2 Constraint Solving on Terms

Hubert Comon[1] and Claude Kirchner[2⋆]

[1] LSV, Ecole Normale Supérieure de Cachan, Cachan, France
[2] LORIA and INRIA, Villers-lès-Nancy, France

2.1 Introduction

In this chapter, we focus on constraint solving on terms, also called Herbrand constraints in the introductory chapter, and we follow the main concepts introduced in that chapter.

The most popular constraint system on terms is probably *unification problems* which consist of (possibly existentially quantified) conjunctions of equations. Such formulas have to be interpreted in the set of terms or one of its quotients (see e.g. [JK91, BS99] for surveys on unification). Other constraint systems on terms proved to be very useful, for instance introducing negation as in PROLOG II [Col82] leads to study *disunification* (see [Com91] for a survey). *Ordering constraints* on terms have been introduced to express ordered strategies in automated deduction (see the introductory chapter and the chapter on "contraints and theorem proving", or the survey [CT94]). Membership constraints may also express typing information and *set constraints* deserved a lot of works recently (see e.g. [PP97] for a survey), ...

We will try here to describe very briefly most of the constraint systems on terms, but our main focus will be on the method we are using. We follow the distinction of [CDJK99] between *syntactic methods* and *semantic methods*. Syntactic methods consist simply of rewriting the constraint into an equivalent one until a solved form is reached, hence rewriting techniques are relevant there (and we refer the reader to [DJ90a, BN98, KK99] for basics on rewriting). Semantic methods are based on another representation of the set of solutions. In the case of constraints on terms, such a representation could be fruitfully given by means of automata.

Therefore, our chapter is divided into two parts: the first part will describe some constraint systems and their corresponding syntactic methods. In particular, we consider unification problems in section 2.3, dis-unification problems in section 2.4, ordering constraints in section 2.5 and matching constraints in section 2.6. The ELAN system [BKK+98], designed to mechanize the use of inference rules, easily implements these ideas: its computation engine rewrites formulas under the control of strategies.

The second part of the lecture is dedicated to automata techniques. The relationship between logic and automata goes back to Büchi, Elgot and

⋆ Both authors supported by the ESPRIT working group CCL-II, ref. WG # 22457.

H. Comon, C. Marché, and R. Treinen (Eds.): CCL'99, LNCS 2002, pp. 47–103, 2001.

Church in the early sixties [Büc60, Elg61, Chu62]. The basic idea is to associate with each atomic formula a device (an automaton) which accepts all the models of the formula. Then, using some closure properties of the recognized languages, we can build an automaton accepting all the models of an arbitrary given formula. This is also the basis of optimal decision techniques (resp. model-checking techniques) for propositional temporal logic (see e.g. [Var96, BVW94]). In this lecture, we illustrate the method with three main examples, using three different notions of automata: Presburger arithmetic and classical word automata in section 2.8, typing constraints and tree automata in section 2.9, set constraints and tree set automata in section 2.10. This second part heavily relies on the book [CDG+97]. There was also a survey of these techniques at the CCL conference in 1994 [Dau94].

2.2 The Principle of Syntactic Methods

As explained in the introductory chapter, a constraint is simply a formula, together with its interpretation domain. The *syntactic methods* rely on an axiomatization of this domain, together with a strategy for the use of these axioms.

More precisely, a *constraint solving* method is defined by a (recursive) set of rewrite rules. Each rewrite rule $\phi \to \psi$ consists of a couple of constraint schemes ϕ, ψ, i.e. constraints with logical variables. For instance

$$x =^? t \wedge P \twoheadrightarrow x =^? t \wedge P\{x \mapsto t\}$$

where x is a variable, t a term, P a formula and $\{x \mapsto t\}$ is the substitution of x with t, is the *replacement rule*, x, t, P being logical variables of an appropriate type.

Each rule is assumed to rewrite a constraint into an equivalent one, i.e. both sides of the rewrite rule should have the same set of solutions in the constraint domain. Let us emphasize two important consequences of this assumption:

- the rewrite rules *replace* a constraint with another one. This makes an important difference with deduction rules: the premises are destroyed.
- If several rules can be applied to the same constraint, i.e. when the system is not deterministic, then we *don't care* which rule is actually applied. This makes a difference with e.g. logic programming in which the nondeterminism is "don't know". For constraint solving, we would like to never have to backtrack.

Often, the rewrite rules are simple combination of the domain axiomatization. (For instance, the replacement rule is a logical consequence of equality axioms).

Then, each rule comes with an additional condition, expressing a *strategy*. This is necessary for the termination of rewriting. For instance, the

replacement rule does not terminate. Hence we restrict it using the following conditions:

x occurs in P and does not occur in t. The rule is applied at top position.

These conditions impose both restrictions to the formulas to which the rule can be applied and restrictions on the positions at which the rule can be applied. The above condition ensures termination of the replacement rule alone (this is left as an exercise).

If the conditions are too strong, then constraint solving may become trivial. For example we could prevent any application of a rule, which, of course, ensures termination but is not very useful. Hence the definition of a constraint solving method includes the key notion of *solved form*. Solved forms are particular constraints, defined by a syntactic restriction and for which the satisfiability test should be trivial. The *completeness* of the set of rewrite rules expresses that any irreducible constraint is a solved form. We will see several examples in what follows.

In summary, a syntactic method consists in

A set of rewrite rules which is *correct*
Conditions on these rewrite rules which ensure *termination*
Solved forms with respect to which the rewrite rules are *complete*.

There are several advantages of such a presentation. First, it allows to define general environments and tools for the design of constraint solving methods. This is precisely the purpose of the programming language ELAN whose first class objects are precisely rules and strategies [KKV95, Cas98, Rin97, KR98]. We use ELAN in this lecture in order to operationalize constraint solvers. The system, together with its environment, the manual and many examples, is available at: `www.loria.fr/ELAN`.

A second advantage is to separate clearly the logical part from the control, which allows easier proofs and helps a better understanding. Finally, note that algorithms which are designed using this method are automatically incremental since new constraints can always be added (using a conjunction) to the result of former simplifications.

2.3 Unification Problems

We now define equational unification. But to give such a definition is not so easy since it should be simple and allow convincing soundness and completeness proofs. This is not the case with many of the definitions of equational unification which are given in the literature. The reason is that there are two contradicting goals: as far as generality is concerned, the definition should not depend upon a particular case, such as syntactic unification. On the other hand, since our approach is based on transforming problems in simpler ones having the same set of solutions, the definition must allow to get a clear,

simple and robust definition of equivalence of two unification problems but also to be able to define what simpler problem means.

