcome by limiting the applicability of directed equations via *constraints*. This approach is presented in [JM 90] in the framework of transformation rules for completion. A general approach to rewriting using constraints is presented in [KKR 91].

In [BPW 89] and [JM 90], only the resulting rewrite systems, but not the initial set of equations may contain constraints, which limits the application of this approach in practice.

We shall make use of constraints in completion in the presence of associative commutative operations with identities for two purposes. Firstly, we shall extend the appoach of [BPW 89] and [JM 90] to undirected equations, such that the given sets of equations may contain constraints. Secondly, we shall provide a framework to interleave associative commutative unification with the process of completion in order to gain the flexibility of postponing particularly hard unification problems, hoping that they can be simplified by later derived directed equations.

In Chapter 2, we present fundamental concepts, including our notion of constraints. The subject of Chapter 3 is the solving of constraints, which can be viewed as an extension of unification in the presence of associative commutative operators and identities. In Chapter 4, we present a transformation system for completion, incorporating the constraint solving methods of the previous chapter. A computer implementation of various components of completion procedures is described in Chapter 5.

Chapter 2

Preliminaries

2.1 Terms

Let S be a set. When X is a sequence of elements of S of length $|X| = n$, we denote its components by $X^1, X^2, \ldots, X^n$. The empty sequence is denoted by λ. An element s of S *occurs* in X, denoted $s \in X$, if $s = X^i$ for some $i \in [1..n]$. ($[1..n]$ denotes the set $\{i \in \mathbf{N} \mid 1 \leq i \leq n\}$.) A sequence Y of length m is called a *subsequence* of a sequence X of length n, denoted $Y \subseteq X$, if there exists a function f from $[1..m]$ to $[1..n]$, such that $i > j$ implies $f(i) > f(j)$ for all $i, j \in [1..m]$ and $Y^i = X^{f(i)}$ for all $i \in [1..m]$. The *number of occurrences* of s in X, denoted $|X|_s$, is the number $|\{i \mid X^i = s\}|$. The *difference* of two sequences X and Y, denoted $X - Y$, is the subsequence Z of X such that for all $s \in S$ the number $|Z|_s$ is $|X|_s - |Y|_s$ if $|X|_s - |Y|_s \geq 0$ and 0 otherwise. A sequence Y is a *permutation* of a sequence X, if for all $s \in S$ holds $|Y|_s = |X|_s$.

Let F be a set of *function symbols* with $F = F_{free} \uplus F_{AC} \uplus F_{zero}$. We call the elements of F_{free} *free function symbols*, the elements of F_{AC} *associative commutative function symbols*, short *AC-symbols*, and the elements of F_{zero} *identities.* We assume that there exists a partial function *zero* from F_{AC} to F_{zero}. The subset of F_{AC} that contains all associative commutative function symbols f for which $zero(f)$ is defined is denoted by F_{AC1}; its elements are called *associative commutative function symbols with identity*, short *AC1-symbols.*

With each function symbol f, we associate a non-empty set $A(f) \subseteq \mathbf{N}$ that indicates the number of arguments f may take. If $f \in F_{zero}$, then $A(f) = \{0\}$; if $f \in F_{AC}$, then $A(f) = \mathbf{N} - \{0, 1\}$. We assume that $A(f)$ contains only one element if $f \in F_{free}$.

Let V be a countable set disjoint from F, whose elements we call *variables.* A *term* t is either a variable or an expression $f(X)$, where X is a sequence of terms and $|X| \in A(f)$. In the latter case, f is called the *root* of t, denoted $root(t)$, and X is called *sequence of arguments* of t, denoted $args(t)$. When $root(t) \in F_{AC}$, t is called *AC-term*; when $root(t) \in F_{AC1}$, t is called *AC1-term.* The set of all terms built from function symbols in F and variables in V is denoted by $T(F, V)$.

To enhance readability, we often use infix notation for symbols in F_{AC}, omit the empty sequence λ and the parentheses around sequences of arguments of length 0 and 1.

Example 2.1 For this and all following examples, let

$$F = F_{free} \uplus F_{AC} \uplus F_{zero} \text{ where}$$
$$F_{free} = \{a, b, c, d, e, -, f, g\},\ F_{AC} = \{\#, +, *\},\ F_{zero} = \{0, 1\},$$
$$zero(+) = 0,\ zero(*) = 1 \quad (+ \text{ and } * \text{ are } AC1\text{-symbols}),$$
$$A(a) = A(b) = A(c) = A(d) = A(e) = 0,\ A(-) = 1,$$
$$A(f) = 1, A(g) = 2, \text{ and } v, w, x, y, z \in V.$$

The expressions r, s, t with

$$\begin{aligned} r &= g(1(\lambda), (f(g(0(\lambda), 1(\lambda))))), \\ s &= +(x, y, *(a(\lambda), z)), \\ t &= \#(1(\lambda), f(0(\lambda)), g(+(f(x), *(-(v), 0(\lambda))), w)) \end{aligned}$$

are terms in $T(F, V)$. The terms s, t are AC-terms. The term s is additionally an $AC1$-term. In infix notation, the terms are written:

$$\begin{aligned} r &= g(1, f(g(0, 1))), \\ s &= x + y + (a * z), \\ t &= 1\#f(0)\#g(f(x) + ((-v) * 0), w). \end{aligned}$$

The function Var maps each term to the set of variables that occur in it: $Var(x) = x$ if x is a variable and $Var(f(X)) = \bigcup_{i\in|X|} Var(X^i)$.

A term s is said to be a *subterm* of a term $t = f(X)$ if either $s = f(Y)$ for some subsequence Y of X, or else s is a subterm of some term $X^i \in X$. When we want to emphasize that a term t contains a term s as a subterm, we denote t by $t[s]$. A term s is a *proper subterm* of a term t if s is a subterm of t and $s \neq t$. The set of all subterms of a term t is denoted by $subt(t)$. A subterm s

of t is called a *non-variable subterm* of t, if $s \notin V$. The set of all non-variable subterms of a term t is denoted by $Fsubt(t)$.

We use a pair $\pi = \langle p, P \rangle$, where p is a sequence of of positive numbers and P is a set of positive numbers, called a *position*, to refer in a term t to a specific subterm in t, denoted by t_π. The pair $\langle \lambda, \emptyset \rangle$ is a position in any term and referring to the term itself; the pair $\langle (i, p), \emptyset \rangle$ is a position in a term $t = f(X)$ and referring to the subterm s of t, if $\langle p, \emptyset \rangle$ is a position in X^i referring to s; and $\langle p, P \rangle$ is a position in a term t and referring to the subterm $s = g(Y)$ of t, if $\langle p, \emptyset \rangle$ is a position in t referring to a term $g(X)$, $g(Y)$ is a subterm of $g(X)$, $2 \leq |P| < |X|$ and $i \in P$ whenever $X^i \in Y$. For instance, the term $x + z$ is the subterm of the term $a * g(x + y + z, 1))$ at position $\langle (2, 1), \{1, 3\} \rangle$. Note, that this notion of subterms and positions is an extension of the notion of "flattened positions" in [Mar 91], where as second components of positions only sets of subsequent numbers are allowed.

A position π in t is called a *non-variable position* in t if $t|_\pi \notin V$. The set of all positions in t is denoted by $pos(t)$ and the set of all non-variable positions in t by $Fpos(t)$.

The result of replacing the subterm at position $\langle p, P \rangle$ in a term t by a term s is denoted by $t[s]_{\langle p,P \rangle}$ and defined to be the term s if $p = \lambda$ and $P = \emptyset$; the term $f(X - Y, s)$ if $t = f(X)$, $p = \lambda$, $P \neq \emptyset$, and $f(Y)$ is the subterm of t at position $\langle p, P \rangle$; and the term $f(X, t'[s]_{\langle p',P \rangle}, Y)$ if $t = f(X, t', Y)$, $p = (i, p')$, and $i = |X| + 1$. For instance, $(g(x + y + z))[f(a) + b]_{\langle (1), \{1,3\} \rangle} = g(y + f(a) + b)$.

A substitution σ is a mapping from variables to terms. The value of a substitution σ for a variable x is denoted by $x\sigma$. We call the set of variables for which $x\sigma \neq x$ the *domain* of σ, denoted $Dom(\sigma)$, and consider only substitutions with finite domain. We denote a substitution σ by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, when $Dom(\sigma) = \{x_1, \ldots, x_n\}$ and $x_i\sigma = t_i$ for $i \in [1..n]$. The set $\bigcup_{x \in Dom(\sigma)} Var(x\sigma)$ is called the *range* of σ, denoted $Ran(\sigma)$. If $W \subseteq V$ then $\Sigma(W)$ denotes the set of all substitutions σ for which $Dom(\sigma) \subseteq W$. If σ is a substitution and $W \subseteq V$, then $\sigma|_W$ denotes the restriction of σ on W: $x\sigma|_W = x\sigma$ if $x \in W$ and $x\sigma|_W = x$ otherwise.

A substitution σ can be uniquely extended to a mapping on terms in such a way that $(f(X))\sigma = f(X^1\sigma, \ldots, X^n\sigma)$, for all terms $f(X)$ where $|X| = n$. The composition of two substitutions σ and τ is denoted by the juxtapositon $\sigma\tau$. That is, $t\sigma\tau = (t\sigma)\tau$ for all terms t.

A term s is an *instance* of (or *matches*) another term t if there exists a substitution σ, called a *matching substitution*, such that $s = t\sigma$. Two terms s and t are said to be *unifiable* if there exists a substitution σ, called a *unifier*, such that $s\sigma = t\sigma$.

Note: The notion of terms can easily be extended to many sorts (see Chapter 5), but considering only one sort simplifies the notation and imposes no restrictions on the validity of our results.

2.2 Relations

If $\rightarrow$ is a binary relation on a set S, we denote by $\leftarrow$ its inverse; by $\leftrightarrow$ its symmetric closure $\rightarrow \cup \leftarrow$; by $\rightarrow^+$ its transitive closure; by $\rightarrow^*$ its transitive-reflexive closure; and by $\leftrightarrow^*$ its symmetric-transitive-reflexive closure. The composition of two relations $\rightharpoonup$ and $\rightharpoondown$ is denoted by $\rightharpoonup \circ \rightharpoondown$. The relation $\rightarrow$ is said to be *terminating* if there is no infinite sequence $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots$. A binary relation $\rightarrow$ is said to be *Church-Rosser* if for any two elements r and s with $r \leftrightarrow^* s$, there exists an element v such that $r \rightarrow^* v \leftarrow^* s$. Terminating Church-Rosser relations are called *convergent*.

An equivalence relation is a reflexive, symmetric, transitive binary relation. Let $\rightarrow$ be a relation and $\diamond$ an equivalence. The relation $\rightarrow$ is called *Church-Rosser modulo* $\diamond$ if for all elements r and s with $r\ (\diamond \cup \leftrightarrow)^*\ s$, there exist terms v and w, such that $r \rightarrow^* v\ \diamond\ w \leftarrow^* s$. We say that $\rightarrow$ is *convergent modulo* $\diamond$ if $\diamond \circ \rightarrow \circ \diamond$ is terminating and Church-Rosser modulo $\diamond$. The relation $\rightarrow$ is said to be locally confluent modulo $\diamond$, if for all elements r, r', s, s' with $r' \leftarrow r\ \diamond\ s \rightarrow s'$, there exist elements r'', s'' such that $r' \leftarrow^* r''\ \diamond\ s'' \rightarrow^* s'$.

An element $s \in S$ is said to be in *normal form* with respect to the relation $\rightarrow$ if there exists no element $s' \in S$ such that $s \rightarrow s'$. A function that assigns to every element in S a normal form with respect to R is denoted by $\rightarrow^!$.

An *ordering* is an irreflexive, transitive binary relation and a *quasi-ordering* is a reflexive, transitive binary relation.

If $\succ$ is a strict ordering, then its reflexive closure $\succeq$ is a quasi-ordering. On the other hand, if $\succeq$ is a quasi-ordering, the corresponding equivalence $\sim$ and ordering $\succ$ are defined as follows: $s \sim t$ if $s \succeq t$ and $t \succeq s$; and $s \succ t$ if $s \succeq t$ and $t \not\succeq s$. An ordering is called *well-founded* if it is terminating.

Definition 2.1 If $>_i$ are orderings on S_i for all $i \in [1..n]$, then their lexicographic combination $(>_1, >_2, \ldots, >_n)$ on n-tuples in $S_1 \times S_2 \times \cdots \times S_n$ is defined as follows:

$$(a_1, a_2, \ldots, a_n)\ (>_1, >_2, \ldots, >_n)\ (b_1, b_2, \ldots, b_n)$$

if there exists $i \in [1..n]$ such that for all $j \in [1..i-1]$ holds $a_j \geq_j b_j$ and $a_i >_i b_i$, where $\geq_j$ denotes the reflexive closure of $>_j$.

Proposition 2.1 *If the orderings $>_i$ are well-founded for all $i \in [1..n]$, then their lexicographic combination $(>_1, >_2, \ldots, >_n)$ is well-founded.*

Definition 2.2 If $\succ$ is an ordering on S, then its set extension $\succ_{set}$ is defined as follows: $M \succ_{set} N$, if $M \neq N$ and for all $s \in N - M$ there exists $s' \in M - N$ such that $s' \succ s$.

Proposition 2.2 *If $\succ$ is well-founded, then its set extension $\succ_{set}$ is well-founded on finite sets.*

A *multiset* over a set S is a mapping M from S to the natural numbers. We say that x is an *element* of M, denoted by $x \in_{mul} M$, if $M(x) > 0$. A multiset M is finite, if $\{x \mid x \in_{mul} M\}$ is finite. We often denote a finite multiset M by $\{A\}_{mul}$, where A is a sequence of elements in S and $|A|_s = M(s)$ for all $s \in S$. The *union*, *intersection*, and *difference* of multisets are defined by

$$\begin{aligned}
M_1 \cup M_2\ (x) &= M_1(x) + M_2(x), \\
M_1 \cap M_2\ (x) &= min(M_1(x), M_2(x)), \\
M_1 - M_2\ (x) &= \begin{cases} M_1(x) - M_2(x) & \text{if } M_1(x) - M_2(x) \geq 0, \\ 0 & \text{otherwise.} \end{cases}
\end{aligned}$$

Definition 2.3 If $\succ$ is an ordering on a multiset S, then its multiset extension $\succ_{mul}$ is defined as follows: $M \succ_{mul} N$, if $M \neq N$ and for all $s \in N - M$ there exists $s' \in M - N$ such that $s' \succ s$.

Proposition 2.3 *If $\succ$ is well-founded, then its multiset extension $\succ_{mul}$ is well-founded on finite multisets.*

2.3 The Associative Commutative Theory with Identities

In order to define the associative commutative theory with identities, we shall briefly present the usual framework of equational theories.

An equation is a pair of terms, written $s \approx t$. If E is a set of equations, we write $s \rightarrow_E t$ to indicate that there exists a term w, a position π in w, a substitution σ, and an equation $u \approx v$ in E, such that $s = w[u\sigma]_\pi$ and $t = w[v\sigma]_\pi$. The relation $\rightarrow_E$ is called the rewrite relation induced by E. We say that s *rewrites* to t by E if $s \rightarrow_E t$. A normal form with respect to $\rightarrow_E$ is said to be *irreducible* by E.

The symmetric-transitive-reflexive closure $\leftrightarrow_E^*$ of $\rightarrow_E$ is called the *equational theory* induced by E. A set E of equations will be called a *rewrite system* if the corresponding rewrite relation $\rightarrow_E$ is the primary object of study. The equations of a rewrite system are also called *rewrite rules*.

A rewrite system R is called *convergent* if the corresponding rewrite relation $\rightarrow_R$ is convergent. A convergent rewrite relation R defines a unique normal form for every term t, denoted by $t \downarrow_R$.

Definition 2.4 For a given set of function symbols $F = F_{free} \uplus F_{AC} \uplus F_{zero}$, we define the *associative commutative theory with identities* of F, denoted by $=_{AC1}$, as the equational theory induced by the set $AC1$ of

- all equations

$$f(X, f(Y), Z) \approx f(X, Y, Z),$$

called *flattening equations* (, and expressing associativity), where $f \in F_{AC}$, X, Y, Z are sequences of terms in $T(F, V)$, $|X| + |Z| \geq 1$, and $|Y| \geq 2$,

- all equations

$$f(X) \approx f(\Pi(X)),$$

called *permutative equations* (, and expressing commutativity), where $f \in F_{AC}$, X is a sequence of terms in $T(F, V)$, $|X| \geq 2$, and $\Pi(X)$ is a permutation of X,

- all equations

$$\begin{aligned} f(X, 0, Y) &\approx f(X, Y) \\ f(Z, 0) &\approx Z \\ f(0, Z) &\approx Z, \end{aligned}$$

called *zero-equations* (, and expressing identity), where $f \in F_{AC1}$, $0 = zero(f)$, X, Y, Z are sequences of terms in $T(F, V)$, $|X| + |Y| \geq 2$, and $|Z| = 1$.

Clearly, every associative commutative theory with identities, where $F_{AC} \neq \emptyset$, is not terminating, due to the permutative equations. The rewrite system consisting of all flattening equations is denoted by $Flatten$, the rewrite system consisting of all permutative equations by $Perm$, and the rewrite system consisting of all zero-equations by $Zero$.

Example 2.2

$$\begin{aligned} x+((1*(y*z))+(0+1)) &\to_{Flatten} x+((1*y*z)+(0+1)) \\ &\to_{Flatten} x+((1*y*z)+0+1) \\ &\to_{Flatten} x+(1*y*z)+0+1 \end{aligned}$$

$$\begin{aligned} 1*(x+(0+y+(0+0))) &\to_{Zero} x+(0+y+(0+0)) \\ &\to_{Zero} x+(0+y+0) \\ &\to_{Zero} x+(0+y) \\ &\to_{Zero} x+y \end{aligned}$$

Proposition 2.4 *The rewrite system $Flatten$ is convergent.*

Proof. The relation $\to_{Flatten}$ is terminating, since every application of a rule to a term decreases the number of occurrences of AC-symbols in it. The relation $\to_{Flatten}$ is Church-Rosser, as can be proved by structural induction. □

Proposition 2.5 *The rewrite system $Zero$ is convergent.*

Proof. The relation $\to_{Zero}$ is terminating, since every application of a rule in $Zero$ to a term t decreases the number of occurrences of identities in t. The relation $\to_{Zero}$ is Church-Rosser, as can be proved by structural induction. □

Proposition 2.6 *The rewrite system $Flatten \cup Zero$ is convergent.*

Proof. The relation $\to_{Flatten \cup Zero}$ is terminating, since every application of a rule in $Flatten \cup Zero$ to a term t decreases the number of occurrences of function symbols in t. The Church-Rosser property then follows from Propositions 2.4 and 2.5. $\square$

Additionally, we can show that $\rightarrow_{Flatten \cup Zero}$ is equal to $\rightarrow_{Zero} \circ \rightarrow_{Flatten}$.

The following technical lemma is used in Chapter 4.