2.3.1 Solutions and Unifiers

When defining equational problems and their solutions and unifiers, we should be aware that some variables may appear or go away when transforming a given unification problem into a simpler one. As a consequence, our definition of a solution to a unification problem should not only care about the variables occurring in the simplified problem, but also of the variables which have appeared at the intermediate steps. The idea that the so-called "new variables" are simply existentially quantified variables appeared first in Comon [Com88] although quantifiers had already been introduced by Kirchner and Lescanne [KL87] both in the more general context of disunification.

Definition 1. Let $\mathcal{F}$ be a set of function symbols, $\mathcal{X}$ be a set of variables, and $\mathcal{A}$ be an $\mathcal{F}$-algebra. A $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-*unification problem* (*unification problem* for short) is a first-order formula without negations nor universal quantifiers whose atoms are $\mathbf{T}, \mathbf{F}$ and $s =^?_{\mathcal{A}} t$, where s and t are terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We call an *equation* on $\mathcal{A}$ any $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification problem $s =^?_{\mathcal{A}} t$ and *multiequation* any multiset of terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Equational problems will be written as a disjunction of existentially quantified conjunctions:

$$\bigvee_{j \in J} \exists \overrightarrow{w_j} \bigwedge_{i \in I_j} s_i =^?_{\mathcal{A}} t_i.$$

When $|J| = 1$ the problem is called a *system*. Variables $\overrightarrow{w}$ in a system $P = \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i$ are called *bound*, while the other variables are called *free*. Their respective sets are denoted by $\mathcal{BV}ar(P)$ and $\mathcal{V}ar(P)$.

The superscripted question mark is used to make clear that we want to solve the corresponding equalities, rather than to prove them.

Example 1. With obvious sets $\mathcal{X}$ and $\mathcal{F}$ and for an $\mathcal{F}$-algebra $\mathcal{A}$,

$$\exists z \; f(x, a) =^?_{\mathcal{A}} g(f(x, y), g(z, a)) \wedge x =^?_{\mathcal{A}} z$$

is a system of equations where the only bound variable is z and the free variables are x and y.

A solution of an equational problem is a valuation of the variables that makes the formula valid:

Definition 2. A $\mathcal{A}$-*solution* (for short a solution when $\mathcal{A}$ is clear from the context) to a $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification system $P = \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i$ is a homomorphism h from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{A}$ such that

$$h, \mathcal{A} \models \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i.$$

$x + 0 \to x$	$0 + x \to x$
$x * 0 \to 0$	$0 * x \to 0$
$pred(succ(x)) \to x$	$succ(pred(x)) \to x$
$opp(0) \to 0$	$opp(opp(x)) \to x$
$x + opp(x) \to 0$	$opp(x) + x \to 0$
$opp(pred(x)) \to succ(opp(x))$	$opp(succ(x)) \to pred(opp(x))$
$succ(x) + y \to succ(x + y)$	$x + succ(y) \to succ(x + y)$
$x + pred(y) \to pred(x + y)$	$pred(x) + y \to pred(x + y)$
$opp(x + y) \to opp(y) + opp(x)$	$x * succ(y) \to (x * y) + x$
$succ(x) * y \to y + (x * y)$	$(x + y) + z \to x + (y + z)$
$opp(y) + (y + z) \to z$	$x + (opp(x) + z) \to z$
$pred(x) * y \to opp(y) + (x * y)$	$x * pred(y) \to (x * y) + opp(x)$

Fig. 2.1. BasicArithmetic: Basic arithmetic axioms.

An $\mathcal{A}$-solution to a $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification problem $D = \bigvee_{j \in J} P_j$, where all the P_j are unification systems, is a homomorphism h from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{A}$ such that h is solution of at least one of the P_j.

We denote by $\mathcal{S}ol_{\mathcal{A}}(D)$ the set of solutions of D in the algebra $\mathcal{A}$. Two $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification problems D and D' are said to be $\mathcal{A}$-*equivalent* if $\mathcal{S}ol_{\mathcal{A}}(D) = \mathcal{S}ol_{\mathcal{A}}(D')$, this is denoted $D' \Leftrightarrow_{\mathcal{A}} D$.

Note that because equations are interpreted as equality, the equation symbol is commutative. Therefore, except if explicitly mentioned, we make no difference between $s =^? t$ and $t =^? s$.

Finding solutions to a unification problem in an arbitrary $\mathcal{F}$-algebra $\mathcal{A}$ is impossible in general and when it is possible it is often difficult. For example, solving equations in the algebra $\mathcal{T}(\mathcal{F})/E$, where E is the BasicArithmetic theory given by the set of equational axioms described in Figure 2.1 is actually the problem of finding integer solutions to polynomial equations with integer coefficients. This is known as Hilbert's tenth problem, shown to be undecidable by Matijasevič [Mat70, DMR76].

Fortunately, we will see that the existence of solutions is decidable for many algebras of practical interest. However, there are in general infinitely many solutions to a unification system P. A first step towards the construction of a finite representation of these solutions is the notion of a *unifier,* which is meant as describing sets of *solutions*:

Definition 3. A $\mathcal{A}$-*unifier* of an $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification system

$$P = \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i$$

is a substitution (i.e. an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$) σ such that

$$\mathcal{A} \models \overline{\forall}\, \exists \overrightarrow{w} \bigwedge_{i \in I} \sigma_{|\mathcal{X} - \overrightarrow{w}}(s_i) = \sigma_{|\mathcal{X} - \overrightarrow{w}}(t_i)$$

where $\overline{\forall}P$ denotes the universal closure of the formula P.

A $\mathcal{A}$-unifier of a $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification problem $D = \bigvee_{j\in J} P_j$, where all the P_j are $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification systems, is a substitution σ such that σ unifies at least one of the P_j.

We denote by $\mathcal{U}_{\mathcal{A}}(D)$ the set of unifiers of D. This is abbreviated $\mathcal{U}(D)$ when $\mathcal{A}$ is clear from the context. Similarly when clear from the context, $\mathcal{A}$-unifiers are called unifiers and $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification is called unification.

The above definition is important, since it allows to define unifiers for equations whose solutions range over an arbitrary $\mathcal{F}$-algebra $\mathcal{A}$. This schematization is in particular due to the ability for the image of unifiers to contain free variables. These free variables are understood as universally quantified.

The relationship between solutions and unifiers is not quite as strong as we would like it to be. Of course, interpreting a unifier in the algebra $\mathcal{A}$ by applying an arbitrary homomorphism yields an homomorphism that is a solution. That all solutions are actually homomorphic images of unifiers is not true in general, but happens to be true in term generated algebras (allowing in particular free algebras generated by a given set of variables).

To illustrate the difference between solutions and unifiers, let us consider the set of symbols $\mathcal{F} = \{0, \succ, *\}$ and the $\mathcal{F}$-algebra $\mathbf{R}$ whose domain is the set of real numbers. Then $h(x) = \sqrt{2}$ is a solution of the equation $x * x =^?_{\mathbf{R}} \succ (\succ (0))$, although no $\mathbf{R}$-unifier exists for this equation, since the square root cannot be expressed in the syntax allowed by $\mathcal{F}$.

Indeed there is a large class of algebras where unifiers could be used as a complete representation of solutions since a unification system P has $\mathcal{A}$-solutions iff it admits $\mathcal{A}$-unifiers, provided $\mathcal{A}$ is a term generated $\mathcal{F}$-algebra.

Example 2. In the BasicArithmetic example, if we consider the equation $x =^? \succ (y)$, then $(x \mapsto 1, y \mapsto 2)$ is one of its $\mathbf{N}$-solutions. Its corresponding $\mathbf{N}$-unifier is $(x \mapsto succ(0), y \mapsto succ(succ(0)))$. The $\mathbf{N}$-unifier $(x \mapsto succ(y))$ also represents the previous $\mathbf{N}$-solution by simply valuating x to $succ(0)$ and y to $succ(succ(0))$.

In the following, we will restrict our attention to the special but fundamental case of the free algebras, initial algebras and their quotients. For these algebras the above property is satisfied since they are term generated by construction. As an important consequence of this property, two unification problems are equivalent iff they have the same sets of unifiers, an alternative definition that we are adopting in the remainder.

Definition 4. For a set of equational axioms E built on terms, for any terms s, t in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, an equation to be solved in $\mathcal{A} = \mathcal{T}(\mathcal{F}, \mathcal{X})/E$ is denoted by $s =^?_E t$. The equivalence of unification problems is denoted by $\Leftrightarrow_E$ and $\mathcal{A}$-unification is called *E-unification.*

Since a unification system is a term whose outermost symbol is the existential quantifier, its body part, i.e., the conjunction of equations occurring

in the system, is actually a subterm. Rewriting this subterm to transform the unification system into a simpler one hides the existential quantifier away. As a consequence, one may simply forget about the existential quantifier for most practical purposes. This is true for syntactic unification, where existential quantification is not needed as long as the transformation rules are chosen among the set given in Section 2.3.6. This is not, however, true of all algorithms for syntactic unification: Huet [Hue76] for example uses an abstraction rule introducing new variables in a way similar to the one used for combination problems.

Example 3. The equation $x+y =^?_E x+a$ is equivalent to the equation $y =^?_E a$, since the value of x is not relevant: our definition allows dropping useless variables. Note that this is not the case if unifiers are substitutions whose domain is restricted to the variables in the problem, a definition sometimes used in the literature, since $\{x \mapsto a, y \mapsto a\}$ would be a unifier of the first problem but not of the second.

Example 4. The equation $x+(y*y) =^?_E x+x$ is equivalent to $P' = \exists z\ x+z =^?_E x+x \wedge z =^?_E y*y$ whereas it is not equivalent to $P'' = x+z =^?_E x+x \wedge z =^?_E y*y$. In P'', z does not get an arbitrary value, whereas it may get any value in P, and in P' as well, since the substitution cannot be applied to the bound variable z.

Exercice 1 — Assuming the symbol + commutative, prove that $x + f(a,y) =^? g(y,b)+f(a,f(a,b))$ is equivalent to $(x =^? g(y,b) \wedge f(a,y) =^? f(a,f(a,b))) \vee (x =^? f(a,f(a,b)) \wedge f(a,y) =^? g(y,b))$.

2.3.2 Generating Sets

Unifiers schematize solutions; let us now consider schematizing sets of unifiers using the notion of complete set of unifiers.

Complete sets of unifiers. Unifiers are representations of solutions but are still infinitely many in general. We can take advantage of the fact that any instance of a unifier is itself a unifier to keep a set of unifiers minimal with respect to instantiation.

In order to express this in general, we need to introduce a slightly more abstract concept of equality and subsumption as follows.

Definition 5. Let $\mathcal{A}$ be an $\mathcal{F}$-algebra. We say that two terms s and t are $\mathcal{A}$-*equal*, written $s =_\mathcal{A} t$ if $h(s) = h(t)$ for all homomorphisms h from $\mathcal{T}(\mathcal{F},\mathcal{X})$ into $\mathcal{A}$. A term t is an $\mathcal{A}$-*instance* of a term s, or s is *more general* than t in the algebra $\mathcal{A}$ if $t =_\mathcal{A} \sigma(s)$ for some substitution σ; in that case we write $s \leq_\mathcal{A} t$ and σ is called a $\mathcal{A}$-match from s to t. The relation $\leq_\mathcal{A}$ is a quasi-ordering on terms called $\mathcal{A}$-*subsumption*.

Subsumption is easily lifted to substitutions:

Definition 6. We say that two substitutions σ and τ are $\mathcal{A}$-*equal* on the set of variables $V \subseteq \mathcal{X}$, written $\sigma =_{\mathcal{A}}^{V} \tau$ if $\sigma(x) =_{\mathcal{A}} \tau(x)$ for all variables x in V. A substitution σ is *more general* for the algebra $\mathcal{A}$ on the set of variable V than a substitution τ, written $\sigma \leq_{\mathcal{A}}^{V} \tau$, if there exists a substitution ρ such that $\rho\sigma =_{\mathcal{A}}^{V} \tau$. The relation $\leq_{\mathcal{A}}^{V}$ is a quasi-ordering on substitutions called $\mathcal{A}$-*subsumption*. V is omitted when equal to $\mathcal{X}$.

The definition of generating sets that we are now presenting is issued from the definition given first by G. Plotkin [Plo72] followed by G. Huet [Hue76] and J.-M. Hullot [Hul80b]. We denote the domain and the rank of a substitution σ respectively $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$.