Lemma 2.1 *If s is the subterm of t at position π and $t' =_{AC1} t$, then there exists a position π' in $t' \downarrow_{Flatten \cup Zero}$ such that $t' \downarrow_{Flatten \cup Zero} |_{\pi'} =_{AC1} s$.*

Proof. (1) If $u \rightarrow_{Flatten} v$ or $u \rightarrow_{Zero} v$, then for every subterm u' in u, there exists a subterm v' in v such that $u' =_{AC1} v'$. ($Flatten$ and $Zero$ preserve subterms up to $=_{AC1}$.) (2) If $r =_{AC1} s$ then $r \downarrow_{Flatten \cup Zero} \leftrightarrow^*_{Perm} s \downarrow_{Flatten \cup Zero}$. (Identities are eliminated, for AC-case see Lemma 2.3 in [Mar 91].) (3) If $r \leftrightarrow^*_{Perm} s$ then for every subterm r' in r there exists a subterm s' in s such that $r' \leftrightarrow s'$. (Every subsequence of the arguments of an AC-term forms a subterm.) From (1) follows that there exists a subterm s' of $t \downarrow_{Flatten \cup Zero}$ such that $s =_{AC1} s'$; from (2) that $t \downarrow_{Flatten \cup Zero} \leftrightarrow^*_{Perm} t' \downarrow_{Flatten \cup Zero}$; and from (3) that there exists a subterm s'' in $t' \downarrow_{Flatten \cup Zero}$ such that $s'' \leftrightarrow_{Perm} s'$. $\square$

In this work, we want to describe methods for constructing a rewrite system R for a given set of equations E, such that $AC1 \cup E$ and $AC1 \cup R$ define the same equational theory and the relation $=_{AC1} \circ \rightarrow_R \circ =_{AC1}$ is convergent modulo $AC1$. This task is called *completion modulo AC1.*

As shown in [BPW 89], we need to limit the applicability of rewrite rules, because for many interesting rewrite systems R, the relation $=_{AC1} \circ \rightarrow_R \circ =_{AC1}$ is not terminating. The following example is given in [JM 90].

Example 2.3 If R contains the rule

$$-(x+y) \approx (-x)+(-y)$$

then $R/AC1$ is not terminating, as shown by the following infinite sequence of rewritings:

$$\begin{array}{rcl} -0 & =_{AC1} & -(0+0) \\ & \to_R & (-0)+(-0) \\ & =_{AC1} & -(0+0)+(-0) \\ & \to_R & (-0)+(-0)+(-0) \\ & \vdots & \vdots \end{array}$$

The left hand side of a rule with a variable as argument of an $AC1$-symbol $+$ "collapses" if instanciated with $zero(+)$. For every equation $r \approx s$, we limit its applicability to a term t by restricting the set of substitutions σ by which a subterm of t $AC1$-matches r. Every equation is associated with an expression, called a *constraint*, describing the set of allowed substitutions. In addition to the restrictions involving the identity symbols, we shall integrate *unification problems* in the constraints. In Chapter 4, we shall use this approach to interleave the processes of completion and unification.

2.4 Constraints

Definition 2.5 Let $F = F_{free} \uplus F_{AC} \uplus F_{zero}$ and V a set of variables. A *constraint* is an expression of one of the following forms:

- A *zero-disequation* is an expression of the form $t \neq 0$, where $t \in T(F,V)$ and $0 \in F_{zero}$.

- An *atomic AC1-unification problem* is an expression of the form $r \stackrel{?}{=} s$, where $r, s \in T(F, V)$. Zero-disequations and atomic $AC1$-unification problems are called *atomic constraints*.

- A *conjunctive constraint* is an expression c of the form $c = \bigwedge_{i \in [1..n]} a_i$, where a_i are atomic constraints for $i \in [1..n]$.

- An *existential constraint* is an expression e of the form $e = \exists \vec{z}.(c, k)$, where $\vec{z}$ is a sequence of pairwise distinct variables in V, and c, k are conjunctive constraints.
(We omit the quantification $\exists \vec{z}$, if $\vec{z}$ is empty. Variables in $\vec{z}$ are called *bound*, other variables occuring in c_1 or c_2 are called *free*. The conjunctive constraint c is called the unsolved part and k is called the solved part of the existential constraint.)

- A *disjunctive constraint* is an expression d of the form $d = \bigvee_{i \in [1..n]} e_i$, where e_i are existential constraints for $i \in [1..n]$.

Definition 2.6 For a constraint C, the *set of constrained variables* of C, denoted $V(C)$, is defined as follows: When $a = t \neq 0$ is a zero-disequation, then $V(a) = Var(t)$; when $a = r \stackrel{?}{=} s$ is an atomic $AC1$-unification problem, then $V(a) = Var(r) \cup Var(s)$; when $c = \bigwedge_{i \in [1..n]} a_i$ is a conjunctive constraint, then $V(c) = \bigcup_{i \in [1..n]} Var(a_i)$; when $e = \exists \vec{z}.(c, k)$ is an existential constraint, then $V(e) = (V(c) \cup V(k)) - \vec{z}$; and when $d = \bigvee_{i \in [1..n]} e_i$ is a disjunctive constraint, then $V(d) = \bigcup_{i \in [1..n]} Var(e_i)$.

Definition 2.7 For a constraint C, the *set of solutions* of C, denoted $Sol(C)$, is defined as follows: When $a = t \neq 0$ is a zero-disequation, then $Sol(a) =$

$\{\sigma \in \Sigma(V(t)) \mid t\sigma \neq_{AC1} 0\}$; when $a = r \stackrel{?}{=} s$ is an atomic $AC1$-unification problem, then $Sol(a) = \{\sigma \in \Sigma(V(t)) \mid r\sigma =_{AC1} s\sigma\}$; when $c = \bigwedge_{i \,\in\, [1..n]} a_i$ is a conjunctive constraint, then $Sol(c) = \bigcap_{i \,\in\, [1..n]} Sol(a_i)$; when $e = \exists \vec{z}.(c, k)$ is an existential constraint, then $Sol(e) = \{\sigma|_{V(e)} \mid \sigma \in Sol(c)$ and $\sigma \in Sol(k)\}$; when $d = \bigvee_{i \,\in\, [1..n]} e_i$ is a disjunctive constraint, then $Sol(d) = \bigcup_{i \,\in\, [1..n]} Sol(e_i)$.

Remarks:

- The symbol $\stackrel{?}{=}$ is used in a commutative way: $r \stackrel{?}{=} s$ is considered to be equal to $s \stackrel{?}{=} r$.

- The symbols $\wedge$ and $\vee$ are used in an associative commutative way: $r \stackrel{?}{=} s \wedge c$ denotes that either $r \stackrel{?}{=} s$ or $s \stackrel{?}{=} r$ occurs in a conjunctive constraint and the remainder of this constraint is denoted by c ; $e \vee d$ denotes that e occurs in a disjunction and the remainder of this disjunction is denoted by d.

- The empty conjunctive constraint is denoted by $\top$, and the empty disjunctive constraint is denoted by $\perp$.

- We may denote conjunctions $a_1 \wedge \cdots \wedge a_n$ by $\bigwedge_{i \,\in\, [1..n]} a_i$ and disjunctions $e_1 \vee \cdots \vee e_n$ by $\bigvee_{i \,\in\, [1..n]} e_i$.

- Let $c = \bigwedge_{i \,\in\, [1..n]} r_i \stackrel{?}{=} s_i$ be a conjunction of atomic unification problems. We denote the set $\bigcup_{i \,\in\, [1..n]} r_i \cup \bigcup_{i \,\in\, [1..n]} s_i$ by $\bigcup c$; the set $\bigcup_{i \in [1..n]} subt(r_i) \cup subt(s_i)$ by $subt(c)$; and the set $\bigcup_{i \in [1..n]} Fsubt(r_i) \cup Fsubt(s_i)$ by $Fsubt(c)$.

- We may interpret a conjunction $x_1 \stackrel{?}{=} t_1 \ \wedge \cdots \wedge \ x_n \stackrel{?}{=} t_n$, where $x_i \neq x_j$ if $i \neq j$, as a substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ and vice versa.
- We extend the function Var from terms to existential constraints:

$$\begin{aligned} & Var(\exists \vec{z}.(\textstyle\bigwedge_{i\in[1..k]} t_i \neq 0_i \wedge \bigwedge_{i\in[1..l]} r_i \stackrel{?}{=} s_i, \bigwedge_{i\in[1..m]} t'_i \stackrel{?}{=} 0'_i \wedge \bigwedge_{i\in[1..n]} r'_i \stackrel{?}{=} s'_i)) \\ & \quad = \ \vec{z} \ \cup \ \textstyle\bigcup_{i\in[1..k]} Var(t_i) \ \cup \bigcup_{i\in[1..l]}(Var(r_i) \cup Var(s_i)) \cup \\ & \qquad \textstyle\bigcup_{i\in[1..m]} Var(t'_i) \ \cup \bigcup_{i\in[1..n]}(Var(r'_i) \cup Var(s'_i)), \end{aligned}$$

  and to disjunctive constraints:

$$Var(\bigvee_{h\in[1..k]} e_h) = \bigcup_{h\in[1..k]} Var(e_h).$$

For a given set of function symbols F and a set of variables V, we denote the set of all constraints by C_{AC1} and call its elements *AC1-unification problems with zero-disequations*. When a constraint in C_{AC1} contains no zero-disequations, we call it an *AC1-unification problem*. When $F = F_{free} \uplus F_{AC}$, i.e. $F_{zero} = \emptyset$, we denote the set of all constraints by C_{AC}, and call its elements *AC-unification problems*. Note that the sets C_{AC1} and C_{AC} are constraint languages in the sense of [Smo 89].

Chapter 3

Unification

Before we use constraints to define constrained rewriting, we shall outline in this chapter how to solve them. We approach the problem of finding the set of all solutions for an $AC1$-unification problem with zero-disequations. Prior to describing methods of solving an $AC1$-unification problem with zero-disequations in the general case in Section 3.4, we shall present the solving of an AC-unification problem in Section 3.2 and the solving of an $AC1$-unification problem in Section 3.3. A computer implementation of the procedures presented in this chapter is described in Chapter 5 and examples are given in Appendix B.

AC- and $AC1$-unification problems are inherently complex. Following the intial work in [LS 75] and [Sti 81] in AC-unification, considerable research has been done. The reader may consult [JK 91] for an extensive list of references.

The approach presented herein is not intended to optimize the running time of the solution of a given $AC1$-unification problem but rather to *modularize* the very process of solving such problems. Our goal is to have the capability of interrupting the process at as many stages as possible and "shelve" partially solved problems for future resolution. Meanwhile, the problem may have become less complex. This approach can be particularly useful in the process of completion of term rewriting systems (see Chapter 4), where unification problems are computed and continuously transformed by simplification rules. The main difficulty in modularizing the solving of $AC1$-unification problems is to prove the termination of the resulting procedures. A considerable part of this chapter shall be devoted to termination proofs.

In this chapter, we shall obey the following normalization convention: All terms reduced to normal forms with respect to the rewrite system $Flatten$. Any operation op resulting in terms is implicitly seen as the composition $op \circ \downarrow_{Flatten}$. This will guarantee that all occuring terms are normal forms with respect to $Flatten$.

Example 3.1 The application of the substitution $\sigma = x \mapsto z + w$ to the term $x + y$ results in $z + w + y$, not in $(z + w) + y$.

3.1 Normal Forms

The set of all solutions for a unification problem is infinite in most cases and we are confronted with the problem of finding a finite representation for it. The following notation is taken from [JK 91] and adapted for our purpose.

When a term t $AC1$-matches a term s, we write $s \trianglelefteq t$. The relation $\trianglelefteq$ is a quasi-ordering on terms, called *$AC1$-subsumption*, whose equivalence $\bowtie$ and strict ordering $\triangleleft$ are respectively called *literal $AC1$-similarity* and *strict $AC1$-subsumption*.

We lift subsumption from terms to substitutions. Two substitutions σ and τ are $AC1$-*equal* on the set of variables $W \subseteq V$, denoted $\sigma =_{AC1}^{W} \tau$, if $x\sigma =_{AC1} x\tau$ for all variables x in W. We write $\sigma \trianglelefteq^{W} \tau$, if there exists a substitution ρ such that $\sigma\rho =_{AC1}^{W} \tau$. We call the quasi-ordering $\trianglelefteq^{W}$ on substitutions $AC1$-*subsumption*, its equivalence $\bowtie^{W}$ *literal* $AC1$-*similarity* and its strict ordering $\triangleleft^{W}$ *strict* $AC1$-*subsumption*.

Definition 3.1 Given an $AC1$-unification problem with zero-disequations C, every set $CSU(C)$ that fulfills

- $CSU(C) \subseteq Sol(C)$, (correctness)
- for all substitutions $\theta \in Sol(C)$ there exists a substitution $\sigma \in CSU(C)$ such that $\sigma \trianglelefteq^{Var(c)} \theta$, (completeness)
- $Ran(\sigma) \cap Dom(\sigma) = \emptyset$ for all substitutions $\sigma \in CSU(C)$, (idempotency)

is called a *complete set of AC1-unifiers* for C and denoted by $CSU(C)$.

A set $CSU(C)$ is called a *complete set of most general* $AC1$-*unifiers* of C, denoted $CSMGU(C)$, if the sustitutions are pairwise incomparable in the quasi-ordering $\trianglelefteq^{V(C)}$.

The goal in solving an $AC1$-unification problem with zero-disequations is the transformation of C into a form from which a finite $CSU(C)$ can easily be derived.

Definition 3.2 An $AC1$-unification problem is said to be in *solved form* if it is of the form:

$$\begin{array}{llr} \exists\vec{z_1}. & (\top, \bigwedge_{i\in[1..n_1]} x_{1,i} \neq 0_{1,i} \wedge \bigwedge_{j\in[1..m_1]} y_{1,j} \stackrel{?}{=} t_{1,j}) & \vee \\ \vdots & \vdots & \vdots \\ \exists\vec{z_k}. & (\top, \bigwedge_{i\in[1..n_k]} x_{k,i} \neq 0_{k,i} \wedge \bigwedge_{j\in[1..m_k]} y_{k,j} \stackrel{?}{=} t_{k,j}) & \end{array}$$

such that for all $h \in [1..k]$, for all $i, i' \in [1..n_h]$ and $j, j' \in [1..m_h]$ holds:

- $x_{h,i}, y_{h,j} \in V$, (solved)
- if $t_{h,j} = 0$, where $0 \in F_{zero}$, then there exists no $i'' \in [1..n_h]$ such that $x_{h,i''} = y_{h,j}$ and $0_{h,i''} = 0$, (disequations not violated)
- $y_{h,j} \neq y_{h,j'}$ if $j \neq j'$, (unique)
- $y_{h,j} \notin Var(t_{h,j'})$. (idempotent)

Proposition 3.1 *Let*

$$C = \begin{array}{llr} \exists\vec{z_1}. & (\top, \bigwedge_{i\in[1..n_1]} x_{1,i} \neq 0_{1,i} \wedge \bigwedge_{j\in[1..m_1]} y_{1,j} \stackrel{?}{=} t_{1,j}) & \vee \\ \vdots & \vdots & \vdots \\ \exists\vec{z_k}. & (\top, \bigwedge_{i\in[1..n_k]} x_{k,i} \neq 0_{k,i} \wedge \bigwedge_{j\in[1..m_k]} y_{k,j} \stackrel{?}{=} t_{k,j}) & \end{array}$$

be an $AC1$-unification problem in solved form. The set of substitutions that are obtained for every $h \in [1..k]$ by limiting the substitution $\bigwedge_{j\in[1..m_h]} y_{h,j} \stackrel{?}{=} t_{h,j}$ to the variables that do not occur in $\vec{z_h}$ is a complete set of $AC1$-unifiers for C.

An atomic unification problem $x \stackrel{?}{=} t$ in the solved part of an existential constraint is ignored when $x \in \vec{z}$, because bound variables do not contribute to substitutions in $Sol(C)$.

Note that we solve an $AC1$-unification problem C to obtain a $CSU(C)$, not a $CSMGU(C)$. In cases, where a $CSMGU(C)$ is required, we must detect redundant unifiers in an additional pass over the obtained $CSU(C)$. This operation can be very time-consuming. For this reason, we operate with $CSU(C)$, accepting a certain overhead produced by redundant unifiers.

3.2 AC-Unification

In this section, we consider sets $F = F_{free} \uplus F_{AC}$ (with no identities declared). Our notations are compatible with $AC1$-unification problems with zero-disequations so that we can extend the methods that we describe here to Section 3.3 and Section 3.4.

In the transformation of an $AC1$-unification problem, atomic $AC1$-unification problems of various forms occur. The most difficult case consists of a problem with the same AC-symbol on both sides. The decomposition step for this case forms the heart of most AC-unification procedures. In [Fag 84], an algorithm to decompose these atomic $AC1$-unification problems is presented. Based on this algorithm, we define a *simplifier for AC-unification problems.*

Definition 3.3 A *simplifier for AC-unification problems* is a partial function $dio_{AC} : C_{AC} \times 2^V \to 2^{C_{AC}}$ such that for all $f \in F_{AC}$ and $W \subseteq V$,

$$dio_{AC}(\exists \vec{z}.(f(X) \stackrel{?}{=} f(Y), \top), W) = \left\{ \begin{array}{c} \exists \vec{z_1}.(\bigwedge_{i \in [1..n_1]} r_{1,i} \stackrel{?}{=} s_{1,i}, \top), \\ \exists \vec{z_2}.(\bigwedge_{i \in [1..n_2]} r_{2,i} \stackrel{?}{=} s_{2,i}, \top), \\ \vdots \\ \exists \vec{z_k}(\bigwedge_{i \in [1..n_k]} r_{k,i} \stackrel{?}{=} s_{k,i}, \top) \end{array} \right\}$$

where

$$Sol(\exists \vec{z}.(f(X) \stackrel{?}{=} f(Y), \top)) = \bigcup_{h \in [1..k]} Sol(\exists \vec{z_h}.(\bigwedge_{i \in [1..n_h]} r_{h,i} \stackrel{?}{=} s_{h,i}, \top))$$

and for all $h \in [1..k]$, for all $i, i' \in [1..m_h]$

1. $\vec{z} \subseteq \vec{z_h}$,
2. $r_{h,i} \in X \cup Y$ (see the algorithm `trans` in [Fag 84]),
3. $r_{h,i} \neq r_{h,i'}$ (see the algorithm `elimcom` in [Fag 84]),
4. if $r_{h,i} \in V$ then $s_{h,i} \in X \cup Y$ or $s_{h,i} = f(Z), Z \subseteq (V - W) \cup X \cup Y$ (see the algorithm `trans` in [Fag 84]),
5. if $r_{h,i} \notin V$, then $r_{h,i}$ and $s_{h,i} \in X \cup Y$ (see Proposition 1 in [Fag 84]).

Table 1 shows the set **AC-Unify** of transformation rules whose goal is to transform any AC-unification problem C into an equivalent AC-unification problem in solved form. Each rule describes a binary relation on constraints. The union of these relations is denoted by $\Rightarrow_{\mathbf{AC-Unify}}$.