Definition 7. Given an equational problem P, $CSU_{\mathcal{A}}(P)$ is a *complete set of unifiers* of P for the algebra $\mathcal{A}$ if:

(i) $CSU_{\mathcal{A}}(P) \subseteq \mathcal{U}_{\mathcal{A}}(P)$, (correctness)
(ii) $\forall \theta \in \mathcal{U}_{\mathcal{A}}(P)$, $\exists \sigma \in CSU_{\mathcal{A}}(P)$ such that $\sigma \leq_{\mathcal{A}}^{\mathcal{V}ar(P)} \theta$, (completeness)
(iii) $\forall \sigma \in CSU_{\mathcal{A}}(P), \mathcal{R}an(\sigma) \cap \mathcal{D}om(\sigma) = \emptyset$. (idempotency)

$CSU_{\mathcal{A}}(P)$ is called a *complete set of most general unifiers* of P in $\mathcal{A}$, and written $CSMGU_{\mathcal{A}}(P)$, if:

(iv) $\forall \alpha, \beta \in CSMGU_{\mathcal{A}}(P), \alpha \leq_{\mathcal{A}}^{\mathcal{V}ar(P)} \beta$ implies $\alpha = \beta$. (minimality)

Furthermore $CSU_{\mathcal{A}}(P)$ is said *outside the set of variables* W such that $\mathcal{V}ar(P) \subseteq W$ when:

(v) $\forall \sigma \in CSU_{\mathcal{A}}(P), \mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(P)$ and $\mathcal{R}an(\sigma) \cap W = \emptyset$. (protection)

Notice that unifiers are compared only on the problem variables (i.e., $\mathcal{V}ar(P)$), a fundamental restriction as pointed out in particular by F. Baader in [Baa91]. The conditions (idempotency) as well as (protection) in the above definition insure that the unifiers are idempotent.

Exercice 2 — Give a description of all the $\emptyset$-unifiers of the equation $x =^{?} y$. Then give a complete set of $\emptyset$-unifiers outside $\{x, y\}$. How does the elements of this set compare to the unifier $\alpha = \{x \mapsto y\}$? Is your complete set of unifiers minimal?

Minimal complete set of unifiers not always exist, as shown by [FH86]. For example, in the theory FH defined by:

$$\mathsf{FH} = \begin{cases} f(0, x) = x \\ g(f(x, y)) = g(y). \end{cases}$$

the equation $g(x) =_{\mathsf{FH}}^{?} g(0)$ has no minimal complete set of FH-unifiers. Indeed, with

$$\begin{aligned} \sigma_0 &= \{x \mapsto 0\} \text{ and} \\ \sigma_i &= \{x \mapsto f(x_i, \sigma_{i-1}(x))\}\ (0 < i) \end{aligned}$$

$\Sigma = \{\sigma_i | i \in \mathbf{N}\}$ is a complete set of FH-unifiers for the equation $g(x) =_{\mathsf{FH}}^{?} g(0)$ and $\forall i \in \mathbf{N}, \sigma_{i+1} \leq_{\mathsf{FH}}^{\{x\}} \sigma_i$.

2.3.3 (Un)-Decidability of Unification

Equational unification and matching are in general undecidable since there exist equational theories that have undecidable word problems. What is more disturbing is that very simple theories have undecidable E-unification problem. Let us review some of them.

Proposition 1. Let DA be the theory built over the set of symbols $\mathcal{F} = \{a, *, +\}$ and consisting of the axioms:

$$\begin{cases} x + (y + z) &= (x + y) + z \\ x * (y + z) &= (x * y) + (y * z) \\ (x + y) * z &= (x * z) + (x * z). \end{cases}$$

Unification is undecidable in DA as shown in [Sza82] using a reduction to Hilbert's tenth problem.

Another simple theory with undecidable unification problem is $D_l AU_r$, consisting in the associativity of $+$, the left distributivity of $*$ with respect to $+$ and a right unit element 1 satisfying $x * 1 = x$ [TA87].

In fact, decidability of unification is even quite sensitive to "new" constants. H.-J. Bürckert shows it by encoding the previous DA theory using new constants. This shows in particular that there exists equational theories for which unification is decidable but matching is not [Bür89].

The decidability of unification for classes of theories is also a very challenging problem. For example, variable permutative theories have an undecidable unification problem, as shown in [NO90], refining a result of [SS90]. Even in theories represented by a canonical term rewriting system (which is a strong requirement) the unification problem is undecidable:

Proposition 2. [Boc87] In the equational theory BasicArithmetic presented by the canonical term rewriting system $\overrightarrow{\text{BasicArithmetic}}$, the unification and matching problems are undecidable.

2.3.4 A Classification of Theories with Respect to Unification

Since we have seen that minimal complete sets of E-unifiers are isomorphic whenever they exist, a classification of theories based on their cardinality makes sense, as pioneered by Szabó and Siekmann [SS84,Sza82,SS82]. But in doing so, we should be careful with the fact that solving one single equation is not general enough a question, as shown by the following result:

Proposition 3. [BHSS89] There exists equational theories E such that all single equations have a minimal complete set of E-unifiers, but some systems of equations are of type zero i.e. have no minimal complete set of E-unifiers.

Thus it makes sense to define the type of an equational theory based on the cardinality of minimal complete sets of E-unifiers for equation systems, when they exist.

Let P be a system of equations in an equational theory E, and let $CSMGU_E(P)$ be a complete set of most general E-unifiers of P, whenever it exists. E-unification is said to be:

U-based if $CSMGU_E(P)$ exists for all problems P (the class of U-based theories is denoted by $\mathcal{U}$),
U-unitary if $E \in \mathcal{U}$ and $|CSMGU_E(P)| \leq 1$ for all P,
U-finitary if $E \in \mathcal{U}$ and $|CSMGU_E(P)|$ is finite for all P,
U-infinitary if E is U-based but not finitary,
U-nullary if E is not U-based,
U-undecidable if it is undecidable whether a given unification problem has unifiers.

Syntactic unification is unitary as we have seen in Section 2.3.6 and so is unification in boolean rings [MN89]. Commutative unification is finitary, as we will see next in Section 2.3.7. So is also associative-commutative unification. Associative unification is infinitary [Plo72], take for example the equation $x + a =^?_{\mathsf{A}(+)} a + x$ of which incomparable $\mathsf{A}(+)$-unifiers are $\{x \mapsto a\}, \{x \mapsto a + a\}, \{x \mapsto a + (a + a)\}, \cdots$. We have seen that the theory FH is nullary.

One can wonder if this classification can be enhanced by allowing U-finitary theories with only 2 most general elements and 3 and 4 ... , but this is hopeless due to the result of [BS86] showing that in any given U-finitary but non U-unitary theory, there exists an equation the complete set of unifiers of which has more that n elements for any given natural number n.

Finally, given a finite presentation of a theory E, its position in the unification hierarchy is undecidable, i.e. it is undecidable whether E is U-unitary, U-finitary, U-infinitary or U-nullary [Nut89].

A summary of the current knowledge on equational unification can be found in [KK99] or [BS99].

2.3.5 Transforming Equational Problems

Solved forms for Unification Problems. We now define the solved forms needed for equational unification. We assume the conjunction symbol ($\wedge$) to be associative and commutative.

Definition 8. A *tree solved form* is any conjunction of equations:

$$\exists \overrightarrow{z},\ x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$$

such that $\forall 1 \leq i \leq n, x_i \in \mathcal{X}$ and:

$$\begin{array}{lll} (i) & \forall 1 \leq i < j \leq n & x_i \neq x_j, \\ (ii) & \forall 1 \leq i, j \leq n & x_i \notin \mathcal{V}ar(t_j), \\ (iii) & \forall 1 \leq i \leq n & x_i \notin \overrightarrow{z}, \\ (iv) & \forall z \in \overrightarrow{z}, \exists 1 \leq j \leq n & z \in \mathcal{V}ar(t_j). \end{array}$$

Given a unification problem P, we say that $\exists \overrightarrow{z},\ x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$ is a tree solved form for P if it is a tree solved form equivalent to P and all variables free in $\exists \overrightarrow{z},\ x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$ are free variables of P.

In the above definition, the first condition checks that a variable is given only one value, while the second checks that this value is a finite term. The third and fourth conditions check that the existential variables are useful, i.e., that they contribute to the value of the other variables.

Tree solved forms have the property to be solvable:

Lemma 1. Let $\mathcal{A}$ be an $\mathcal{F}$-algebra. A unification problem P with tree solved form:

$$P = \exists \overrightarrow{z},\ x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$$

has, up to $\mathcal{A}$-subsumption equivalence, a unique most general idempotent unifier $\{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ in $\mathcal{A}$ which is denoted μ_P.

The notion of tree solved form could be extended to allow structure sharing, leading to the so-called dag solved form:

Definition 9. A *dag solved form* is any set of equations

$$\exists \overrightarrow{z}\ , x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$$

such that $\forall 1 \leq i \leq n, x_i \in \mathcal{X}$ and:

$$\begin{array}{lll} (i) & \forall 1 \leq i < j \leq n & x_i \neq x_j, \\ (ii) & \forall 1 \leq i \leq j \leq n & x_i \notin \mathcal{V}ar(t_j), \\ (iii) & \forall 1 \leq i \leq n & t_i \in \mathcal{X} \Rightarrow x_i,\ t_i \notin \overrightarrow{z}, \\ (iv) & \forall z \in \overrightarrow{z}, \exists 1 \leq j \leq n & z \in \mathcal{V}ar(t_j). \end{array}$$

Given a unification problem P, we say that $\exists \overrightarrow{z}\ , x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$ is a dag solved form for P if it is a dag solved form equivalent to P and all variables free in $\exists \overrightarrow{z}\ , x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$ are free variables of P.

Of course, a tree solved form for P is a dag solved form for P. Dag solved forms save space, since the value of the variable x_j need not be duplicated in the t_i for $i \leq j$. Conversely, a dag solved form yields a tree solved form by replacing x_i by its value t_i in all t_j such that $j < i$ and removing the remaining unnecessary existentially quantified variables. Formally the job is done by the following transformation rule:

Eliminate $P \wedge x =^? s$
$\quad \mapsto\!\!\!\!\rightarrow \{x \mapsto s\}P \wedge x =^? s$ if $x \notin \mathcal{V}ar(s), s \notin \mathcal{X}, x \in \mathcal{V}ar(P)$

Dag2Tree: Transformation of dag to tree solved forms

together with the simplification rules described in figure 2.2.

As a consequence we get solvability of dag solved forms:

Lemma 2. A unification problem $P = \exists \overrightarrow{z}\ , x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$ in dag solved form has, up to $\mathcal{A}$-subsumption equivalence, a unique most general idempotent unifier $\sigma = \sigma_n \cdots \sigma_2 \sigma_1$, where $\sigma_i = \{x_i \mapsto t_i\}$.

Dag solved forms relate to the so-called occur-check ordering on $\mathcal{X}$:

Definition 10. Given a unification problem P, let $\sim_P$ be the equivalence on $\mathcal{X}$ generated by the pairs (x, y) such that $x =^? y \in P$. The *occur-check* relation $\prec^{oc}$ on $\mathcal{X}$ defined by P is the quasi-ordering generated by the pairs (x', y') such that $x' \sim_P x, x =^? f(s_1, ..., s_n) \in P, y \in \mathcal{V}ar(f(s_1, ..., s_n)), y \sim_P y'$.

Example 5. For the system $P = (x =^? f(u, a) \wedge u =^? g(f(a, x)) \wedge x =^? y \wedge x =^? z)$ we have $x \sim_P y \sim_P z$ and $y \prec^{oc} u \prec^{oc} x$.

In a dag solved form, any two variables are not in the equivalence of the occur-check ordering. Conversely, a system of equations of the form $x =^? t$ with $x \in \mathcal{X}$ and such that $\prec^{oc}$ is acyclic, can be ordered (using topological sort) so as to meet the above condition. Accordingly, such a set of equations will be considered in dag solved form.

In the following, we refer without further precision to the most general unifier associated to a particular solved form by either one of the above lemmas.

Equivalence. We now state some commonly used transformations preserving the set of E-unifiers.

Proposition 4. Let E be a set of equational axioms built on terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Then, one can replace any subterm t in an equational problem P with an E-equal term without changing the set of E-unifiers of P.

In particular rewriting by some term rewriting system that is a sub-theory of E preserves the set of E-unifiers.

One quite important set of rules preserve equivalence of equational problems; they are the rules that allow manipulating the connectors $\wedge$ and $\vee$. The most commonly used such rules are described in Figure 2.2. This set of rules can be checked to be confluent modulo associativity-commutativity of

$$
\begin{array}{lll}
\textit{Associativity-}\wedge & (P_1 \wedge P_2) \wedge P_3 & = P_1 \wedge (P_2 \wedge P_3) \\
\textit{Associativity-}\vee & (P_1 \vee P_2) \vee P_3 & = P_1 \vee (P_2 \vee P_3) \\
\textit{Commutativity-}\wedge & P_1 \wedge P_2 & = P_2 \wedge P_1 \\
\textit{Commutativity-}\vee & P_1 \vee P_2 & = P_2 \vee P_1
\end{array}
$$

$$
\begin{array}{lll}
\textit{Trivial} & P \wedge (s =^? s) & \to P \\
\textit{AndIdemp} & P \wedge (e \wedge e) & \to P \wedge e \\
\textit{OrIdemp} & P \vee (e \vee e) & \to P \vee e \\
\textit{SimplifAnd1} & P \wedge \mathbf{T} & \to P \\
\textit{SimplifAnd2} & P \wedge \mathbf{F} & \to \mathbf{F} \\
\textit{SimplifOr1} & P \vee \mathbf{T} & \to \mathbf{T} \\
\textit{SimplifOr2} & P \vee \mathbf{F} & \to P \\
\textit{DistribCoD} & P \wedge (Q \vee R) & \to (P \wedge Q) \vee (P \wedge R) \\
\textit{Propag} & \exists \overrightarrow{z} : (P \vee Q) & \to (\exists \overrightarrow{z} : P) \vee (\exists \overrightarrow{z} : Q) \\
\textit{EElimin0} & \exists z : P & \to P \\
 & & \quad \mathsf{if}\ z \notin \mathcal{V}ar(P) \\
\textit{EElimin1} & \exists z : z =^? t \wedge P & \to P \\
 & & \quad \mathsf{if}\ z \notin \mathcal{V}ar(P) \cup \mathcal{V}ar(t)
\end{array}
$$