Definition 3.4 An *AC-unification procedure* is a subset U of the relation $\Rightarrow_{\mathbf{AC-Unify}}$ such that every normal form with respect to U is a normal form with respect to $\Rightarrow_{\mathbf{AC-Unify}}$.

An AC-unification procedure applies the rules in **AC-Unify** to an AC-unification problem until no further rules are applicable.

Lemma 3.1 (Soundness) *Every AC-unification procedure preserves the unifiers: If $C_1 \Rightarrow^*_{\mathbf{AC-Unify}} C_2$ then $Sol(C_1) = Sol(C_2)$.*

Delete:

$$\exists\vec{z}.(r\overset{?}{=}s \wedge c, k) \ \vee \ d \quad \Rightarrow \quad \exists\vec{z}.(c, k) \ \vee \ d$$
$$\text{if } r =_{AC1} s$$

Fail:

$$\exists\vec{z}.(f(X)\overset{?}{=}g(Y) \wedge c, k) \ \vee \ d \quad \Rightarrow \quad d$$
$$\text{if } f \neq g$$

Merge:

$$\exists\vec{z}.(x\overset{?}{=}r \wedge x\overset{?}{=}s \wedge c, k) \ \vee \ d \quad \Rightarrow \quad \exists\vec{z}.(x\overset{?}{=}r \wedge r\overset{?}{=}s \wedge c, k) \ \vee \ d$$
$$\text{if } \begin{cases} x \in V \\ r \notin V \end{cases}$$

Check:

$$\exists\vec{z}.(x\overset{?}{=}f(X) \wedge c, k) \ \vee \ d \quad \Rightarrow \quad d$$
$$\text{if } \begin{cases} x \in V \\ x \in Var(f(X)) \end{cases}$$

Eliminate:

$$\exists\vec{z}.(x\overset{?}{=}t \wedge c, k) \ \vee \ d \quad \Rightarrow \quad \exists\vec{z}.(c\sigma, x\overset{?}{=}t \wedge k\sigma) \ \vee \ d$$
$$\text{where } \sigma = \{x \mapsto t\}$$
$$\text{if } \begin{cases} x \in V \\ x \notin Var(t) \end{cases}$$

Free Decompose:

$$\exists\vec{z}.(f(X)\overset{?}{=}f(Y) \wedge c, k) \ \vee \ d \quad \Rightarrow \quad \exists\vec{z}.(\textstyle\bigwedge_{i\in[1..|X|]} X^i\overset{?}{=}Y^i, k) \ \vee \ d$$
$$\text{if } \begin{cases} f \in F_{free} \\ f(X) \neq_{AC1} f(Y) \\ f(X) \notin subt(c) \text{ or } f(Y) \notin subt(c) \end{cases}$$

AC Decompose:

$$\exists\vec{z}.(f(X)\overset{?}{=}f(Y) \wedge c, k) \ \vee \ d \quad \Rightarrow$$
$$\bigvee_{(\exists\vec{z'}.(c',\top))\in dio_{AC}(\exists\vec{z}.(f(X)\overset{?}{=}f(Y),\top),W)} \exists\vec{z'}.(c' \wedge c, k) \ \vee \ d$$
$$\text{where } W = Var(\exists\vec{z}.(f(X)\overset{?}{=}f(Y) \wedge c, k))$$
$$\text{if } \begin{cases} f \in F_{AC} \\ f(X) \neq_{AC1} f(Y) \\ f(X) \notin subt(c) \text{ or } f(Y) \notin subt(c) \end{cases}$$

Table 1: The transformation rules **AC-Unify**

Proof. For every transformation rule R in **AC-Unify**, we prove that if $C_1 \Rightarrow_R C_2$ then $Sol(C_1) = Sol(C_2)$. The lemma follows by induction.

- **Delete.** If $r =_{AC1} s$ then any substitution is in $Sol(r \stackrel{?}{=} s)$, so that for $C_1 = \exists\vec{z}.(r \stackrel{?}{=} s \wedge c, k) \vee d$ and $C_2 = \exists\vec{z}.(c, k) \vee d$ holds $Sol(C_1) = Sol(C_2)$.

- **Fail.** If $C_1 = \exists\vec{z}.(f(X) \stackrel{?}{=} g(Y) \wedge c, k) \vee d$ and $f \neq g$, then $Sol(f(X) \stackrel{?}{=} g(Y))$ $= \emptyset$, because, since $F_{zero} = \emptyset$, $AC1$ does not change the root symbol of any term. Thus, for $C_2 = d$, we have $Sol(C_1) = Sol(C_2)$.

- **Merge.** If $C_1 = \exists\vec{z}.(x \stackrel{?}{=} r \wedge x \stackrel{?}{=} s \wedge c, k) \ \vee \ d$ and $C_2 = \exists\vec{z}.(x \stackrel{?}{=} r \wedge r \stackrel{?}{=} s \wedge c, k) \ \vee \ d$, then $\sigma \in Sol(x \stackrel{?}{=} r \wedge x \stackrel{?}{=} s) \Leftrightarrow x\sigma =_{AC1} r\sigma \ and \ x\sigma =_{AC1} s\sigma \Leftrightarrow x\sigma =_{AC1} r\sigma \ and \ r\sigma =_{AC1} s\sigma$ ($=_{AC1}$ is transitive) $\Leftrightarrow \sigma \in Sol(x \stackrel{?}{=} r \wedge r \stackrel{?}{=} s)$ Therefore $Sol(x \stackrel{?}{=} r \wedge x \stackrel{?}{=} s) = Sol(x \stackrel{?}{=} r \wedge r \stackrel{?}{=} s)$.

- **Check.** Let $C_1 = \exists\vec{z}.(x \stackrel{?}{=} f(X) \wedge c, k) \ \vee \ d$ and $x \in Var(f(X))$. Assume $\sigma \in Sol(x \stackrel{?}{=} f(X))$. Since $F_{zero} = \emptyset$, $AC1$ is non-collapsing. On the other hand $x\sigma$ is a proper subterm of $f(X)\sigma$, which is a contradiction. Thus, $Sol(x \stackrel{?}{=} t) = \emptyset$, and for $C_2 = d$, we obtain $Sol(C_1) = Sol(C_2)$.

- **Eliminate.** Let $C_1 = \exists\vec{z}.(x \stackrel{?}{=} t \wedge c, k) \vee d$ and $C_2 = \exists\vec{z}.(c\sigma, x \stackrel{?}{=} t \wedge k\sigma) \vee d$, where $\sigma = \{x \mapsto t\}$. Let $\rho \in Sol(x \stackrel{?}{=} t \wedge c, k)$. First, we state $x\rho =_{AC1} t\rho$. Let $r \stackrel{?}{=} s$ be any atomic $AC1$-unification problem in c or k. Then $\rho \in Sol(r \stackrel{?}{=} s)$, and thus $r\rho =_{AC1} s\rho$. We have $\sigma\rho =_{AC1} \rho$, i.e. $y\sigma\rho =_{AC1} y\rho$ for all $y \in V$. (If $y \neq x$, then σ does not change y; if $y = x$, then $y\sigma = t$ and $t\rho =_{AC1} x\rho = y\rho$.) $r\rho =_{AC1} s\rho \ \Leftrightarrow \ r\sigma\rho =_{AC1} s\sigma\rho$, and therefore $\rho \in Sol(r \stackrel{?}{=} s) \Leftrightarrow \rho \in Sol(r\sigma \stackrel{?}{=} s\sigma)$. Thus, $Sol(x \stackrel{?}{=} t \wedge c, k) = Sol(c\sigma, x \stackrel{?}{=} t \wedge k\sigma)$ and $Sol(C_1) = Sol(C_2)$.

- **Free Decompose.** Let $C_1 = \exists\vec{z}.(f(X)\stackrel{?}{=}f(Y)\wedge c, k) \vee d$, $f \in F_{free}$, and $C_2 = \exists\vec{z}.(\bigwedge_{i\in[1..|X|]} X^i\stackrel{?}{=}Y^i \wedge c, k) \vee d$. The theory $=_{AC1}$ does not involve the symbol f. Thus, for a substitution θ to fulfill $f(X^1,\ldots,X^n)\theta =_{AC1} f(Y^1,\ldots,Y^n)\theta$, we must have $X^i\theta =_{AC1} Y^i\theta$, for $i \in [1..n]$. Therefore $Sol(\exists\vec{z}.(f(X)\stackrel{?}{=}f(Y)\wedge c, k)) = Sol(\exists\vec{z}.(\bigwedge_{i\in[1..|X|]} X^i\stackrel{?}{=}Y^i \wedge c, k))$ and $Sol(C_1) = Sol(C_2)$.

- **AC Decompose.** Let $C_1 = \exists\vec{z}.(f(X)\stackrel{?}{=}f(Y) \wedge c, k) \vee d$, $root(r) = root(s) \in F_{AC}$, and $C_2 = \bigvee_{(\exists\vec{z'}.(c',\top))\in dio_{AC}(\exists\vec{z}.(f(X)\stackrel{?}{=}f(Y),\top),W)} \exists\vec{z'}.(c' \wedge c, k) \vee d$, where $W = Var(\exists\vec{z}.\ (f(X)\stackrel{?}{=}f(Y) \wedge c, k))$. By definition of dio_{AC} (see Definition 3.3), we obtain that $Sol(\exists\vec{z}.f(X)\stackrel{?}{=}f(Y),\top) = \bigcup_{e\in dio_{AC}(\exists\vec{z}.(f(X)\stackrel{?}{=}f(Y),\top),W)} Sol(e)$. By distributivity, we obtain that $Sol(\exists\vec{z}.(f(X)\stackrel{?}{=}f(Y)\wedge c, k)) = Sol(\bigvee_{(\exists\vec{z'}.(c',\top))\in dio_{AC}(\exists\vec{z}.(f(X)\stackrel{?}{=}f(Y),\top),W)} \exists\vec{z'}.(c' \wedge c, k))$. Here, we use that the variables introduced by **AC Decompose** do not occur in C_1.

$\square$

Lemma 3.2 (Completeness) *Every normal form with respect to* **AC-Unify** *is an AC-unification problem in solved form.*

Proof. Clearly, every normal form is of the form

$$C = \bigvee_{h\in[1..k]} \exists\vec{z_h}.(c_h, k_h)$$

where c_h and k_h are conjunctive constraints for all $h \in [1..k]$.

Assume c_h contains an atomic $AC1$-unification problem $r \stackrel{?}{=} s$. Then r and s cannot be both non-variable terms, for otherwise **Delete**, **Fail**, **Free Decompose**, or **AC Decompose** would apply. Note, that if all atomic $AC1$-unification problems in c_i are of the form $f(X) \stackrel{?}{=} f(Y)$ then **Free Decompose** or **AC Decompose** applies to at least one of them. (In a set of terms, not all terms can be proper subterms of other terms in the set.)

So, assume that $r = x \in V$. If $s = x$, **Delete** applies. If $s \neq x$, then **Check** applies if $x \in Var(s)$ and **Eliminate** applies otherwise. Since AC-unification problems do not contain zero-disequations, $c_i = \top$, for all i, and $C = \bigvee_{h \in [1..k]} \exists \vec{z_h}.(\top, \bigwedge_{i \in [1..n_h]} r_{h,i} \stackrel{?}{=} s_{h,i})$. The only rule that changes the solved part of an existential constraint is **Eliminate**. The rule **Eliminate** removes an atomic $AC1$-unification problem of the form $x \stackrel{?}{=} t$, where $x \in V$, from the unsolved part. The variable x is eliminated from the unsolved and the solved part, and $x \stackrel{?}{=} t$ is added to the solved part. The variable x will not occur in the unsolved part again, because **AC Decompose** uses only variables in the solved part and variables that do not occur in the existential constraint, and the other rules use only variables that occur in the solved part. If a variable x occurs on the left hand side of an atomic $AC1$-unification problem in an existential constraint e in C, then x occurs nowhere else in e (idempotency and uniqueness). We conclude that C is of the form $C = \bigvee_{h \in [1..k]} \exists \vec{z_1}.(\top, \bigwedge_{i \in [1..n_1]} x_{1,i} \stackrel{?}{=} t_{1,i})$ such that for all $h \in [1..k]$ and $i, i' \in [1..n_h]$, $x_{h,i} \in V$, $x_{h,i} \neq x_{h,i'}$ if $i \neq i'$, and $x_{h,i} \notin Var(t_{h,i'})$. $\square$

Not all AC-unification procedures are terminating (for an example see [Fag 84]). One way to guarantee termination is to apply the decomposition step recursively in a sense that once an atomic $AC1$-unification problem $f(X)\stackrel{?}{=}f(Y)$ is simplified, all emerging subproblems are completely solved before another atomic problem is examined ([Sti 81],[Fag 84]). This method is not satisfactory for our approach to completion modulo $AC1$, where we want to be able to interrupt the solving of a constraint when a particularly hard atomic $AC1$-unification problem arises. Other approaches to AC-unification are described in [Kir 89] and [Bou 90].

We shall present a class of terminating AC-unification procedures that are based on [Fag 84] but can be interrupted between every application of **AC Decompose**. We require that, after the application of **Free Decompose** and **AC Decompose**, some rules are "eagerly" applied. To prove the termination of these AC-unification procedures, we develop an extension of the complexity used in the termination proof of [Fag 84].

In the following, let @ be a new symbol. The set of *immediate operators* of a variable x in a term t is the set

$$OP(x,t) = \begin{cases} \{root(t|_{\langle p,\emptyset\rangle}) \mid \langle p,\emptyset\rangle \in pos(t) \ and \ \exists i \in \mathbf{N}.\ t|_{\langle\langle p,i\rangle,\emptyset\rangle} = x\} & \\ & \text{if } x \neq t, \\ \{@\} & \text{if } x = t. \end{cases}$$

(We include the occurrence of a variable in the topmost position: If $x = t$, we say x occurs in t under @.)

The set of immediate operators of a variable x in an existential constraint

$$e = \exists \vec{z}.(\bigwedge_{i\in[1..n]} x_i \neq 0_i \ \wedge \bigwedge_{j\in[1..m]} r_j \stackrel{?}{=} s_j \ , \ k)$$

is the set

$$OP(x,e) \;=\; \bigcup_{j\in[1..m]} OP(x,r_j) \cup OP(x,s_j).$$

Definition 3.5 The complexity μ_e of an existential $AC1$ constraint $e = \exists\vec{z}.\,(c \wedge g, k)$, where $g = \bigwedge_{i\in[1..n]} x_i \neq 0_i$ and $c = \bigwedge_{j\in[1..n]} r_j \stackrel{?}{=} s_j$, is defined as

$$\mu_e(e) = (\alpha(e), \beta(e), \gamma(e), \delta(e), \varepsilon(e), \zeta(e), \eta(e)),$$

where

- $\alpha(e) = |\{i \mid r_i \in V \text{ and } s_i \in V\}|$, $\alpha(e)$ is the number of atomic $AC1$-unification problems between variables in the unsolved part;
- $\beta(e) = |\{x \in V \mid |OP(x,e)| \geq 2\}|$, $\beta(e)$ is the number of distinct variables that occur in r_i and s_i immediately under at least two different function symbols;
- $\gamma(e) = |Fsubt(e) - \{0(\lambda) \in T(F,V) \mid 0 \in F_{zero}\}|$, $\gamma(e)$ is the number of distinct non-variable subterms of r_i and s_i;
- $\delta(e) = |\{i \mid r_i \in V \text{ or } s_i \in V\}|$, $\delta(e)$ is the number of atomic $AC1$-unification problems in the unsolved part of which at least one side consists of a variable;
- $\varepsilon(e) = |\{i \mid r_i \notin V, r_i \notin subt(c) - \bigcup c, r_i \notin F_{zero}\}| \;+\; |\{i \mid s_i \notin V, s_i \notin subt(c) - \bigcup c, s_i \notin F_{zero}\}|$, $\varepsilon(e)$ is the number of occurrences of non-variable terms on one side of an atomic problem in c that are not identities and not proper subterms of terms in c;
- $\zeta(e) = \sum_{i\,\in\,[1..n]}(|\{\pi \in pos(r_i) \mid r_i|_p \notin F_{zero}\}| + |\{\pi \in pos(s_i) \mid s_i|_p \notin F_{zero}\}|)$, $\zeta(e)$ is the number of all positions in e at which no identities occur; and

- $\eta(e) = n$, $\eta(e)$ is the number of atomic AC-unification problems in c.

The complexity μ_d of a disjunctive AC-unification problem $d = \bigvee_{i \in [1..n]} e_i$ is defined as the multiset $\mu_d(d) = \{\mu_e(e_i) \mid i \in [1..n]\}_{mul}$.

Example 3.2 The constraint

$$e \;=\; \exists v, w.(x+v+w \stackrel{?}{=} f(a)+g(f(a),x) \;\wedge\; v \stackrel{?}{=} x \;\wedge\; x * f(v) \stackrel{?}{=} 1 \;\wedge\; z \stackrel{?}{=} f(a), y \stackrel{?}{=} b)$$

has the complexity $\mu_e(e) = (1,2,6,3,3,20,4)$. As a disjunctive constraint, e has the complexity $\mu_d(e) = \{(1,2,6,3,3,20,4)\}_{mul}$.

We use the seven-fold lexicographic combination of the "greater-than"-ordering $>$ on natural numbers, denoted $>_e$, to compare complexities of existential constraints, and the multiset extension of $>_e$, denoted $>_d$, to compare complexities of disjunctive constraints. Propositions 2.1 and 2.3 imply that $>_e$ and $>_d$ are well-founded.

Lemma 3.3 (Termination) *Every AC-unification procedure in which, after every application of* **Free Decompose** *and* **AC Decompose***, all newly introduced atomic problems with variables on at least one side are eliminated via* **Delete***,* **Check** *and* **Eliminate** *terminates on every AC-unification problem.*

Proof. We prove that if $C_1 \Rightarrow_R C_2$ for a rule $R =$ **Delete**, **Fail**, **Merge**, **Check**, or **Eliminate** then $\mu_d(C_1) >_d \mu_d(C_2)$; and if $C_1 \Rightarrow_R C_2'$ for a rule $R =$ **Free Decompose**, **AC Decompose**, and $C_2 = \Rightarrow^!_Q (C_2')$, where $Q =$ **Delete**$\cup$**Check**$\cup$**Eliminate**, then $\mu_d(C_1) >_d \mu_d(C_2)$. The termination follows by induction on the number of applications of rules in **AC-Unify** to the given AC-unification problem.

- **Delete.** If $e_1 = \exists\vec{z}.(r\stackrel{?}{=}s \wedge c, k)$, $e_2 = \exists\vec{z}.(c, k)$, $C_1 = e_1 \vee d$, and $C_2 = e_2 \vee d$, then $\Gamma(e_1) \geq \Gamma(e_2)$, where $\Gamma \in \{\alpha, \beta, \gamma, \delta, \varepsilon, \zeta\}$, while $\eta(e_1) > \eta(e_2)$, and therefore $\mu_d(C_1) >_d \mu_d(C_2)$.