Fig. 2.2. *RAUP*: Rules and Axioms for Connectors Simplification in Unification Problems

conjunction and disjunction ($\vee$ and $\wedge$). Note that we choose to use distributivity (the rule *DistribCoD*) in such a way that we get disjunctive normal forms, a most convenient representation of *unification* problems. Remember also that we assume the equation symbol $=^?$ to be commutative.

Proposition 5. All the rules in *RAUP* preserve the set of $\mathcal{A}$-unifiers, for any $\mathcal{F}$-algebra $\mathcal{A}$.

2.3.6 Syntactic Unification

Syntactic unification or $\emptyset$-unification is the process of solving equations in the free algebra $\mathcal{T}(\mathcal{F}, \mathcal{X})$. In this section, unification problems are assumed to be unquantified conjunctions of equations. This is so because there is no need for new variables to express the (unique) most general unifier. The notions of solved forms are also used without quantifiers and we can now define the transformation rules.

Notation: Given a set of equations P, remember that $\{x \mapsto s\}P$ denotes the system obtained from P by replacing all the occurrences of the variable x by the term s. The size of a term s is denoted $|s|$.

Let *SyntacticUnification* be the set of transformation rules defined in Figure 2.3. The transformation rules *Conflict* and *Decompose* must be understood as schemas, f and g being quantified over the signature. This is handled

$$
\begin{array}{lll}
\textit{Delete} & P \wedge s =^? s & \\
& \mapsto\!\!\!\!\twoheadrightarrow P & \\
\textit{Decompose} & P \wedge f(s_1, \ldots, s_n) =^? f(t_1, \ldots, t_n) & \\
& \mapsto\!\!\!\!\twoheadrightarrow P \wedge s_1 =^? t_1 \wedge \ldots \wedge s_n =^? t_n & \\
\textit{Conflict} & P \wedge f(s_1, \ldots, s_n) =^? g(t_1, \ldots, t_p) & \\
& \mapsto\!\!\!\!\twoheadrightarrow \mathbf{F} & \text{if } f \neq g \\
\textit{Coalesce} & P \wedge x =^? y & \\
& \mapsto\!\!\!\!\twoheadrightarrow \{x \mapsto y\}P \wedge x =^? y & \text{if } x, y \in \mathcal{V}ar(P) \text{ and } x \neq y \\
\textit{Check*} & P \wedge x_1 =^? s_1[x_2] \wedge \ldots & \\
& \quad \ldots \wedge x_n =^? s_n[x_1] & \\
& \mapsto\!\!\!\!\twoheadrightarrow \mathbf{F} & \text{if } s_i \notin \mathcal{X} \text{ for some } i \in [1..n] \\
\textit{Merge} & P \wedge x =^? s \wedge x =^? t & \\
& \mapsto\!\!\!\!\twoheadrightarrow P \wedge x =^? s \wedge s =^? t & \text{if } 0 < |s| \leq |t| \\
\textit{Check} & P \wedge x =^? s & \\
& \mapsto\!\!\!\!\twoheadrightarrow \mathbf{F} & \text{if } x \in \mathcal{V}ar(s) \text{ and } s \notin \mathcal{X} \\
\textit{Eliminate} & P \wedge x =^? s & \\
& \mapsto\!\!\!\!\twoheadrightarrow \{x \mapsto s\}P \wedge x =^? s & \text{if } x \notin \mathcal{V}ar(s), s \notin \mathcal{X}, \\
& & \quad x \in \mathcal{V}ar(P)
\end{array}
$$

Fig. 2.3. *SyntacticUnification*: Rules for syntactic unification

in ELAN by using a specific construction called "For Each" and used in Figure 2.4. We avoid merging *Coalesce* and *Eliminate* into a single rule on purpose, because they do not play the same role. *Coalesce* takes care of variable renaming: this is the price to pay for alpha-conversion. *Eliminate* is quite different from *Coalesce* because it makes terms growing, thus we will see how to avoid applying it.

First, all these rules are sound i.e. preserve the set of unifiers:

Lemma 3. All the rules in *SyntacticUnification* are sound.

A strategy of application of the rules in *SyntacticUnification* determines a unification procedure. Some are complete, some are not, but a brute force fair strategy is complete:

Theorem 1. *Starting with a unification problem P and using the above rules repeatedly until none is applicable results in* $\mathbf{F}$ *iff P has no unifier, or else it results in a tree solved form of P:*

$$x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n.$$

Moreover

$$\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

is a most general unifier of P.

Exercice 3 — In fact, the condition $0 < |s| \leq |t|$ is fundamental in the *Merge* rule. Give a unification problem P such that without that condition *Merge* does not terminate.

Now that we have proved that the whole set of rules terminates, we can envisage complete restrictions of it. Let us first define useful subsets of the rules in *SyntacticUnification*. We introduce the set of rules:

TreeUnify = {*Delete, Decompose, Conflict, Coalesce, Check, Eliminate*}
and
DagUnify = {*Delete, Decompose, Conflict, Coalesce, Check*, Merge*}.

Corollary 1. Starting with a unification problem P and using the rules *TreeUnify* repeatedly until none is applicable, results in **F** iff P has no unifier, or else in a tree solved form:

$$x_1 =^? t_1 \wedge \ldots \wedge x_n =^? t_n$$

such that $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a most general unifier of P.