- **Fail.** If $C_1 = \exists\vec{z}.(f(X)\stackrel{?}{=}g(Y) \wedge c, k) \ \vee \ d$, and $C_2 = d$, then $\mu_d(C_1) = \mu_d(C_2) \cup \{\mu_e(\exists\vec{z}.(f(X)\stackrel{?}{=}g(Y) \wedge c, k))\}_{mul} >_d \mu_d(C_2)$.

- **Merge.** Let $e_1 = \exists\vec{z}.(x\stackrel{?}{=}r \wedge x\stackrel{?}{=}s \wedge c, k)$, where $r \notin V$, $e_2 = \exists\vec{z}.(x\stackrel{?}{=}r \wedge r\stackrel{?}{=}s \wedge c, k)$, $C_1 = e_1 \vee \ d$, and $C_2 = e_2 \vee \ d$. We have $\beta(e_1) \geq \ \beta(e_2)$ and $\gamma(e_2) \geq \gamma(e_2)$. If $s \in V$ then $\alpha(e_1) > \alpha(e_2)$, and otherwise $\delta(e_1) > \delta(e_2)$. Thus we obtain $\mu_d(C_1) >_d \mu_d(C_2)$.

- **Check.** Similar to **Fail**.

- **Eliminate.** Let $e_1 = \exists\vec{z}.(x\stackrel{?}{=}t \wedge c, k)$, $e_2 = \exists\vec{z}.(c\sigma, k)$, where $\sigma = \{x \mapsto t\}$, $C_1 = e_1 \vee d$, and $C_2 = e_2 \vee d$. If $t \in V$ then $\alpha(e_1) > \alpha(e_2)$. Otherwise $\alpha(e_1) \geq \alpha(e_2)$, $\beta(e_1) \geq \beta(e_2)$ (Since $t \notin V$, no variable is placed under a new symbol.), $\gamma(e_1) \geq \gamma(e_2)$, and $\delta(e_1) > \delta(e_2)$, thus $\mu_d(C_1) >_d \mu_d(C_2)$.

- **Free Decompose.** We prove that the application of **Free Decompose**, followed by the exhaustive application of **Delete**, **Check** and **Eliminate** on the newly introduced atomic problems, decreases the complexity of an AC-unification problem.

  Let $e_1 = \exists\vec{z}.(f(X)\stackrel{?}{=}f(Y) \wedge c, k)$, where $f \in F_{free}$ and $|X| = |Y|$, $e_2' = \exists\vec{z}.(\bigwedge_{i\in[1..|X|]} X^i\stackrel{?}{=}Y^i \ \wedge c, k)$, and $e_2 = \exists\vec{z}.(c'\sigma \wedge c\sigma, \sigma \wedge k\sigma)$, where c' and σ are defined as follows:

  Let $(x_1\stackrel{?}{=}t_1 \ \wedge \cdots \wedge \ x_n\stackrel{?}{=}t_n \ \wedge \ r_1\stackrel{?}{=}s_1 \ \wedge \cdots \wedge \ r_m\stackrel{?}{=}s_m) = \ \bigwedge_{i\in[1..|X|]} X^i\stackrel{?}{=}Y^i$, where $x_i \in V$ for all $i \in [1..n]$, and r_i or $s_i \notin V$ for all $i \in [1..m]$.

(In the decomposition, we separate the atomic problems of which at least one side consists of a variable from the rest.)

Let R be the relation **Delete** $\cup$ **Check** $\cup$ **Eliminate**. Then $\sigma = \Rightarrow^!_R (x_1 \stackrel{?}{=} t_1 \ \wedge \cdots \wedge \ x_n \stackrel{?}{=} t_n)$ and $c' = (r_1 \stackrel{?}{=} s_1 \ \wedge \cdots \wedge \ r_m \stackrel{?}{=} s_m)$.

Let $C_1 = e_1 \vee d$ and $C_2 = e_2 \vee d$.

We show: $\mu_e(e_1) >_e \mu_e(e_2)$.

We obtain $\alpha(e_1) \geq \alpha(e_2)$, since σ eliminates the atomic problems with variables on both sides that are introduced by the decomposition.

We show: $\beta(e_1) \geq \beta(e_2)$.

Because σ may substitute a variable by another variable, there may exist variables x with $|OP(x, e_2)| \geq 2$ and $|OP(x, e_1)| \not\geq 2$. In this case, there exists $g \in F$ with $g \neq f$ such that $g \in OP(x, e_2)$ and $g \notin OP(x, e_1)$, and there exists $y \in V$ such that $(y \mapsto x) \in \sigma$ and $g \in OP(y, e_1)$. Furthermore, $f \in OP(y, e_1)$ (because $y \in X \cup Y$) and therefore $|OP(y, e_1)| \geq 2$. The variable y does not occur in e_2, therefore $|OP(y, e_2)| \not\geq 2$.

In other words, whenever there is a variable x such that $|OP(x, e_2)| \geq 2$ and $|OP(x, e_1)| \not\geq 2$, there is a corresponding variable y such that $|OP(y, e_2)| \not\geq 2$ and $|OP(y, e_1) \geq 2$. Thus $\beta(e_1) \geq \beta(e_2)$.

If at most one of the terms $f(X)\sigma$ and $f(Y)\sigma$ occurs in e_2, then $\gamma(e_1) > \gamma(e_2)$, and $\mu_e(e_1) >_e \mu_e(e_2)$.

Now, assume that both $f(X)\sigma$ and $f(Y)\sigma$ occur in e_2. None of the occurrences of terms in $c'\ \sigma$ count for $\varepsilon(e_2)$, since they are proper subterms of $f(X)\sigma$ or $f(Y)\sigma$. At least one of the occurrences of $f(X)$

and $f(Y)$ counts for $\varepsilon(e_1)$ (see condition to apply **Free Eliminate**). Clearly, this occurrence has no corresponding occurrence in e_2. Therefore $\varepsilon(e_1) > \varepsilon(e_2)$, and $\mu_e(e_1) >_e \mu_e(e_2)$.

We conclude that $\mu_d(C_1) >_d \mu_d(C_2)$.

- **AC Decompose.** We prove that the application of **AC Decompose**, followed by the exhaustive application of **Delete**, **Check**, and **Eliminate** on the newly introduced atomic problems, decreases the complexity of an AC-unification problem. To describe the eager application of **Delete**, **Check**, and **Eliminate** more precisely, we define the function $elim$:

  Let $e = \exists\vec{z}.(f(X)\stackrel{?}{=}f(Y), \top)$, $f \in F_{AC}$ and R be the relation **Delete** $\cup$ **Check** $\cup$ **Eliminate**. Let

$$\begin{array}{ll} \exists\vec{z_1}.(c_1, \sigma_1) & \vee \\ \vdots & \vdots \\ \exists\vec{z_n}.(c_n, \sigma_n) & \end{array} = \Rightarrow^!_R \Big(\bigvee_{i\in[1..|dio_{AC}(e)|]} dio_{AC}(e)\Big).$$

  Then $elim(e) = \{(z_i, c_i, \sigma_i) \mid i \in [1..n]\}$.

  Now, let $C_1 = \exists\vec{z}.(f(X)\stackrel{?}{=}f(Y) \wedge c, k) \ \vee \ d$, where $f \in F_{AC}$, and $C_2 = \bigvee_{(z_i,c_i,\sigma_i)\in elim(e)} \exists\vec{z_i}.(c_i\sigma_i \wedge c\sigma_i, \sigma_i \wedge k\sigma_i) \vee d$.

  Let $e_1 = \exists\vec{z}.(f(X)\stackrel{?}{=}f(Y) \wedge c, k)$, and $e_2 = \exists\vec{z_j}.(c_j\sigma_j \wedge c\sigma_j, \sigma_j \wedge k\sigma_j)$, where $(z_j, c_j, \sigma_j) \in elim(e)$.

  The conjunctive constraint c_j consists of proper non-variable subterms of $f(X)$ and $f(Y)$ (see Property 5 in Definition 3.3). The substitution σ_j represents the accumulated substitutions resulting from exhaustive elimination using the atomic problems introduced by dio_{AC}. If $(x \mapsto t) \in \sigma_j$, then $x \in X \cup Y$ (see Property 4 in Definition 3.3).

We obtain $\alpha(e_1) \geq \alpha(e_2)$, since σ_j eliminates the atomic problems with variables on both sides that are introduced by the decomposition, and $\beta(e_1) \geq \beta(e_2)$ (argument similar to **Free Decompose**).

If there exists $(x \mapsto t) \in \sigma_j$ with $t \notin V$, such that $|OP(x, e_1)| \geq 2$ then $\beta(e_1) > \beta(e_2)$, and $\mu_e(e_1) >_e \mu_e(e_2)$.

Now, we assume that $OP(x, e_1) = \{f\}$ for all $(x \mapsto t) \in \sigma_j$ where $t \notin V$. Let $e_2' = \exists \vec{z_j}.(f(X)\sigma_j \stackrel{?}{=} f(Y)\sigma_j \wedge c_j\sigma_j \wedge c\sigma_j, \sigma_j \wedge k\sigma_j)$.

We can show by induction on the number of applications of **Eliminate** in the construction of σ_j that $\gamma(e_1) \geq \gamma(e_2')$. (Here, we use that $Flatten$ applies, when we substitute a variable argument in a term with root f by another term with root f. Because $OP(x, e_1) = \{f\}$ for all $(x \mapsto t) \in \sigma_j$ where $t \notin V$, every newly built term is flattened into another term.)

If at most one of the terms $f(X)\sigma_j$ and $f(Y)\sigma_j$ occurs in e_2, then $\gamma(e_1) > \gamma(e_2)$, and $\mu_e(e_1) >_e \mu_e(e_2)$.

Now, assume that both $f(X)\sigma_j$ and $f(Y)\sigma_j$ occur in e_2. We obtain $\delta(e_1) \geq \delta(e_2)$, since all newly introduced atomic $AC1$-unification problems are eliminated. None of the occurrences of terms in $c_j\sigma_j$ count for $\varepsilon(e_2)$, since they are proper subterms of $f(X)\sigma_j$ or $f(Y)\sigma_j$. The constraint $c\sigma$ has at most as many occurrences of terms that count for $\varepsilon(e_2)$ than has c that count for $\varepsilon(e_1)$. (Here, we use that $@ \notin OP(x, c)$, for all x with $(x \mapsto t) \in \sigma_j$ and $t \notin V$.)

Furthermore, at least one of the occurrences of $f(X)$ and $f(Y)$ in the atomic problem $f(X)\stackrel{?}{=}f(Y)$ counts for $\varepsilon(e_1)$ (see condition for application of **AC Decompose**). Clearly, this occurrence has no corresponding occurrence in e_2. Therefore $\varepsilon(e_1) > \varepsilon(e_2)$, and $\mu_e(e_1) >_e \mu_e(e_2)$.

We have shown that for any existential constraint $e_2 \in \bigcup_{(z_i,c_i,\sigma_i)\in elim(e)} \exists \vec{z_i}.(c_i\sigma_i \wedge c\sigma_i, \sigma_i \wedge k\sigma_i)$ holds $\mu_e(e_1) >_e \mu_e(e_2)$.

To obtain C_2, the existential constraint e_1 is replaced in C_1 by a finite number of existential constraints such that e_1 is bigger than all of them under $>_e$. By definition of the multiset extension, we obtain $\mu_d(C_1) >_d \mu_d(C_2)$.

□

3.3 AC1-Unification

The equations $f(Z,0) \approx Z$ and $f(0,Z) \approx Z$ are collapsing, in the sense that one side consists of a term that occurs in the other side as a proper subterm. This makes solving of $AC1$-unification problems more complicated than solving AC-unification problems.

Similar to the previous section, we define a *simplifier for AC1-unification problems* to describe the decomposition of atomic $AC1$-unification problems with $AC1$-terms on both sides.

Definition 3.6 A *simplifier for $AC1$-unification problems* is a partial function

$dio_{AC1} : C_{AC1} \times 2^V \rightarrow 2^{C_{AC1}}$ such that for all $f \in F_{AC1}$ and $W \subseteq V$,

$$dio_{AC}(\exists\vec{z}.(f(X)\stackrel{?}{=}f(Y), \top), W) \;=\; \left\{\begin{array}{ll} \exists\vec{z_1}. & (\bigwedge_{i\in[1..n_1]} r_{1,i}\stackrel{?}{=}s_{1,i}, \top), \\ \exists\vec{z_2}. & (\bigwedge_{i\in[1..n_2]} r_{2,i}\stackrel{?}{=}s_{2,i}, \top), \\ \vdots & \qquad\vdots \\ \exists\vec{z_k}. & (\bigwedge_{i\in[1..n_k]} r_{k,i}\stackrel{?}{=}s_{k,i}, \top), \\ \vdots & \qquad\vdots \end{array}\right\}$$

where

$$Sol(\exists\vec{z}.(f(X)\stackrel{?}{=}f(Y), \top)) = \bigcup_{h\in[1..k]} Sol(\exists\vec{z_h}.(\bigwedge_{i\in[1..n_h]} r_{h,i}\stackrel{?}{=}s_{h,i}, \top))$$

and for all $h \in [1..k]$, for all $i, i' \in [1..m_h]$

1. $\vec{z} \subseteq \vec{z_h}$,
2. $r_{h,i} \in X \cup Y$,
3. $r_{h,i} \neq r_{h,i'}$,
4. if $r_{h,i} \in V$ or $root(r_{h,i}) \in F_{AC1}$ then $s_{h,i} \in X \cup Y$, or $s_{h,i} = f(Z)$, where $Z \subseteq (V - W) \cup X \cup Y$,
5. if $r_{h,i} \notin V$ and $root(r_{h,i}) \notin F_{AC1}$ then $r_{h,i}$ and $s_{h,i} \in X \cup Y$.

The simplification is often much easier with than without identities (for many examples see [BHK 88]).

Note that $AC1$-terms in X and Y must be handled like variables, because they may collapse and unify with any other term. Therefore, we cannot maintain the properties 4 and 5 in Definition 3.3. However, if F_{AC} contains only one element, then no $AC1$-term can occur in X and Y, since we consider flattened terms. In this case, the properties 4 and 5 in Definition 3.3 are valid (see Lemma 3.6).

The set **AC1-Unify** of transformation rules consists of the rules **Delete**, **Merge**, **Eliminate**, **Free Decompose**, and **AC Decompose** in Table 1 and the rules in Table 2. The union of the relations described by the rules in **AC1-Unify** is denoted by $\Rightarrow_{\mathbf{AC1-Unify}}$.

Definition 3.7 An *AC1-unification procedure* is a subset U of the relation $\Rightarrow_{\mathbf{AC1-Unify}}$ such that every normal form with respect to U is a normal form with respect to $\Rightarrow_{\mathbf{AC1-Unify}}$.

Similar to the previous section, we state the soundness, completeness, and termination of $AC1$-unification procedures. We give only the parts of the proofs that differ from the corresponding proof in the previous section.

Lemma 3.4 (Soundness) *Every $AC1$-unification procedure preserves the unifiers: If $C_1 \Rightarrow^*_{AC1-Unify} C_2$, then $Sol(C_1) = Sol(C_2)$.*

Proof.

- **Fail.** If $f, g \notin F_{AC1}$ and $f \neq g$ then $Sol(f(X) \stackrel{?}{=} g(Y)) = \emptyset$, since $AC1$ can only change the root symbols of terms t with $root(t) \in F_{AC1}$. Thus, for $C_1 = \exists \vec{z}.(f(X) \stackrel{?}{=} g(Y) \wedge c, k) \ \vee \ d$ and $C_2 = d$, we obtain $Sol(C_1) = Sol(C_2)$.

- **Check.** Let $C_1 = \exists \vec{z}.(x \stackrel{?}{=} f(X) \wedge c, k) \ \vee \ d$, $f \notin F_{AC1}$, and $x \in Var(f(X))$. Assume $\sigma \in Sol(x \stackrel{?}{=} f(X))$. Since x occurs in $f(X)$, every subterm of $x\sigma$ must occur in $f(X)\sigma$. Since $f \notin F_{AC1}$, $root(x\sigma) =$

Fail:

$$\exists \vec{z}.(f(X)\stackrel{?}{=}g(Y) \wedge c, k) \ \vee \ d \quad \Rightarrow \quad d$$
$$\text{if} \begin{cases} f, g \notin F_{AC1} \\ f \neq g \end{cases}$$

Check:

$$\exists \vec{z}.(x\stackrel{?}{=}f(X) \wedge c, k) \ \vee \ d \quad \Rightarrow \quad d$$
$$\text{if} \begin{cases} x \in V \\ f \notin F_{AC1} \\ x \in Var(f(X)) \end{cases}$$

Collapse 1:

$$\exists \vec{z}.(f(X)\stackrel{?}{=}t \wedge c, k) \ \vee \ d \quad \Rightarrow$$
$$\bigvee_{i \in [1..|X|]} \exists \vec{z}.(\bigwedge_{j \in [1..|X|]; j \neq i} X^j \stackrel{?}{=} zero(f) \ \wedge \ X^i \stackrel{?}{=} t \ \wedge \ c \ , \ k) \ \vee \ d$$
$$\text{if} \begin{cases} \text{either } t \in V \text{ and } t \in Var(f(X)) \\ \text{or } root(t) \notin F_{AC1} \\ f \in F_{AC1} \\ f(X) \notin subt(c) \end{cases}$$

Collapse 2:

$$\exists \vec{z}.(f(X)\stackrel{?}{=}g(Y) \wedge c, k) \ \vee \ d \quad \Rightarrow$$
$$\bigvee_{i \in [1..|X|]} \exists \vec{z}.(\bigwedge_{j \in [1..|X|]; j \neq i} X^j \stackrel{?}{=} zero(f) \ \wedge \ X^i \stackrel{?}{=} g(Y) \ \wedge \ c \ , \ k)$$
$$\vee \ \bigvee_{i \in [1..|Y|]} \exists \vec{z}.(\bigwedge_{j \in [1..|Y|]; j \neq i} zero(f) \stackrel{?}{=} Y^j \ \wedge \ f(X) \stackrel{?}{=} Y^i \ \wedge \ c \ , \ k)$$
$$\vee \ d$$
$$\text{if} \begin{cases} f, g \in F_{AC1} \\ f \neq g \end{cases}$$

AC1 Decompose:

$$\exists \vec{z}.(f(X)\stackrel{?}{=}f(Y) \wedge c, k) \ \vee \ d \quad \Rightarrow$$
$$\bigvee_{(\exists \vec{z'}.(c', \top)) \in dio_{AC1}(\exists \vec{z}.(f(X) \stackrel{?}{=} f(Y), \top), W)} \exists \vec{z'}.(c' \wedge c, k) \ \vee \ d$$
$$\text{where } W = Var(\exists \vec{z}.(f(X)\stackrel{?}{=}f(Y) \wedge c, k))$$
$$\text{if} \begin{cases} f \in F_{AC1} \\ f(X) \neq_{AC1} f(Y) \\ f(X) \notin subt(c) \text{ or } f(Y) \notin subt(c) \end{cases}$$

Table 2: The transformation rules **AC1-Unify**

$root(f(X)\sigma)$. Either a proper subterm of $f(X)\sigma$ contains $x\sigma$ or $Flatten$ has applied and $|args(x\sigma)| < |args(f(X)\sigma)|$. Both lead to a contradiction. (If $f(Y) =_{AC1} f(Z)$, then $|Y| = |Z|$ and there exists a permutation π, such that $Y^i =_{AC1} Z^{\pi(i)}$, for all $i \in [1..|Y|]$.) Thus $Sol(x \stackrel{?}{=} f(X)) = \emptyset$; and for $C_2 = d$, we obtain $Sol(C_1) = Sol(C_2)$.

- **Collapse 1.** Let $C_1' = \exists\vec{z}.(f(X) \stackrel{?}{=} t \wedge c, k)$, where $f \in F_{AC1}$, and $t \in V$ or $root(t) \notin F_{AC1}$, $C_1 = C_1' \vee d$, $C_2' = \bigvee_{i\in[1..|X|]} \exists\vec{z}.(\bigwedge_{j\in[1..|X|];j\neq i} X^j \stackrel{?}{=} zero(f) \wedge X^i \stackrel{?}{=} t \;\wedge\; c\;,\; k)$, and $C_2 = C_2' \vee d$.

  We show that $Sol(C_1') = Sol(C_2')$.

  "$\subseteq$": If $\sigma \in Sol(f(X) \stackrel{?}{=} t)$ then $f(X)\sigma =_{AC1} t\sigma$. Either t is in V or $root(t) \notin F_{AC1}$. $Zero$ must apply to $f(X)\sigma$ $|X| - 1$ times at top-level, leaving an instance of a term X^i for some $i \in [1..|X|]$: $X^i\sigma =_{AC1} t$. In this case, $X^j\sigma =_{AC1} zero(f)$ for all $j \in [1..|X|], j \neq i$ and thus $\sigma \in Sol(\bigwedge_{j\in[1..|X|],j\neq i} X^j \stackrel{?}{=} zero(f) \wedge X^i \stackrel{?}{=} t)$. Since all $i \in [1..|X|]$ are covered in C_2', we conclude $Sol(C_1') \subseteq Sol(C_2')$.

  "$\supseteq$": If $\sigma \in Sol(\bigwedge_{j\in[1..|X|],j\neq i} X^j \stackrel{?}{=} zero(f) \wedge X^i \stackrel{?}{=} t)$, for some $i \in [1..|X|]$, then $Zero$ applies to $f(X)\sigma$ $|X| - 1$ times at top-level since $\sigma X^j =_{AC1} zero(f)$, for all $j \in [1..|X|], j \neq i$, leaving an instance of X^i.

- **Collapse 2.** similar to **Collapse 1**, but both the left and the right hand side can collapse.

- **AC1 Decompose.** similar to **AC Decompose**.

$\square$

Lemma 3.5 (Completeness) *Every normal form with respect to* **AC1-Unify** *is an* $AC1$*-unification problem in solved form.*

Proof. Let $C = \bigvee_{h \in [1..k]} \exists \vec{z_h}.(c_h, k_h)$ be a normal form with respect to **AC1-Unify**. Assume that for some $h \in [1..k]$ $c_h = r \stackrel{?}{=} s \wedge c'_h$, where $r = f(X)$ and $s = g(Y)$ are non-variable terms. If both $f \in F_{AC1}$ and $g \in F_{AC1}$, then **Collapse 2** applies if $f \neq g$ and **AC1 Decompose** or **Delete** applies otherwise. If only one of the symbols f and g are in F_{AC1}, then **Collapse 1** applies. If $f = g \notin F_{AC1}$ then **AC Decompose**, **Free Decompse** or **Delete** applies, and if $f, g \notin F_{AC1}$ and $f \neq g$, then **Fail** applies. Note that the restrictions in **Collapse 1**, **Free Decompose**, **AC Decompose**, and **AC1 Decompose** cause no deadlock. (In a set of terms, not all terms can be strictly contained in other terms.)

So, assume that $r = x \in V$. If $s = x$, **Delete** applies. If $s \neq x$, then **Check** applies if $x \in Var(s)$ and $root(s) \notin F_{AC1}$, **Collapse 1** applies if $x \in Var(s)$ and $root(s) \in F_{AC1}$, and **Eliminate** applies otherwise.

The validation of the solved form properties is similar to the proof of Lemma 3.2. □

Lemma 3.6 (Termination) *Every* $AC1$*-unification procedure in which, after every application of* **Collapse 1**, **Collapse 2**, **Free Decompose**, **AC Decompose** *and* **AC1 Decompose** *all newly introduced atomic problems with variables on at least one side are eliminated using* **Delete**, **Check** *and* **Eliminate** *terminates on every* $AC1$*-unification problem that contains only one* AC*-symbol.*

Proof. The limitation on the presence of only one AC-symbol enables us to use the same ordering as in the proof of Lemma 3.3 (see Definition 3.6). The proof for the rules that occur already in **AC-Unify** is similar to the proof of Lemma 3.3. The proof for the rule **AC1 Decompose** is similar to the proof for the rule **AC Decompose**. We discuss the remaining rules:

- **Collapse 1.** We prove that the application of **Collapse 1**, followed by the application of **Delete**, **Check**, and **Eliminate** on the newly introduced atomic problems with variables on at least one side, decreases the complexity of an $AC1$-unification problem.

  Let $e_1 = \exists\vec{z}.(f(X)\stackrel{?}{=}t \wedge c, k)$, where either $t \in V$ and $t \in Var(f(X))$ or $root(t) \notin F_{AC1}$, $f \in F_{AC1}$, and $f(X) \notin subt(c)$, and let $C_1 = e_1 \vee d$. Let $C_2' = \bigvee_{i\in[1..|X|]} \exists\vec{z}.(\bigwedge_{j\in[1..|X|];j\neq i} X^j\stackrel{?}{=}zero(f) \;\wedge\; X^i\stackrel{?}{=}t \;\wedge\; c\;,\; k) \;\vee\; d$, and $e_2' = \exists\vec{z}.(\bigwedge_{j\in[1..|X|];j\neq h} X^j\stackrel{?}{=}zero(f) \;\wedge\; X^h\stackrel{?}{=}t \;\wedge\; c\;,\; k)$,
  for some $h \in [1..|X|]$.

  1. If $X^h, t \in V$ and $X^h = t$ then **Delete** applies to $X^h\stackrel{?}{=}t$. $e_2 = \exists\vec{z}.(\bigwedge_{j\in[1..|X|];j\neq h;X^j\notin V} X^j\sigma\stackrel{?}{=}zero(f) \;\wedge\; c\sigma, k\sigma)$, where $\sigma = \{X^j \mapsto zero(f) \;\mid X^j \in V, j \neq h\}$. We obtain $\alpha(e_1) \geq \alpha(e_2)$ and $\beta(e_1) \geq \beta(e_2)$. If $f(X)$ does not occur in c, then $\gamma(e_1) > \gamma(e_2)$, and therefore $\mu_e(e_1) >_e \mu_e(e_2)$.

     Now, we assume that $f(X)$ occurs in c. We obtain $\gamma(e_1) \geq \gamma(e_2)$ and $\delta(e_1) \geq \delta(e_2)$. Since $f(X)\sigma$ occurs in $c\sigma$, the terms $X^j\sigma$ do not count for $\varepsilon(e_2)$. Furthermore, $f(X)$ counts for $\varepsilon(e_1)$, since $f(X) \notin subt(c)$ (see condition to apply **Collapse 1**). Clearly, this occurrence of $f(X)$ has no corresponding occurrence in e_2. Thus,

$\varepsilon(e_1) > \varepsilon(e_2)$, and therefore $\mu_e(e_1) >_e \mu_e(e_2)$.

2. If $X^h, t \in V$ and $X^h \neq t$ then **Eliminate** applies to $X^h \stackrel{?}{=} t$, resulting in $e_2 = \exists \vec{z}.(\bigwedge_{j \in [1..|X|]; j \neq h; X^j \notin V} X^j \sigma \stackrel{?}{=} zero(f) \ \wedge \ c\sigma, k\sigma)$, where $\sigma = \{X^j \mapsto zero(f) \ \mid X^j \in V, j \neq h\} \cup \{t \mapsto X^h\}$. We obtain $\alpha(e_1) \geq \alpha(e_2)$. The term t occurs under @ and, since $t \in Var(f(X))$, under at least one other symbol. Therefore, we obtain $\beta(e_1) \geq \beta(e_2)$. Clearly, $\gamma(e_1) \geq \gamma(e_2)$. The atomic $AC1$-unification problem $f(X) \stackrel{?}{=} t$ has no corresponding occurrence in e_2, thus $\delta(e_1) > \delta(e_2)$, and therefore $\mu_e(e_1) >_e \mu_e(e_2)$.

3. If $X^h \notin V$ and $t \in V$ then **Eliminate** applies to $X^h \stackrel{?}{=} t$ and e_2 as in 2. Proof as in 1.

4. If $X^h \in V$ and $t \notin V$ then **Eliminate** applies to $X^h \stackrel{?}{=} t$. $e_2 = \exists \vec{z}.(\bigwedge_{j \in [1..|X|]; j \neq h; X^j \notin V} X^j \sigma \stackrel{?}{=} zero(f) \ \wedge \ c\sigma, k\sigma)$, where $\sigma = \{X^j \mapsto zero(f) \ \mid X^j \in V, j \neq h\} \cup \{X^h \mapsto t\}$. Proof as in 1.

5. If X^h and $t \notin V$ then $e_2 = \exists \vec{z}.(\bigwedge_{j \in [1..|X|]; j \neq h; X^j \notin V} X^j \sigma \stackrel{?}{=} zero(f) \ \wedge \ X^h \sigma \stackrel{?}{=} t\sigma \ \wedge \ c\sigma, k\sigma)$, where $\sigma = \{X^j \mapsto zero(f) \ \mid X^j \in V, j \neq h\}$. Proof as in 1.

We proved that, after the exhaustive application of **Delete**, **Fail**, and **Eliminate**, every newly constructed existential constraint e_2 fulfills $\mu(e_1) >_e e_2$; thus the complexity of the $AC1$-unification problem decreases.

- **Collapse 2.** We prove that the application of **Collapse 2**, followed by the application of **Delete**, **Check**, and **Eliminate** on the newly introduced atomic problems with variables on at least one side, decreases the complexity of an $AC1$-unification problem.

Let $e_1 = \exists\vec{z}.(f(X)\stackrel{?}{=}g(Y) \wedge c, k)$, where $f, g \in F_{AC1}$, and $C_1 = e_1 \vee d$.
Let $C_2' =$
$\bigvee_{i\in[1..|X|]} \exists\vec{z}.(\bigwedge_{j\in[1..|X|];j\neq i} X^j\stackrel{?}{=}zero(f) \;\wedge\; X^i\stackrel{?}{=}g(Y) \;\wedge\; c\;,\; k) \;\vee$
$\bigvee_{i\in[1..|Y|]} \exists\vec{z}.(\bigwedge_{j\in[1..|Y|];j\neq i} zero(f)\stackrel{?}{=}Y^j \;\wedge\; f(X)\stackrel{?}{=}Y^i \;\wedge\; c\;,\; k) \;\vee\; d$, and $e_2' = \exists\vec{z}.(\bigwedge_{j\in[1..|X|];j\neq h} X^j\stackrel{?}{=}zero(f) \;\wedge\; X^h\stackrel{?}{=}g(Y) \;\wedge\; c\;,\; k)$, for some $h \in [1..|X|]$ (proof for e_2' in second disjunction similar).

1. If $X^h \in V$ then **Eliminate** applies to $X^h\stackrel{?}{=}g(Y)$, and $e_2 = \exists\vec{z}.(\bigwedge_{j\in[1..|X|];j\neq h;X^j\notin V} X^j\sigma\stackrel{?}{=}zero(f) \;\wedge\; c\sigma, k\sigma)$, where $\sigma = \{X^j \mapsto zero(f) \mid X^j \in V, j \neq h\} \cup \{X^h \mapsto g(Y)\}$. We obtain $\alpha(e_1) \geq \alpha(e_2)$. If $|OP(X^h, e)| \geq 2$, then $\beta(e_1) > \beta(e_2)$. Otherwise, we obtain $\beta(e_1) \geq \beta(e_2)$, and since X^h occurs only under the AC-symbol f, *Flatten* applies at every application of $\{X^h \mapsto g(Y)\}$, such that $\gamma(e_1) > \gamma(e_2)$.

2. If $X^h \notin V$ then $e_2 = \exists\vec{z}.(\bigwedge_{j\in[1..|X|];j\neq h;X^j\notin V} X^j\sigma\stackrel{?}{=}zero(f) \wedge X^h\sigma\stackrel{?}{=} g(Y)\sigma \;\wedge\; c\sigma, k\sigma)$, where $\sigma = \{X^j \mapsto zero(f) \mid X^j \in V, j \neq h\}$.

   For $\Gamma \in \{\alpha, \beta, \gamma, \delta, \varepsilon\}$, we obtain $\Gamma(e_1) \geq \Gamma(e_2)$, and additionally $\zeta(e_1) > \zeta(e_2)$, since all positions in e_2 have a corresponding position in e_1 (the terms $X^h\sigma$ have their corresponding position in $f(X)$), and the position $\langle\lambda, \emptyset\rangle$ in $f(X)$ has no corresponding position in e_2.

We proved that, after the exhaustive application of **Delete**, **Fail**, and **Eliminate**, every newly constructed existential constraint e_2 fulfills

$\mu(e_1) >_e e_2$; thus the complexity of the $AC1$-unification problem decreases.

□

3.4 AC1-Unification with Zero-Disequations

During the solving $AC1$-unification problems with zero-disequations, we transfer zero-disequations of the form $t \neq 0$ in the unsolved part to the solved part, in which only zero-disequations of the form $x \neq 0$, where $x \in V$, are allowed. We shall resolve zero-disequations similarly to atomic $AC1$-unification problems. In addition any substitution resulting from a solution of a unification problem must fulfill all zero-disequations.

All terms in the solved part are kept in normal form with respect to the rewrite system $Zero$. This ensures, together with the normal form computation with respect to $Flatten$, that if a term $t =_{AC1} 0$ for some $0 \in F_{zero}$ then $t = 0$, so that violations of zero-disequations can easily be detected.

Notation: We extend the function $\downarrow_{Zero}$ from terms to conjunctive constraints. Let

$$c = \bigwedge_{i\in[1..n]} r_i \stackrel{?}{=} s_i \;\wedge \bigwedge_{i\in[1..n]} x_i \neq f_i$$

be a conjunctive constraint. Then

$$c\downarrow_{Zero} = \bigwedge_{i\in[1..n]} r_i\downarrow_{Zero} \stackrel{?}{=} s_i\downarrow_{Zero} \;\wedge \bigwedge_{i\in[1..n]} x_i \neq f_i.$$

The set **Disequation-AC1-Unify** of transformation rules consists of the rules in **AC1-Unify**, where the rule **Eliminate** is replaced by the rules **Eliminate Pass** and **Eliminate Clash**, and the new rules **Disequation Delete**, **Disequation Eliminate Pass**, **Disequation Eliminate Clash**, and **Disequation Decompose** in Table 3. The union of the relations described by the rules in **Disequation-AC1-Unify** is denoted by $\Rightarrow_{\mathbf{Disequation-AC1-Unify}}$.

Definition 3.8 A *disequation-AC1-unification procedure* is a subset U of the relation $\Rightarrow_{\mathbf{Disequation-AC1-Unify}}$ such that every normal form with respect to U is a normal form with respect to $\Rightarrow_{\mathbf{Disequation-AC1-Unify}}$.

Similar to the previous sections, we state the soundness, completeness, and termination of disequation-$AC1$-unification procedures. We give only the parts of the proofs that differ from the corresponding proof in the previous section.

Lemma 3.7 (Soundness) *Every disequation-$AC1$-unification procedure preserves the unifiers: If $C_1 \Rightarrow^*_{\mathbf{Disequation-AC1-Unify}} C_2$, then $Sol(C_1) = Sol(C_2)$.*

Proof.

- **Eliminate Pass.** similar to **Eliminate** in the previous section. (Clearly, the application of $\downarrow_{Zero}$ has no impact on the set of solutions of a constraint.)

- **Eliminate Clash.** Let $C_1 = \exists \vec{z}.(x \stackrel{?}{=} t \wedge c, k) \ \vee \ d$. Clearly, $Sol(x \stackrel{?}{=} f \wedge x \neq f) = \emptyset$. Thus if for some atomic $AC1$-unification problem $x \stackrel{?}{=} f$ in $(x \stackrel{?}{=} t \wedge k\sigma) \downarrow_{Zero}$ the zero-disequation $x \neq f$ occurs in k, then $Sol(C_1) = Sol(d)$.

Eliminate Pass:

$$\exists\vec{z}.(x\stackrel{?}{=}t \wedge c, k) \;\vee\; d \;\Rightarrow\; \exists\vec{z}.(\; c\sigma \wedge g\sigma, (x\stackrel{?}{=}t \wedge k\sigma)\downarrow_{Zero}) \;\vee\; d$$

$$\text{where} \begin{cases} \sigma = \{x \mapsto t\} \\ k = g \wedge k' \\ g = \bigwedge_{s \neq 0 \in k, x \in Var(s)} s \neq 0 \end{cases}$$

$$\text{if} \begin{cases} x \in V \\ x \notin Var(t) \\ \text{for no } x\stackrel{?}{=}f \text{ in } (x\stackrel{?}{=}t \wedge k\sigma)\downarrow_{Zero} \\ x \neq f \text{ occurs in } k \end{cases}$$

Eliminate Clash:

$$\exists\vec{z}.(x\stackrel{?}{=}t \wedge c, k) \;\vee\; d \;\Rightarrow d$$

$$\text{if} \begin{cases} x \in V \\ x \notin Var(t) \\ \text{for some } x\stackrel{?}{=}f \text{ in } (x\stackrel{?}{=}t \wedge k\sigma)\downarrow_{Zero} \\ x \neq f \text{ occurs in } k \end{cases}$$

$$\text{where } \sigma = \{x \mapsto t\}$$

Disequation Delete:

$$\exists\vec{z}.(f(X) \neq 0 \wedge c, k) \;\vee\; d \;\Rightarrow\; \exists\vec{z}.(c, k) \;\vee\; d$$

$$\text{if } f \notin F_{AC1}$$

Disequation Eliminate Pass:

$$\exists\vec{z}.(x \neq 0 \wedge c, k) \;\vee\; d \;\Rightarrow\; \exists\vec{z}.(c, x \neq 0 \wedge k) \;\vee\; d$$

$$\text{if} \begin{cases} x \in V \\ x\stackrel{?}{=}0 \notin k \end{cases}$$

Disequation Eliminate Clash:

$$\exists\vec{z}.(x \neq 0 \wedge c, k) \;\vee\; d \;\Rightarrow\; d$$

$$\text{if} \begin{cases} x \in V \\ x\stackrel{?}{=}0 \in k \end{cases}$$

Disequation Decompose:

$$\exists\vec{z}.(f(X) \neq 0 \wedge c, k) \;\vee\; d \;\Rightarrow\; \bigvee_{g \in DNF} \exists\vec{z}.(g \wedge c, k) \;\vee\; d$$

where DNF is the set of conjunctions in the disjunctive normal form of

$$\begin{array}{ll} (X^1 \neq 0 \vee X^2 \neq zero(f) \vee X^3 \neq zero(f) \vee \cdots \vee X^{|X|} \neq zero(f)) & \wedge \\ (X^1 \neq zero(f) \vee X^2 \neq 0 \vee X^3 \neq zero(f) \vee \cdots \vee X^{|X|} \neq zero(f)) & \wedge \\ \vdots & \vdots \\ (X^1 \neq zero(f) \vee X^2 \neq zero(f) \vee X^3 \neq zero(f) \vee \cdots \vee X^{|X|} \neq 0) & \end{array}$$

Table 3: The transformation rules **Disequation-AC1-Unify**

- **Disequation Delete.** Let $C_1 = \exists\vec{z}.(f(X) \neq 0 \wedge c, k) \ \vee \ d$, where $f \notin F_{AC1}$. $AC1$ can only change the root symbols of terms with $root(t) \in F_{AC1}$. Thus any substitution is in $Sol(f(X) \neq 0)$ and $Sol(C_1) = Sol(\exists\vec{z}.(c, k) \ \vee \ d)$.