Exercice 4 — Apply the set of rules *TreeUnify* to the following unification problems:

$$f(g(h(x), a), z, g(h(a), y)) =^? f(g(h(g(y, y)), a), x, z)$$
$$f(g(h(x), a), z, g(h(a), y)) =^? f(a, x, z)$$

We can also forbid the application of the *Eliminate* rule, in which case we get dag solved forms:

Corollary 2. Starting with a unification problem P and using the rules *DagUnify* repeatedly until none is applicable, results in **F** iff P has no unifier, or else in a dag solved form

$$x_1 =^? t_1 \wedge \ldots \wedge x_n =^? t_n$$

such that $\sigma = \{x_n \mapsto t_n\} \ldots \{x_1 \mapsto t_1\}$ is a most general unifier of P.

Exercice 5 — Apply the set of rules *DagUnify* to the following unification problem:

$$f(g(h(x), a), z, g(h(a), y)) =^? f(g(h(g(y, y)), a), x, z)$$

Compare with what you get using the set of rules *TreeUnify*.

The previous results permit to implement complete solving strategies in ELAN. The complete module performing syntactic unification is given in Figure 2.4 and is available with the standard library of ELAN [BCD+98]. Notice that a unification problem is built over the binary operator $\wedge$ (the conjunction) which satisfies $AC(\wedge)$-axioms. Therefore, the unification rules are applied modulo these axioms. The decomposition rule depends on the signature;

```
module unification[Vars,Fss]
import global termV[Vars] termF[Fss] unifPb ;
 local Fss int identifier bool pair[identifier,int]
 list[pair[identifier,int]] eq[variable] eq[term] eq[Fsymbol]
 occur[variable,term] occur[variable,unifPb] ;
end

stratop global
 unify        : <unifPb -> UnifPb> bs;  end
                                                                    10
rules for unifPb
 P : unifPb; s,t : term; x,y : variable;
local
 [delete]     P ^ x=y => P
                  if eq_variable(x,y)                  end
 [coalesce]   P ^ x=y => apply(x->y,P) ^ x=y
                  if occurs x in P
                  and occurs y in P and neq_variable(x,y) end
 [conflict]   P ^ s=t => false
                  if not(isvar(s)) and not(isvar(t))                20
                  and neq_Fsymbol(head(s),head(t))  end
 [occ_check]  P ^ x=s => false
                  if occurs x in s and not isvar(s)   end
 [occ_check]  P ^ s=x => false
                  if occurs x in s and not isvar(s)   end
 [eliminate]  P ^ x=s => apply(x->s,P) ^ x=s
                  if not(occurs x in s)
                  and occurs x in P and not isvar(s)  end
 [eliminate]  P ^ s=x => apply(x->s,P) ^ x=s
                  if not(occurs x in s)                             30
                  and occurs x in P and not isvar(s)  end
 [trueadd] P => true ^ P                                 end
 [trueelim] true ^ P=> P                                 end
 [identity] P => P                                       end
end

 FOR EACH SS:pair[identifier,int]; F:identifier; N:int
 SUCH THAT SS:=(listExtract) elem(Fss) AND F:=()first(SS)
       AND N:=()second(SS) :{
 rules for unifPb                                                   40
 s_1,...,s_N:term; t_1,...,t_N:term;
 local
 [decompose] P ^ F(s_1,...,s_N)=F(t_1,...,t_N)
 =>
 P { ^ s_I=t_I }_I=1...N
 end end }

strategies for unifPb
implicit
 [] unify => dc one(trueadd) ;                                      50
 repeat*(dc(dc one(delete), dc one(decompose),
 dc one(conflict), dc one(coalesce),
 dc one(occ_check), dc one(eliminate))) ;
 dc one(trueelim, identity)
end end end
```

Fig. 2.4. Syntactic Unification in ELAN

a nice feature allowed in ELAN is to allow a natural expression of such rules by the use of a very general pre-processor. Note also the way strategies are described: dc expresses a "dont care" choice, repeat* applies as much as

$$
\begin{array}{ll}
\textit{Decompose} & P \wedge f(s_1, \ldots, s_n) =^? f(t_1, \ldots, t_n) \\
& \mapsto\!\!\!\!\!\rightarrow \\
& P \wedge s_1 =^? t_1 \wedge \ldots \wedge s_n =^? t_n \\
& \mathsf{if}\ f \neq + \\
\textit{ComMutate} & P \wedge s_1 + s_2 =^?_{\mathsf{C}} t_1 + t_2 \\
& \mapsto\!\!\!\!\!\rightarrow \\
& P \wedge \begin{pmatrix} s_1 =^?_{\mathsf{C}} t_1 \wedge s_2 =^?_{\mathsf{C}} t_2 \\ \vee \\ s_1 =^?_{\mathsf{C}} t_2 \wedge s_2 =^?_{\mathsf{C}} t_1 \end{pmatrix}
\end{array}
$$

Fig. 2.5. *CommutativeUnification*: The main rules for commutative unification

possible its argument and one allows to keep only one result out of all the possible ones.

Let us finally mention that syntactic unification has been proved to be linear in the size of the equation to be solved [PW78], provided that equality between two variable occurrences can be tested in constant time. But quasi-linear algorithms using dag-solved forms are often more useful in practice.

One may also ask whether syntactic unification can be speeded up by using massive parallelism. It is somewhat surprising that this is not the case, due to the non-linearity of terms. Dwork et al. [DKM84] show that unification of terms is logspace complete for P: unless $P \subset NC$, no parallel algorithm for unifying s and t will run in a time bounded by a polynomial in the logarithm of $|s| + |t|$ with a number of processors bounded by a polynomial in P.

2.3.7 Unification Modulo Commutativity

As in the free case, unification rules for unification modulo commutativity transform a unification problem into a set of equivalent unification problems in solved form. These rules are the same as for syntactic unification but should now embed the fact that some symbols are commutative. We give the modified rules for a commutative theory where we assume that $\mathcal{F} = \{a, b, \ldots, f, g, \ldots, +\}$ and $+$ is commutative, the others function symbols being free. Then a set of rules for unification in this theory can be easily obtained by adding a *mutation* rule to the set *Dag-Unify*, which describes the effect of the commutativity axiom. We call the resulting set of rules *CommutativeUnification*: the modified rules are described in Figure 2.5. It is very important to notice that a more modular and clever way to get a unification algorithm for this theory is indeed to *combine* unification algorithm for free symbols and for individual commutative symbols. This is detailed in the lecture by [Baader and Schultz].