- **Disequation Eliminate Pass.** trivial.

- **Disequation Eliminate Clash.** similar to **Eliminate Clash**.

- **Disequation Decompose.** Let $C_1 = \exists\vec{z}.(f(X) \neq 0 \wedge c, k) \ \vee \ d$, where $f \in F_{AC1}$. The zero-disequation $f(X) \neq 0$ is violated, if and only if for one term X^i in X holds $X^i =_{AC1} 0$ and for all other terms X^j in X holds $X^j =_{AC1} zero(f)$. The negation of this fact is expressed by the formula in **Disequation Decompose**, of which DNF is a disjunctive normal form. Distributivity yields that $Sol(C_1) = Sol(\bigvee_{g \in DNF} \exists\vec{z}.(g \wedge c, k) \vee d)$.

□

Lemma 3.8 (Completeness) *Every normal form with respect to* **Disequation-AC1-Unify** *is an* $AC1$*-unification problem with zero-disequations in solved form.*

Proof. Let $C = \bigvee_{h \in [1..k]} \exists\vec{z}.(c_h, k_h)$ be a normal form with respect to **Disequation-AC1-Unify**. From the argumentation in the proof of Lemma 3.5 follows that c_h is a conjunction of zero-disequations for all $h \in [1..k]$. Assume $t \neq 0 \in c_h$. If $t \in V$ then **Disequation Eliminate Pass** or **Disequation Eliminate Clash** applies. If $t = f(X)$ then **Disequation Decompose** or **Disequation Delete** applies. Therefore, c_h must be $\top$ for all $h \in [1..k]$.

The proof of uniqueness and idempotency follows from the argumentation in the proof of Lemma 3.2.

We show that in every existential constraint in C, every replacement in the solved part fulfills all zero-disequations in the solved part.

Assume that the atomic $AC1$-unification problem $x \stackrel{?}{=} 0$ in k_h violates the zero-disequation $x \neq 0$ in k_h. The atomic $AC1$-unification problem $x \stackrel{?}{=} 0$ cannot have been added to k_h after the zero-disequation $x \neq 0$, since the conditions to apply **Eliminate Pass** would not have been fulfilled. On the other hand, the zero-disequation $x \neq 0$ cannot have been added to k_h after the atomic $AC1$-unification problem $x \stackrel{?}{=} 0$, since the conditions to apply **Disequation Eliminate Pass** would not have been fulfilled, which is a contradiction. □

The simplest way to ensure the termination of disequation-AC1-unification procedures is to solve first the atomic unification problems and then the zero-disequations.

Lemma 3.9 (Termination) *Every disequation-$AC1$-unification procedure that first applies the rules in* **AC1-Unify** *(including* **Eliminate Pass** *and* **Eliminate Clash***) as described in Lemma 3.6 and then the rules* **Disequation Delete**, **Disequation Eliminate Pass**, **Disequation Eliminate Clash**, *and* **Disequation Decompose** *in any order, terminates on every $AC1$-unification problem with zero-disequations that contains only one AC-symbol.*

Proof. The former process terminates due to Lemma 3.6. The termination of the latter process can be shown using an ordering based on the size of the terms in the zero-disequations of the unsolved parts of the existential constraints. □

Note that violations of zero-disequations can be detected earlier, when the solving of atomic $AC1$-unification problems and the solving of zero-disequations are interleaved. However, in this case the termination proof is harder because in **Eliminate Pass** zero-disequations in the solved part are moved back to the unsolved part.

Chapter 4

Completion

In this chapter, we shall approach the task of completion of equation sets in the presence of associative commutative function symbols and identities. Prior to describing procedures for completion in Section 4.4, we shall present a notion of rewriting in $AC1$-theories in Section 4.1 that is reasonably efficient and—for our purpose—equivalent to the concept of rewriting modulo $AC1$. In Section 4.2, we introduce some operations on constraints that we shall use in Section 4.3 to interleave constraint solving with the process of completion. In Section 4.4, a variety of completion procedures is described using a set of transformation rules. Suitable application strategies for these rules are discussed in Section 4.5.

4.1 Constrained AC1-rewriting

Definition 4.1 A *constrained equation* is a triple $C \mid l \approx r$, where C is a constraint, and l and r are terms.

If E is a set of constrained equations, we write $s \rightarrow_E t$ to indicate that there exist terms w and l', a position π in w, a substitution σ, and a constrained equation $C \mid l \approx r$ in E, such that $\sigma \in Sol(C)$, $s = w[l']_\pi$, $l' =_{AC1} l\sigma$, and $t = w[r\sigma]_\pi$. The relation $\rightarrow_E$ is called the rewrite relation induced by E. We say that s *rewrites* to t by E if $s \rightarrow_E t$. A normal form with respect to $\rightarrow_E$ is said to be *irreducible* by E.

A set E of equations will be called a *constrained rewrite system* if the rewrite relation $\rightarrow_E$ is the primary object of study. The constrained equations of a constrained rewrite system are also called *constrained rewrite rules.*

Several different ways have been devised to integrate equational theories into the rewriting process. The most general approach, applied to the $AC1$-case, is *rewriting modulo AC1.*

Definition 4.2 We say that the constrained rewrite system R *rewrites* the term v to the term w *modulo AC1*, denoted $v \xrightarrow[R/AC1]{\sigma} w$, if there exist terms v', w' such that $v =_{AC1} v' \rightarrow_R w' =_{AC1} w$.

We write $v \xrightarrow[\mathcal{E}/AC1]{\sigma} w$ to specify the constrained rewrite rule and the substitution by which v' is rewritten to w'.

Definition 4.3 The constrained rewrite system R is called *convergent modulo AC1* if $\xrightarrow[R/AC1]{}$ is terminating and Church-Rosser modulo $AC1$.

The task of *completion modulo AC1* of a given set E of constrained equations is to construct a constrained rewrite system R such that $AC1 \cup E$ and $AC1 \cup R$ define the same equational theory and R is convergent modulo $AC1$.

Several weaker notions of rewriting in an equational theory are in use (for the general case, see [Bac 91]; for the $AC1$-case, see [JM 90]). We shall employ our extended concept of subterms and positions to define yet another version of rewriting in the theory $AC1$.

Definition 4.4 We say that the constrained rewrite system R *AC1-rewrites* the term v to the term w, denoted $v \xrightarrow[AC1\backslash R]{} w$, if $v \downarrow_{Flatten \cup Zero} \rightarrow_R w$. We write $v \xrightarrow[AC1\backslash \mathcal{E}]{\sigma} w$ to specify the constrained rewrite rule and the substitution by which $v \downarrow_{Flatten \cup Zero}$ is rewritten to w.

Proposition 4.1 *If $r =_{AC1} s \xrightarrow[R/AC1]{} t$ then there exists a term u such that $r \xrightarrow[AC1\backslash R]{} u =_{AC1} t$.*

Proof. Let $r =_{AC1} s \xrightarrow[R/AC1]{} t$. The definition of $\xrightarrow[R/AC1]{}$ implies that there exist terms s' and t' such that $s =_{AC1} s' \rightarrow_R t' =_{AC1} t$. Let r' be the normal form of r with respect to $Flatten \cup Zero$. Let π be the position at which $\rightarrow_R$ applies to s'. There exists a term w, a substitution σ, and a constrained rewrite rule $\mathcal{E} = C \mid l \approx r$ in R, such that $\sigma \in Sol(C)$, $s' = w[l\sigma]_\pi$ and $t' = w[r\sigma]_\pi$. From Lemma 2.1 follows that there exists a position π' in r' such that $r'|_{\pi'} =_{AC1} s'|_\pi$. Therefore, the constrained rewrite rule $\mathcal{E}$ applies to r' at position π' resulting in a term u. Since $r' =_{AC1} s'$ and $r'|_{\pi'} =_{AC1} s'|_\pi$, we get $u =_{AC1} t'$ and thus $u =_{AC1} t$. Figure 1 depicts the situation. $\square$

We can say that $AC1$-rewriting rewrites equivalence classes with respect to $AC1$ and is equivalent to rewriting modulo $AC1$, when we are only interested in one representantive of the equivalence class of terms to which a term is rewritten. In particular, we obtain the following results.

Proposition 4.2 *(1) The relation $\xrightarrow[AC1\backslash R]{}$ terminates if and only if $\xrightarrow[R/AC1]{}$ terminates; (2) the relation $\xrightarrow[AC1\backslash R]{}$ is Church-Rosser modulo $AC1$ if and only if $\xrightarrow[R/AC1]{}$ is Church-Rosser modulo $AC1$.*

$$\begin{tikzcd}
r' \arrow[dr, "R"'] & r \arrow[l, "!"', "Flatten \cup Zero"] \arrow[r, equal, "AC1"'] \arrow[d, "AC1\backslash R"] & s \arrow[r, equal, "AC1"'] \arrow[d, "R/AC1"] & s' \arrow[d, "R"] \\
& u \arrow[r, equal, "AC1"'] & t \arrow[r, equal, "AC1"'] & t'
\end{tikzcd}$$

Figure 1: Illustration to Proposition 4.1

Proof. (sketch) Using Proposition 4.1, we can show that if $\xrightarrow[R/AC1]{}$ does not terminate then neither does $\xrightarrow[AC1\backslash R]{}$ (see Figure 2). The reverse direction is trivial.

We can show by induction using Proposition 4.1 that if $\xrightarrow[R/AC1]{}$ is Church-Rosser modulo $AC1$ then so is $\xrightarrow[AC1\backslash R]{}$ (see Figure 3). The reverse direction is trivial. □

Now, we can rephrase a well-known fact (see [Huet 80]) using $AC1$-rewriting instead of rewriting modulo $AC1$ in a lemma that shall be the foundation of our completion procedures.

Lemma 4.1 *Let R be constrained rewrite system such that $\xrightarrow[AC1\backslash R]{}$ terminates. The relation $\xrightarrow[R/AC1]{}$ is Church-Rosser modulo $AC1$ if and only if $\xrightarrow[AC1\backslash R]{}$ is locally confluent modulo $AC1$.*

Remark: $AC1$-rewriting will be reasonably efficient, when all terms, including the left and right hand side of constrained rewrite rules are kept in normal

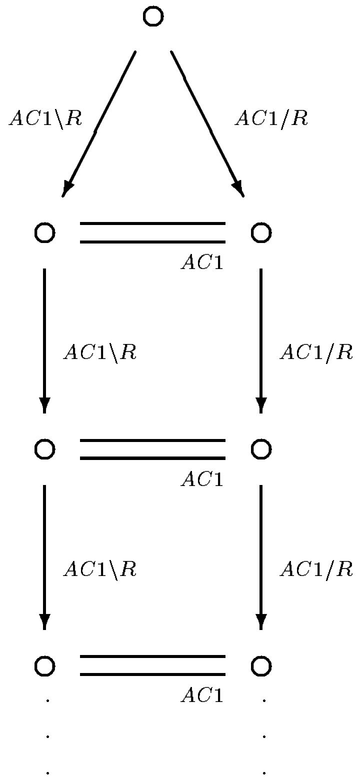

Figure 2: Illustration to Proposition 4.2 (Termination)

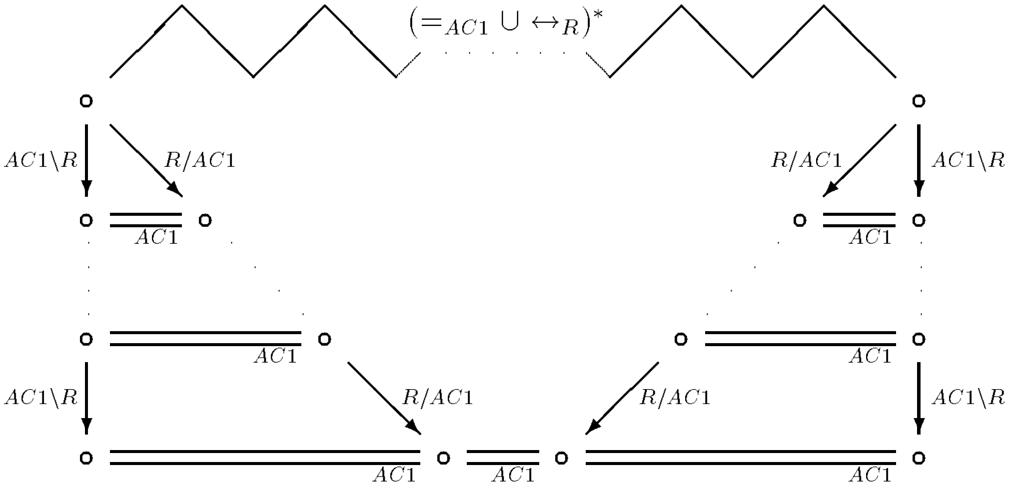

Figure 3: Illustration to Proposition 4.2 (Church-Rosser)

form with respect to $Flatten \cup Zero$. In this case, $AC1$-matching procedures as described in Appendix A can be applied.

During the process of completion, local confluence modulo $AC1$ is tested whenever a new constrained rewrite rule is found. On failure of this test, "critical pairs" are added. In rewriting modulo an equational theory, "extensions" of rewrite rules are often used to test for local confluence. The concepts of critical pairs and extensions are captured by the following lemma that implies a criterion for local confluence modulo $AC1$.

Lemma 4.2 (Critical Pair Lemma) *Let $\mathcal{E}_1 = C_1 \mid l \approx r$ and $\mathcal{E}_2 = C_2 \mid g \approx d$ be constrained rewrite rules that have no variables in common. There exist terms t, v and w such that $v \xleftarrow[AC1\backslash\mathcal{E}_1]{} t \xrightarrow[AC1\backslash\mathcal{E}_2]{} w$ if and only if*

- *there exist terms v' and w' such that $v \xrightarrow[AC1\backslash\mathcal{E}_2]{} v' =_{AC1} w' \xleftarrow[AC1\backslash\mathcal{E}_1]{} w$, or*
- *there exists a subterm l' in l and a substitution $\sigma \in Sol(C_1) \cap Sol(C_2)$ such that $l'\sigma =_{AC1} g\sigma$, or*
- *there exists a subterm g' in g and a substitution $\sigma \in Sol(C_1) \cap Sol(C_2)$ such that $g'\sigma =_{AC1} l\sigma$, or*
- *there exists a subterm l'' in $l+x$ and a substitution $\sigma \in Sol(C_1) \cap Sol(C_2)$ such that $l''\sigma =_{AC1} g\sigma$, where $root(l) = + \in F_{AC}$ and x is a new variable, or*
- *there exists a subterm g'' in $g+x$ and a substitution $\sigma \in Sol(C_1) \cap Sol(C_2)$ such that $g''\sigma =_{AC1} l\sigma$, where $root(g) = + \in F_{AC}$ and x is a new variable.*

4.2 Some Operations on Constraints

Let $C_1 = \bigvee_{i\in[1..n]} \exists \vec{z_i}.(c_i, k_i)$ and $C_2 = \bigvee_{j\in[1..m]} \exists \vec{z_j}'.(c'_j, k'_j)$ be disjunctive constraints and let $DNF = \bigvee_{i\in[1..n']} \exists \vec{z_i}'.g_i$ be a disjunctive normal form of the formula $\bigvee_{i\in[1..n]} \exists \vec{z_i}.c_i \wedge k_i \;\; \wedge \;\; \bigvee_{j\in[1..m]} \exists \vec{z_j}'.c'_j \wedge k'_j$. Then $C_1 \wedge C_2$ denotes the disjunctive constraint $\bigvee_{i\in[1..n']} \exists \vec{z_i}'.(g_i, \top)$.

Proposition 4.3 *For any two constraints C_1 and C_2 holds $Sol(C_1 \wedge C_2) = Sol(C_1) \cap Sol(C_2)$.*

The application of a substitution σ to a constraint $C = \bigvee_{i\in[1..n]} \exists \vec{z_i}.(c_i, k_i)$ is defined to be the constraint $\bigvee_{i\in[1..n]}(c_i\sigma \wedge k_i\sigma, \top)$.

Proposition 4.4 *If $Ran(\sigma) \cap Var(C) = \emptyset$, $Dom(\sigma) \cap \bigcup_{i\,\in\,[1..n]} \vec{z_i} = \emptyset$ and $\sigma \in Sol(C)$, then $Sol(C\sigma) = \{\tau\sigma \mid \tau \in Sol(C)\}$.*

Let $C = \bigvee_{i\in[1..n]}(\bigwedge_{j\in[1..m_i]} t_{i,j} \neq 0_{i,j}, \bigwedge_{j'\in[1..m'_i]} t'_{i,j'} \neq 0'_{i,j'})$ be a disjunctive constraint that contains no atomic unification problem and let $DNF = \bigvee_{i\in[1..k]} \bigwedge_{j\in[1..l_i]} s_{i,j} \stackrel{?}{=} 1_{i,j} \;\; \wedge \;\; \bigwedge_{j'\in[1..l'_i]} s'_{i,j'} \stackrel{?}{=} 1'_{i,j'}$ be a disjunctive normal form of the formula $\bigwedge_{i\in[1..n]} \bigvee_{j\in[1..m_i]} t_{i,j} \stackrel{?}{=} 0_{i,j} \;\; \vee \;\; \bigvee_{j'\in[1..m'_i]} t'_{i,j'} \stackrel{?}{=} 0'_{i,j'}$. Then the negation of C, denoted $\overline{C}$ is defined to be the constraint $\bigvee_{i\in[1..k]}(\bigwedge_{j\in[1..l_i]} s_{i,j} \stackrel{?}{=} 1_{i,j} \;\; \wedge \;\; \bigwedge_{j'\in[1..l'_i]} s'_{i,j'} \stackrel{?}{=} 1'_{i,j'}, \top)$.

Proposition 4.5 *If C is a disjunctive constraint that contains no atomic unification problem, then $\Sigma(V(C)) = Sol(C) \uplus Sol(\overline{C})$.*

4.3 Auxiliary Transformations

$AC1$-completion can be viewed as a transformation process on pairs consisting of a set of unoriented constrained equations on the left hand side and an constrained rewrite system on the right hand side.

In order to interleave constraint solving with completion, we introduce the set **Aux** of transformation rules in Table 4. Each transformation rule in **Aux** represents a binary relation on pairs of sets of constrained equations. We denote the union of these relations by $\Rightarrow_{\mathbf{Aux}}$.

Proposition 4.6 (Soundness) *If* $E; N \Rightarrow_{\mathbf{Aux}} E'; N'$ *then the relations* $\xrightarrow[AC1\backslash E \cup N]{}$ *and* $\xrightarrow[AC1\backslash E' \cup N']{}$ *are equal.*

Note that in **Equation Distribute** and **Rule Distribute** the constraint C may be $\perp$. In this case, the rule or equation is discarded.

4.4 The Transformation Rules AC1-Complete

Table 5 shows the set $\mathbf{AC1-Complete}$ of transformation rules, whose goal is to generate from a given set of constrained equations a constrained rewrite system that is convergent modulo $AC1$. Each transformation rule describes a relation on pairs of sets of constrained equations. The union of these relations is denoted by $\Rightarrow_{\mathbf{AC1-Complete}}$.

Equation Unify:

$$E \cup \{C \mid l \approx r\} \ ; \ N \quad \Rightarrow \quad E \cup \{C' \mid l \approx r\} \ ; \ N$$
$$\text{if } C \Rightarrow_{\mathbf{Disequation-AC1-Unify}} C'$$

Rule Unify:

$$E \ ; \ N \cup \{C \mid l \approx r\} \quad \Rightarrow \quad E \ ; \ N \cup \{C' \mid l \approx r\}$$
$$\text{if } C \Rightarrow_{\mathbf{Disequation-AC1-Unify}} C'$$

Equation Distribute:

$$E \cup \{\textstyle\bigvee_{h\in[1..k]} e_h \mid l \approx r\} \ ; \ N \quad \Rightarrow \quad E \cup \textstyle\bigcup_{h\in[1..k]}\{e_h \mid l \approx r\} \ ; \ N$$

Rule Distribute:

$$E \ ; \ N \cup \{\textstyle\bigvee_{h\in[1..k]} e_h \mid l \approx r\} \quad \Rightarrow \quad E \ ; \ N \cup \textstyle\bigcup_{h\in[1..k]}\{e_h \mid l \approx r\}$$

Equation Apply:

$$E \cup \{\exists\vec{z}.(\top, \textstyle\bigwedge_{i\in[1..n]} x_i \neq 0_i \wedge \textstyle\bigwedge_{j\in[1..m]} y_j \stackrel{?}{=} t_j) \mid l \approx r\} \ ; \ N$$
$$\Rightarrow \quad E \cup \{\textstyle\bigwedge_{i\in[1..n]} x_i \neq 0_i \mid l\sigma \approx r\sigma\} \ ; \ N$$
$$\text{where } \sigma = \textstyle\bigwedge_{j\in[1..m]} y_j \stackrel{?}{=} t_j$$
$$\text{if } \exists\vec{z}.(\top, \textstyle\bigwedge_{i\in[1..n]} x_i \neq 0_i \wedge \textstyle\bigwedge_{j\in[1..m]} y_j \stackrel{?}{=} t_j) \text{ in solved form}$$

Rule Apply:

$$E \ ; \ N \cup \{\exists\vec{z}.(\top, \textstyle\bigwedge_{i\in[1..n]} x_i \neq 0_i \wedge \textstyle\bigwedge_{j\in[1..m]} y_j \stackrel{?}{=} t_j) \mid l \approx r\}$$
$$\Rightarrow \quad E \ ; \ N \cup \{\textstyle\bigwedge_{i\in[1..n]} x_i \neq 0_i \mid l\sigma \approx r\sigma\}$$
$$\text{where } \sigma = \textstyle\bigwedge_{j\in[1..m]} y_j \stackrel{?}{=} t_j$$
$$\text{if } \exists\vec{z}.(\top, \textstyle\bigwedge_{i\in[1..n]} x_i \neq 0_i \wedge \textstyle\bigwedge_{j\in[1..m]} y_j \stackrel{?}{=} t_j) \text{ in solved form}$$

Table 4: The transformation rules **Aux**

Remarks:

- We assume given a predicate $\mathcal{T}$ such that if $\mathcal{T}\mathcal{E}$ for all rules $\mathcal{E}$ in R then $\xrightarrow[AC1\backslash R]{}$ terminates (see Section 4.5 for further discussion of termination).

- The symbol $\simeq$ is used to denote unoriented equations in a commutative way: $E \cup \{C \mid l \simeq r\}$ denotes that either $C \mid l \approx r$ or $C \mid r \approx l$ occurs in a set of constrained equations and the remainder of this set is denoted by E.

- The symbol $\succ$ in **Collapse** denotes the strict part of the *encompassment ordering*, which is defined as follows: $s \succeq t$ if some subterm of s is an instance of t, but not vice versa.

Definition 4.5 An $AC1$-completion procedure is a subset of the relation $\mathbf{Aux} \cup \mathbf{AC1} - \mathbf{Complete}$.

Lemma 4.3 (Soundness) *If $E; N \Rightarrow_{\mathbf{AC1-Complete}} E'; N'$ then the relations $\xleftrightarrow[AC1\backslash E \cup N]{}{}^{*}$ and $\xleftrightarrow[AC1\backslash E' \cup N']{}{}^{*}$ are equal.*

We describe the fairness of a derivation in $\mathbf{Aux} \cup \mathbf{AC1} - \mathbf{Complete}$. Let $(CP(N); N) = \Rightarrow^{!}_{\mathbf{Deduce}} (\emptyset; N)$ be the pair of sets of constrained equations resulting from exhaustively applying the rule **Deduce** to the constrained rewrite system N. $CP(N)$ is called the set of critical equations of N. Let $(EXT(N); N) = \Rightarrow^{!}_{\mathbf{DeduceExtended}} (\emptyset; N)$ be the pair of sets of constrained equations resulting from exhaustively applying the rule **Deduce Extend** to

Deduce:

$$E \; ; \; N \;\Rightarrow\; E \cup \{C_1 \wedge C_2 \wedge l|_\pi \stackrel{?}{=} g \mid l[d]_\pi \approx r\} \; ; \; N$$
$$\text{if} \begin{cases} C_1 \mid l \approx r \text{ and } C_2 \mid g \approx d \;\in\; N \\ \pi \in Fpos(l) \end{cases}$$

Deduce Extended:

$$E \; ; \; N \;\Rightarrow$$
$$E \cup \{C_1 \wedge C_2 \wedge (l+x)|_\pi \stackrel{?}{=} g \mid (l+x)[d]_\pi \approx r\} \; ; \; N$$
$$\text{if} \begin{cases} C_1 \mid l \approx r \text{ and } C_2 \mid g \approx d \;\in\; N \\ \pi \in Fpos(l+x) \end{cases}$$

Orient:

$$E \cup \{C \mid l \simeq r\} \; ; \; N \;\Rightarrow\; E \; ; \; N \cup \{C \mid l \approx r\}$$
$$\text{if } \mathcal{T}(C \mid l \approx r)$$

Simplify:

$$E \cup \{C_1 \mid l \simeq r\} \; ; \; N \;\Rightarrow$$
$$E \cup \{C_1 \wedge C_2\theta \mid l' \approx r\} \cup \{C_1 \wedge \overline{C_2}\theta \mid l \approx r\} \; ; \; N$$
$$\text{if} \begin{cases} C_2 \mid g \approx d \;\in\; N \\ l \xrightarrow[AC1\backslash C_2 \mid g \approx d]{\theta} l' \end{cases}$$

Delete:

$$E \cup \{C \mid l \approx r\} \; ; \; N \;\Rightarrow\; E \; ; \; N$$
$$\text{if } l =_{AC1} r$$

Compose:

$$E \; ; \; N \cup \{C_1 \mid l \approx r\} \;\Rightarrow$$
$$E \; ; \; N \cup \{C_1 \wedge C_2\theta \mid l \approx r'\} \cup \{C_1 \wedge \overline{C_2}\theta \mid l \approx r\}$$
$$\text{if} \begin{cases} C_2 \mid g \approx d \;\in\; N \\ r \xrightarrow[AC1\backslash C_2 \mid g \approx d]{\theta} r' \end{cases}$$

Collapse:

$$E \; ; \; N \cup \{C_1 \mid l \approx r\} \;\Rightarrow$$
$$E \cup \{C_1 \wedge C_2\theta \mid l' \approx r\} \; ; \; N \cup \{C_1 \wedge \overline{C_2}\theta \mid l \approx r\}$$
$$\text{if} \begin{cases} C_2 \mid g \approx d \;\in\; N \\ l \xrightarrow[AC1\backslash C_2 \mid g \approx d]{\theta} l' \\ l \succ g \end{cases}$$

Table 5: The transformation rules **AC1-Complete**

the constrained rewrite system N. $CP(N)$ is called the set of extensions of N. Let $E_0; N_0 \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}} E_1; N_1 \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}} \cdots$ be a derivation in $\mathbf{Aux} \cup \mathbf{AC1-Complete}$. The set of constrained equations $\bigcup_{n=0}^{\infty}(\bigcap_{i=n}^{\infty} N_i)$ is denoted by N_∞.

A derivation $E_0; N_0 \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}} E_1; N_1 \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}} \cdots$ is said to be fair, if

$$\bigcup_{\mathcal{E}\in CP(N_\infty)\cup EXT(N_\infty)} (\xrightarrow[AC1\backslash\mathcal{E}]{}) \subseteq \bigcup_{i=0}^{\infty}(\xrightarrow[AC1\backslash E_i]{}).$$

In a fair derivation $E_0; N_0 \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}} E_1; N_1 \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}}$ ... all critical equations and critical extensions of N_∞ are instances of equations in the union of all E_i in the sense that whenever $r \xrightarrow[AC1\backslash\mathcal{E}]{} s$ for two terms r and s and a constrained equation $\mathcal{E} \in CP(N_\infty) \cup EXT(N_\infty)$ there exists an $i \in \mathbf{N}$ and a constrained equation $\mathcal{E}' \in E_i$ such that $r \xrightarrow[AC1\backslash\mathcal{E}']{} s$. An $AC1$-completion procedure is fair, if all derivations it produces are fair.

Lemma 4.4 (Completeness) *Let the derivation* $E_0; N_0 \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}} E_1; N_1 \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}} \cdots \Rightarrow_{\mathbf{Aux}\cup\mathbf{AC1-Complete}} \emptyset; N_k$ *be fair. For all terms v and w holds: If* $v \xleftrightarrow[AC1\backslash E_0 \cup N_0]{} w$ *then there exist terms u and u' such that* $v \xrightarrow[AC1\backslash N_k]{}{}^{*} u =_{AC1} u' \xleftarrow[AC1\backslash N_k]{}{}^{*} w$.

Proof. (sketch) As in [JM 90], the completeness can be shown using proof orderings, a technique developed in [BD 89]. A rewrite system on proofs, based on the transformation rules in **AC1-Complete**, must be defined and the correctness and termination of this rewrite system must be established. □

Equation Split:
$E \cup \{C \mid l \simeq r\}\ ;\ N \quad \Rightarrow \quad E \cup \{C \wedge C' \mid l \simeq r\} \cup \{C \wedge \overline{C'} \mid l \simeq r\}\ ;\ N$

Table 6: The transformation rule **Equation Split**

4.5 Application Strategies

The transformation rules **AC1-Complete** describe a large class of procedures, not all of which are suitable for completion. An obvious limitation is imposed by the negation of constraints in the rules **Simplify**, **Compose** and **Collapse**. Since negation of constraints is only defined on constraints that contain no atomic unification problems, we require that the rules in N are always in normal form with respect to **Aux**.

The difficulties in applying the rule **Orient** are to find an appropriate predicate $\mathcal{T}$ and to transform a given constrained equation $C \mid l \approx r$ such that $\mathcal{T}$ is fulfilled. Both problems are solved in [JM 90], where a criterion for termination of constrained rewrite systems is developed and an algorithm to generate a constraint for a constrained equation such that the criterion is fulfilled is presented.

Let $\mathcal{E} = C \mid l \approx r$ be an $AC1$-equation in E and C' a constraint such that $\mathcal{T}(C \wedge C' \mid l \approx r)$. We can split $\mathcal{E}$ into two parts: $\mathcal{E}' = C \wedge C' \mid l \approx r$ and $\mathcal{E}'' = C \wedge \overline{C'} \mid l \approx r$. Now, we can apply **Orient** to $\mathcal{E}'$. $\mathcal{E}''$ remains in E. We can schematize this operation by the transformation rule **Equation Split** shown in Table 6, whose soundness follows from Propositions 4.3 and 4.5. Regarding the application of the transformation rules in **Aux** to the unoriented equations, we shall discuss two strategies.

The first strategy is to apply **Aux** eagerly, transforming all occurring constraints into solved constraints consisting of conjunctions of zero-disequations. This leads to a total separation of completion and unification as in [JM 90], but maintaining the possibility of constraining unoriented equations.

The second strategy consists in postponing the solving of particularly hard unification problems and "shelving" the unoriented equations in which they occur. Proceeding with the completion process, the rule **Simplify** may apply to the postponed unification problem and make it easier to solve. In applying the rule **Orient**, we may solve unification problems in unoriented equations (lazily) only when no other equation can be oriented.

Example 4.1 For the unification problem

$$(x+1)*(y+1)*(z+1) \stackrel{?}{=} u*v*w,$$

216 solutions are computed. In the presence of the rewrite rule $x+1 \approx f(x)$, we can simplify the problem to

$$f(x)*f(y)*f(z) \stackrel{?}{=} u*v*w,$$

for which only 27 solutions are computed.

We can show the termination of lazy unification under certain conditions, imposed by the complexity measure in Definition 3.5, on the form of the rules with which the unification problems are simplified.

Chapter 5

Implementation

In this chapter, we present an overview of the experimental environment for Term Rewriting in Associative Commutative theories with Identities, *TRACONE*, which is currently being implemented in the programming language Standard ML (see [HMM 86]) in Version 75 of New Jersey.

The overall structure of *TRACONE* is outlined in Figure 4. Each module represents an SML-functor that can be separately compiled, allowing for an easy testing and recombination of the modules and a fast loading of the system. We shall briefly discuss the main features of the modules.

- A variety of modules is designed to provide basic abstract data types and functions. The module *parser* offers the facility to read data from a file and lexically analyze it. The abstract data structure of a finite binary relation and specializations of it are provided by the modules *relation*, *function*, and *ordering*. Various versions of the abstract data types of sets and multisets are provided, allowing the user to specify the equality of elements and the ordering in which the elements are stored.

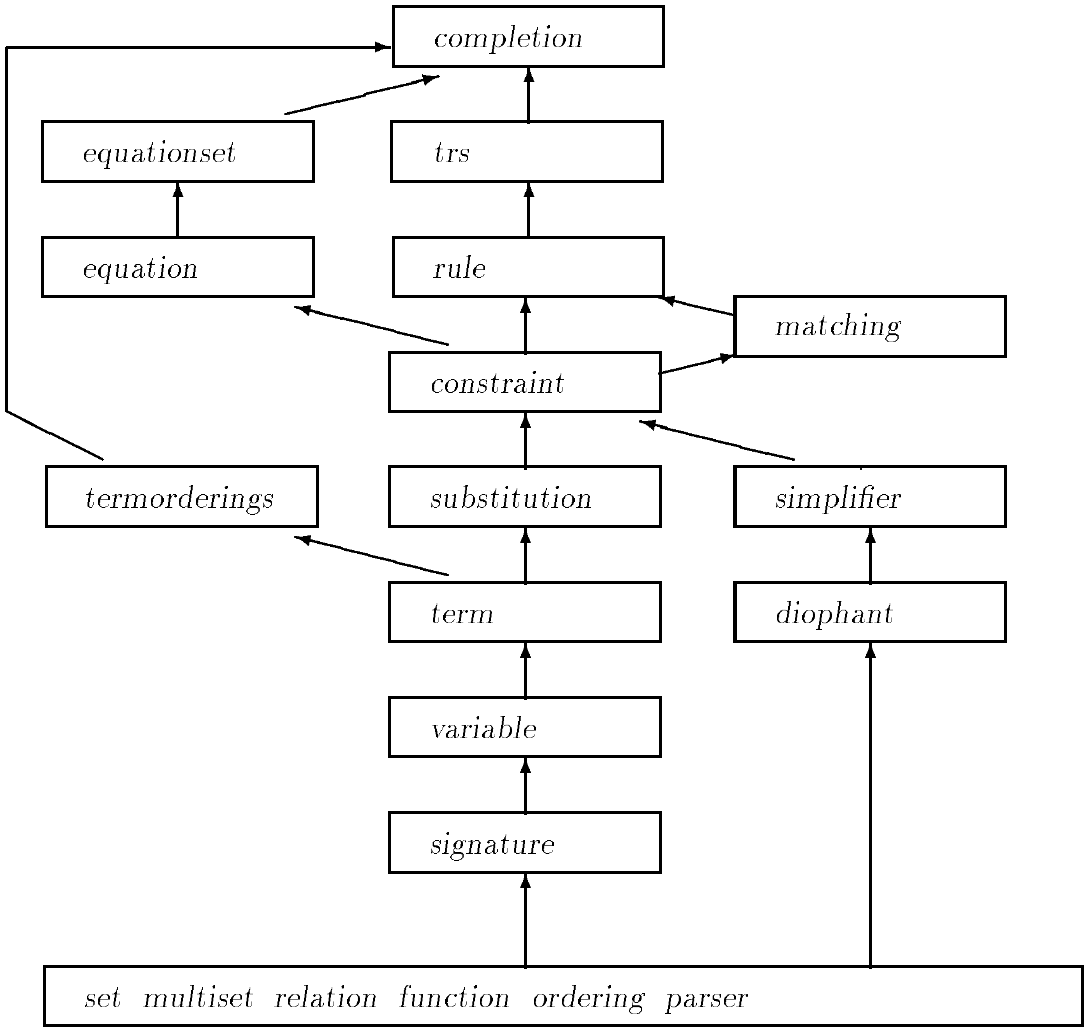

Figure 4: The structure of *TRACONE*

- The module *signature* is designed to interpret a user given signature, providing for multiple types, declaration of infix notation, associativity and commutativity of function symbols, and identity symbols.

- In *variable*, multi-typed variables are specified, including a mechanism to make data structures variable distinct, and a utility to generate new variables.

- The module *term* provides various functions operating on terms. The implementation of subterms and positions of terms is consistent with Section 2.1. Equality on terms is implemented as equality modulo $AC1$ and after every operation on terms the result is normalized with respect to $Flatten \cup Zero$.

- The module *substitution* provides the application of substitutions to variables, terms and substitutions.

- The notion of constraints in *constraint* is consistent with Section 2.4. Full and partial solving of constraints is supported, implementing the transformation rules in Section 3.4. For the basic simplification steps in the AC- and $AC1$-case,

- the module *simplifier* is used. It represents an implementation of the function `dio` in [Fag 84], using

- the solving of linear diophantine equations, implemented in *diophant* and based on [Huet 78].

- The module *matching* provides constrained $AC1$-matching and represents an optimized implementation of the transformation rules given in Appendix A.
- The modules *rules* and *trs* offer mechanisms to build $AC1$-rewrite systems and compute normal forms, whereas
- the modules *equation* and *equationset* provide methodes to build sets of unoriented equations.
- In the module *termorderings*, various versions of recursive path orderings ([Der 82]) are integrated into the framework of AC-path orderings (see [BP 85]). The user can specify a precedence ordering and the status of free function symbols (lexicographic left, lexicographic right, and multiset). AC-symbols have the status multiset.

In Appendix B, the use of *TRACONE* is demonstrated on a variety of examples.

Bibliography

[Bac 91] L. Bachmair. *Canonical Equational Proofs.* Birkhäuser Boston Incorporation, 1991.

[BD 89] L. Bachmair, N. Dershowitz. *Completion for rewriting modulo a congruence.* Theoretical Computer Science 1989 **67**, pp. 173-201.

[BP 85] L. Bachmair, D. Plaisted. *Termination orderings for associative-commutative rewriting systems.* Journal of Symbolic Computation 1985 **1**, pp. 329–349.

[BHK 88] H.-J. Bürckert, A. Herold, D. Kapur, J. Siekmann, M. Stickel, M. Tepp, H. Zhang. *Opening the AC-unification race.* Journal of Automated Reasoning 1988 **4,1**, pp. 465–474.

[BPW 89] T.B. Baird, G.E. Peterson, R.W. Wilkerson. *Complete Sets of Reductions Modulo Associativity, Commutativity and Identity.* In: Proc. Rewriting Techniques and Applications 1989, Chapel Hill, Lecture Notes in Computer Science 355, pp. 29–44, Springer-Verlag, 1989.

[Bou 90] A. Boudet. *Unification dans les mélanges de théories equationnelles.* PhD thesis, Université de Paris-Sud, Centre d'Orsay, France, 1990.

[Der 82] N. Dershowitz. *Orderings for term-rewriting systems.* Theoretical Computer Science 1982 **17,3**, pp. 279–301.

[Fag 84] F. Fages. *Associative-Commutative Unification.* In: Proceedings 7^{th} International Conference on Automated Deduction, Napa Valley (California, USA), R. Shostak, editor, Lecture Notes in Computer Science 170, pp. 194–208, Springer-Verlag, 1984.

[HMM 86] R. Harper, D. MacQueen, R. Milner. Standard ML. Edinburgh University Internal Report ECS-LFCS-86-2, 1986.

[Huet 78] G. Huet. *An algorithm to generate the basis of sulutions to homogeneous linear diophantine equations.* Information Processing Letters 1978 **7,3**, pp. 144–147.

[Huet 80] G. Huet. *Confluent reductions: Abstract properties and applications to term rewriting systems.* Journal of the Association for Computing Machinery 1980 **27**, pp. 797–821.

[JK 91] J.-P. Jouannaud, C. Kirchner. *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification.* Research Report CRIN 1991, to appear in: Festschrift for Robinson. J. L. Lassez, G. Plotkin, editors. MIT Press.

[JM 90] J.-P. Jouannaud, C. Marché. *Completion modulo associativity, commutativity and identity (AC1).* In: Proc. Int. Symp. DISCO 1990, Lecture Notes in Computer Science 429, pp. 111–120, Springer-Verlag, 1990.

[Kir 89] C. Kirchner. *From unification in combination of equational theories to a new AC-unification algorithm.* In: Resolution of Equations in Algebraic Structures, H. Ait-Kaci, M. Nivat, editors, Volume 2: Rewriting Techniques, pp. 171–210, Academic Press, 1989.

[KB 70] D. E. Knuth, P. B. Bendix. *Simple word problems in universal algebras.* In: Computational Problems in Abstract Algebra, J. Leech, editor, pp. 263–297, Oxford, Pergamon Press.

[KKR 91] C. Kirchner, H. Kirchner, M. Rusinowitch. *Deduction with Symbolic Constraints.* Revenue Française d' Intelligence Artificielle 1990 **4,3**, special issue on automated deduction, pp. 9–52.

[LB 77] D. S. Lankford, A. M. Ballantyne. *Decision procedures for simple equational theories with associative-commutative axioms: Complete sets of associative-commutative reductions.* Technical report ATP-39, Department of Mathematics and Computer Science, University of Texas, Austin, 1977.

[LS 75] M. Livesey, J. Siekmann. *Temination and decidability results for string unification.* Technical report CSM-12, University of Essex, 1975.

[Mar 91] C. Marché. *On ground AC-completion.* In: Proceedings 4^{th} International Conference on Rewriting Techniques and Applications, Como (Italy), R. Book, editor, Lecture Notes in Computer Science 488, pp. 411–422, Springer-Verlag, 1991.

[PS 81] G. E. Peterson, M. E. Stickel. *Complete sets of reductions for some equational theories.* Journal of the Association for Computing Machinery 1981 **28**, pp. 233–264.

[Smo 89] G. Smolka. *Logic Programming over Polymorphically Order Sorted Types.* PhD thesis, Universität Kaiserslautern, FB Informatik, West-Germany, 1989.

[Sti 81] M. E. Stickel. *A Unification Algorithm for Associative-Commutative Functions.* Journal of the Association for Computing Machinery 1981 **28**, pp. 423–434.

Appendix A

AC1-matching

A term s $AC1$-matches a term t via the substitution σ, if $s =_{AC1} t\sigma$. Just like $AC1$-unification, $AC1$-matching can be described with transformation rules. In Table 7, we give the set **AC1-Match** of transformation rules. Note that we require both terms to be in normal form with respect to $Flatten \cup Zero$. $AC1$-matching procedures can be defined similar to $AC1$-unification procedures (see Definition 3.7) and can be proved to be sound and complete similar to Lemmas 3.4 and 3.5. All $AC1$-matching procedures terminate as can be shown using an ordering based on (the multiset extension of the set extension of) a simple ordering on terms.

Remarks:

- We employ the notation from section 3.2 but can do without existential quantification. The symbol $\stackrel{?}{=}_M$ is **not** commutative.
- In the rules **AC Decompose** and **AC1 Decompose**, the following notations are used: $\hat{f}(X)$ denotes $zero(f)$ if $|X| = 0$, X^1 if $|X| = 1$,

Delete:

$$(r \stackrel{?}{=}_M s \wedge c, k) \;\vee\; d \quad \Rightarrow \quad (c, k) \;\vee\; d$$
$$\text{if } r =_{AC1} s \text{ and } r \text{ ground}$$

Fail:

$$(f(X) \stackrel{?}{=}_M g(Y) \wedge c, k) \;\vee\; d \quad \Rightarrow \quad d$$
$$\text{if } f \notin F_{AC1} \text{ and } f \neq g$$

Eliminate:

$$(x \stackrel{?}{=}_M t \wedge c, k) \;\vee\; d \quad \Rightarrow \quad (c, x \stackrel{?}{=}_M t \wedge k) \;\vee\; d$$
$$\text{if } \begin{cases} x \in V \\ \text{no } x \stackrel{?}{=}_M t' \text{ with } t' \neq_{AC1} t \text{ in } k \end{cases}$$

Clash:

$$(x \stackrel{?}{=}_M t \wedge c, k) \;\vee\; d \quad \Rightarrow \quad d$$
$$\text{if } \begin{cases} x \in V \\ \text{some } x \stackrel{?}{=}_M t' \text{ with } t' \neq_{AC1} t \text{ in } k \end{cases}$$

Free Decompose:

$$(f(X) \stackrel{?}{=}_M f(Y) \wedge c, k) \;\vee\; d \quad \Rightarrow \quad (\textstyle\bigwedge_{i \in [1..|X|]} X^i \stackrel{?}{=}_M Y^i, k) \;\vee\; d$$
$$\text{if } f \in F_{free}$$

AC Decompose:

$$(f(X) \stackrel{?}{=}_M f(Y) \wedge c, k) \;\vee\; d \quad \Rightarrow$$
$$\textstyle\bigvee_{p \in ACPartit(f,X,Y)} (\bigwedge_{i \in [1..|X|]} X^i \stackrel{?}{=}_M \hat{f}(p(i)) \;\wedge c, k) \;\vee\; d$$
$$\text{if } f \in F_{AC} \text{ and } f \notin F_{AC1}$$

AC1 Decompose:

$$(f(X) \stackrel{?}{=}_M f(Y) \wedge c, k) \;\vee\; d \quad \Rightarrow$$
$$\textstyle\bigvee_{p \in AC1Partit(f,X,Y)} (\bigwedge_{i \in [1..|X|]} X^i \stackrel{?}{=}_M \hat{f}(p(i)) \;\wedge c, k) \;\vee\; d$$
$$\text{if } f \in F_{AC1}$$

AC1 Collapse:

$$(f(X) \stackrel{?}{=}_M t \wedge c, k) \;\vee\; d \quad \Rightarrow$$
$$\textstyle\bigvee_{i \in [1..|X|]} (\bigwedge_{j \in [1..|X|], j \neq i} X^j \stackrel{?}{=}_M zero(f) \;\wedge X^i \stackrel{?}{=}_M t \wedge c, k) \;\vee\; d$$
$$\text{if } f \in F_{AC1} \text{ and } t \in V \text{ or } root(t) \neq f$$

Table 7: The transformation rules **AC1-Match**

and $f(X)$ if $|X| > 1$. When $f \in F_{AC}$ and X and Y are sequences of terms, then $ACPartit(f, X, Y)$ is the set of all mappings p from $[1..|X|]$ to the set of all subsequences of Y such that for all $j \in [1..|Y|]$ holds $|Y|_{Y^j} = \sum_{i\in[1..|X|]} |p(i)|_{Y^j}$, and such that for all $i \in [1..|X|]$ holds $|p(i)| \geq 1$. When $f \in F_{AC1}$ then $AC1Partit$ is the set of all mappings p from $[1..|X|]$ to the set of all subsequences of Y such that for all $j \in [1..|Y|]$ holds $|Y|_{Y^j} = \sum_{i\in[1..|X|]} |p(i)|_{Y^j}$.

- When we want to find the substitutions with which a term s matches a term t, we apply the rules in **AC1-Match** exhaustively to the matching problem $(t \downarrow_{Flatten \cup Zero} \stackrel{?}{=}_M s \downarrow_{Flatten \cup Zero} \wedge \top) \vee \bot$.

- We obtain the matching substitutions from a solved form $\bigvee_{i\in[1..n]}(\top, k_i)$ by removing redundant atomic problems from k_i for each $i \in [1..n]$. Redundant are problems of the form $x \stackrel{?}{=}_M y$, where $x = y$, and problems $x \stackrel{?}{=}_M t$, for which $x \stackrel{?}{=}_M t'$ occurs in the rest of k_i. Note that in the latter case, $t =_{AC1} t'$ (see condition for **Eliminate**).

- A complete set of matching substitutions is computed. If a most general complete set of matching substitutions is wanted, redundant substitutions must be eliminated in an additional pass over the resulting set of substitutions.

Appendix B

Examples

We demonstrate the use of *TRACONE* by giving an example session. The result of an evaluation of an SML-statement is indicated by ">>".

B.1 Rewriting

The file `sign` contains the signature from Example 2.1 in the syntax of *TRACONE*.

```
(* sign *)
          a  :      ->  x
          b  :      ->  x
          c  :      ->  x
          d  :      ->  x
          e  :      ->  x
          -  :  x   ->  x
          f  :  x   ->  x
```

```
             g   :  x x  ->  x
ac    infix  #   :  x x  ->  x
ac    infix  +   :  x x  ->  x
zero         0  to  +
ac    infix  *   :  x x  ->  x
zero         1  to  *
```

The file `sign` is interpreted as a signature and a string as a term on this signature.

```
val sign = read_signature "sign";
val t1 = Term.parse "1 + (0 + (f(x_1) + f(x_2)))" sign;
Term.print t1;
>>  f(x_1) + f(x_2)
```

Note that terms are reduced to normal form with respect to $Flatten \cup Zero$ after every operation on terms such as parsing. *TRACONE* uses the extended notion of subterms and positions as described in Section 2.1.

```
val t2 = Term.parse "g(0,x_1) * (x_1 + x_2) * 0" sign;
Term.print t2;
>>  (x_1 + x_2) * g(0,x_1) * 0
positions t2;
>>  [position ([],[2,3]),position ([],[1,3]),position ([],[1,2]),
     position ([],[]),   position ([3],[]),  position ([2],[]),
     position ([2,2],[]),position ([2,1],[]),position ([1],[]),
     position ([1,2],[]),position ([1,1],[])]
Term.list_print (subterms t2);
```

```
>>  g ( 0 , x_1 ) * 0
    ( x_1 + x_2 ) * 0
    ( x_1 + x_2 ) * g ( 0 , x_1 )
    ( x_1 + x_2 ) * g ( 0 , x_1 ) * 0
    0
    g ( 0 , x_1 )
    x_1
    0
    x_1 + x_2
    x_2
    x_1
```

Note that the order of the subterms in an AC-term may change. The file `ring` contains a convergent $AC1$-rewrite system for the algebra *commutative ring with unit* (see [BPW 89]).

```
(* ring *)
                          |  u + (- v) + v  =>  u                .
                          |        - (- u)  =>  u                .
u /= 0  v /= 0            |      - (u + v)  =>  (- u) + (- v)    .
w /= 1  u /= 0  v /= 0    |    w * (u + v)  =>  (w * u) + (w * v).
u /= 1                    |          u * 0  =>  0                .
u /= 1                    |       ( -v) * u =>  - (v * u)        .
```

The file `ring` is interpreted as a rewrite system and the terms `t3` and `t4` are reduced to normal form with respect to this rewrite system using $AC1$-matching.

```
val ring = Trs.read "ring" sign;
val t3 = Term.parse "f(x_1) + f(x_2) + (-(f(x_1)))" sign;
Term.print t3;
>>  (-(f(x_1))) + f(x_2) + f(x_1)
val normal = Trs.normal_form ring t3;
Term.print normal;
>>  f(x_2)
val t4 = Term.parse "f((x_1 + x_2) * g(x_3,x_1))" sign;
Term.print t4;
>>  f((x_1 + x_2) * g(x_3,x_1))
val normal = Trs.normal_form ring t4;
Term.print normal;
>>  f((x_1 * g(x_3,x_1)) + (x_2 * g(x_3)))
```

Note that the rewriting facilities of *TRACONE* represent the implementation of a programming language based on term rewriting, in which functions can be declared as being associative commutative and as identities to associative commutative functions.

B.2 Constraint Solving

The solving of the constraint $C_1 = x_1 \# x_2 \stackrel{?}{=} x_3 \# x_4$ demonstrates the solving of linear diophantine equations and the introduction of new variables.

```
val solutions_1 = Constraint.solve C_1;
Constraint.list_print solutions_1;
```

```
>> (* Constraint: *)
   (* Substitution *)
   x_1   <-   x_6 # x_5
   x_2   <-   x_8 # x_7
   x_3   <-   x_8 # x_6
   x_4   <-   x_7 # x_5

   (* Constraint: *)
   (* Substitution *)
   x_1   <-   x_3 # x_5
   x_4   <-   x_2 # x_5

   (* Constraint: *)
   (* Substitution *)
   x_1   <-   x_5 # x_4
   x_3   <-   x_2 # x_5

   (* Constraint: *)
   (* Substitution *)
   x_2   <-   x_3 # x_5
   x_4   <-   x_5 # x_1

   (* Constraint: *)
   (* Substitution *)
   x_3   <-   x_2
```

```
   x_4   <-   x_1

   (* Constraint: *)
   (* Substitution *)
   x_2   <-   x_5 # x_4
   x_3   <-   x_5 # x_1

   (* Constraint: *)
   (* Substitution *)
   x_3   <-   x_1
   x_4   <-   x_2

val number_of_solutions = length solutions_1;

>> val number_of_solutions = 7 : int
```

The presence of the indentity 0 for the function symbol + makes all but one solution of $C_2 \ = \ x_1 + x_2 \stackrel{?}{=} x_3 + x_4$ redundant.

```
val solutions_2 = Constraint.solve C_2;
Constraint.list_print solutions_2;

>> (* Constraint *)
   (* Substitution *)
   x_1   <-   x_6 + x_5
   x_2   <-   x_8 + x_7
   x_3   <-   x_8 + x_6
```

```
   x_4   <-   x_7 + x_5

val number_of_solutions = length solutions_2;

>> val number_of_solutions = 1 : int
```

The following examples are taken from the benchmark `acuni` in [BHK 88] (`acuni-001, acuni-036, acuni-064, acuni-079, acuni-096`) and were run on a SUN-4 (Sparc).

For $C_3 = x\#a\#b \stackrel{?}{=} u\#c\#d\#e$, we obtain:

```
>> Computing time for constraint solving in seconds: 0.450000
   val number_of_solutions = 2 : int
```

For $C_4 = x\#y\#a \stackrel{?}{=} u\#v\#w\#c$, we obtain:

```
>> Computing time for constraint solving in seconds: 3.200000
   val number_of_solutions = 204 : int
```

For $C_5 = x\#x\#a \stackrel{?}{=} u\#v\#c\#d$, we obtain:

```
>> Computing time for constraint solving in seconds: 1.190000
   val number_of_solutions = 60 : int
```

For $C_6 = x\#x\#y \stackrel{?}{=} u\#v\#c\#d$, we obtain:

```
>> Computing time for constraint solving in seconds: 3.580000
   val number_of_solutions = 228 : int
```

For $C_7 = x\#x\#x \stackrel{?}{=} u\#v\#w\#c$, we obtain:

```
>> Computing time for constraint solving in seconds: 172.070000
   val number_of_solutions = 6006 : int
```

The solving of the constraint $C_8 = x_1 \neq 1 \wedge x_1 \stackrel{?}{=} x_2 + x_3$ demonstrates the application of the transformation rule **Disequation Decompose** in **Disequation-AC1-Unify**.

```
val solutions_8 = Constraint.solve C_8;
Constraint.list_print solutions_8;


>> (* Constraint: *)
   (* Substitution *)
   x_1   <-   x_2 + x_3
   (* Disequations *)
   x_2  /=  0    x_2  /=  1

   (* Constraint: *)
   (* Substitution *)
   x_1   <-   x_2 + x_3
   (* Disequations *)
   x_3  /=  0    x_2  /=  0

   (* Constraint: *)
   (* Substitution *)
   x_1   <-   x_2 + x_3
   (* Disequations *)
   x_2  /=  1    x_3  /=  1
```

```
   (* Constraint: *)
   (* Substitution *)
   x_1   <-   x_2 + x_3
   (* Disequations *)
   x_3  /=  0    x_3  /=  1

val number_of_solutions = length solutions_8;

>> val number_of_solutions = 4 : int
```