Delete	$P \wedge s =^?_E s$	$\mapsto\!\!\!\!\rightarrow$	P
Decompose	$P \wedge f(\overrightarrow{s}) =^?_E f(\overrightarrow{t})$	$\mapsto\!\!\!\!\rightarrow$	$P \wedge s_1 =^?_E t_1 \wedge \ldots \wedge s_n =^?_E t_n$
Coalesce	$P \wedge x =^?_E y$	$\mapsto\!\!\!\!\rightarrow$	$\{x \mapsto y\}P \wedge x =^?_E y$ if $x, y \in \mathcal{V}ar(P)$ and $x \neq y$
Eliminate	$P \wedge x =^?_E s$	$\mapsto\!\!\!\!\rightarrow$	$\{x \mapsto s\}P \wedge x =^?_E s$ if $x \notin \mathcal{V}ar(s)$, $s \notin \mathcal{X}$ and $x \in \mathcal{V}ar(P)$
LazyPara	$P \wedge s =^?_E t$	$\mapsto\!\!\!\!\rightarrow$	$P \wedge s\|_p =^?_E l \wedge s[r]_p =^?_E t$ if $s\|_p \notin \mathcal{X}$ and $s\|_p(\Lambda) = l(\Lambda)$ where $l = r \in E$

Fig. 2.6. *GS-Unify*: Gallier and Snyder's rules for E-unification

We see how easy it is here to obtain a set of rules for unification modulo commutativity from the set of rules for syntactic unification. Note that again, there is no need of using existential quantifiers here.

Theorem 2. *Starting with a unification problem P and using repeatedly the rules CommutativeUnification, given in figure 2.5, until none is applicable results in* $\mathbf{F}$ *iff P has no C-unifier, or else it results in a finite disjunction of tree solved form:*

$$\bigvee_{j \in J} x_1^j =^?_C t_1^j \wedge \cdots \wedge x_n^j =^?_C t_n^j$$

having the same set of C-unifiers than P. Moreover:

$$\Sigma = \{\sigma^j | j \in J \text{ and } \sigma^j = \{x_1^j \mapsto t_1^j, \ldots, x_n^j \mapsto t_n^j\}\}$$

is a complete set of C-unifiers of P.

As for syntactic unification, specific complete strategies can be designed. Note also that by removing *Eliminate*, we get a set of rules for solving equations over infinite trees, exactly as in the free case.

2.3.8 General *E*-Unification

As we have seen, equational unification is undecidable since unification of ground terms boils down to the word problem. It is indeed semi-decidable by interleaving production of substitutions with generation of equational proofs. Gallier and Snyder gave a complete set of rules for enumerating a complete set of unifiers to a unification problem P in an arbitrary theory E [GS87, GS89, Sny88]. This set of rules is given is Figure 2.6.

The rule *LazyPara* (for lazy paramodulation) implements a lazy (since the induced unification problem is not solved right-away) use of the equations in E. Every time such an equation is used in the rule set, the assumption is

tacitly made that the variables of the equation are renamed to avoid possible captures.

Gallier and Snyder prove that for any E-unifier γ of a problem P, there exists a sequence of rules (where *Eliminate* and *Coalesce* are always applied immediately after *LazyPara*) whose result is a tree solved form yielding an idempotent unifier $\sigma \leq \gamma$. In this sense, the set of rules is complete. This result is improved in [DJ90b] where a restrictive version of *LazyPara* is proved to be complete. General E-unification transformations have also been given in [Höl89].

2.3.9 Narrowing

Narrowing is a relation on terms that generalizes rewriting in using unification instead of matching in order to apply a rewrite rule. This relation has been first introduced by M. Fay to perform unification in equational theories presented by a confluent and terminating term rewriting system, and this is our motivation for introducing it now. Another important application is its use as an operational semantics of logic and functional programming languages like BABEL [MNRA92], EQLOG [GM86] SLOG [Fri85], and this will be used in the chapter [[TOY]].

Narrowing a term t is finding an instantiation of t such that one rewrite step becomes applicable, and to apply it. This is achieved by replacing a non-variable subterm which unifies with a left-hand side of a rewrite rule by the right-hand side, and by instantiating the result with the computed unifier. In this process, is it enough to take the most general unifier. If this process is applied to an equation seen as a term with top symbol $=^?$, and is iterated until finding an equation whose both terms are syntactically unifiable, then the composition of the most general unifier with all the substitutions computed during the narrowing sequence yields a unifier of the initial equation in the equational theory. The narrowing process that builds all the possible narrowing derivations starting from the equation to be solved, is a general unification method that yields complete sets of unifiers, provided that the theory is presented by a terminating and confluent rewrite system [Fay79, Hul80a]. Furthermore, this method is incremental since it allows building, from a unification algorithm in a theory A, a unification procedure for a theory $R \cup A$, provided the class rewrite system defined by R and A is Church-Rosser and terminating modulo A [JKK83].

However, the drawback of such a general method is that it very often diverges and several attempts have been made to restrict the size of the narrowing derivation tree [Hul80a, NRS89, WBK94]. A successful method to solve this problem has been first proposed by J.-M. Hullot in restricting narrowing to *basic* narrowing. It has the main advantage to separate the solving of the syntactic unification constraint from the narrowing process itself. It is in fact a particular case of deduction with constraints, and the terminology "basic," indeed comes from this seminal work [Hul80a].

In this section we present the two relations of narrowing and basic (or constraint) narrowing and their application to the unification problem in equational theories presented by a terminating and confluent term rewrite system.

Narrowing relations.

Definition 11. (Narrowing) A term t is *narrowed* into t', at the non variable position $p \in \mathcal{D}om(t)$, using the rewrite rule $l \to r$ and the substitution σ, when σ is a most general unifier of $t|_p$ and l and $t' = \sigma(t[r]_p)$. This is denoted $t \leadsto_{[p,l\to r,\sigma]} t'$ and it is always assumed that there is no variable conflict between the rule and the term, i.e. that $\mathcal{V}ar(l,r) \cap \mathcal{V}ar(t) = \emptyset$.

For a given term rewriting system R, this generates a binary relation on terms called *narrowing* relation and denoted $\leadsto^R$.

Note that narrowing is a natural extension of rewriting since unification is used instead of matching. As a consequence the rewriting relation is always included in the narrowing one: $\longrightarrow^R \subseteq \leadsto^R$ since, for terms with disjoint sets of variables, a match is always a unifier.

Example 6. If we consider the rule $f(f(x)) \to x$ then the term $f(y)$ narrows at position Λ:

$$f(y) \leadsto_{[\Lambda, f(f(x))\to x, \{(x\mapsto z),(y\mapsto f(z))\}]} z.$$

On this example, we can notice that narrowing may introduce new variables, due to the unification step. Now, if we narrow the term $g(y, f(y))$, we get the following derivation:

$$\begin{aligned} g(y, f(y)) &\leadsto_{[2, f(f(x))\to x, \{(x\mapsto z),(y\mapsto f(z))\}]} g(f(z), z) \\ &\leadsto_{[1, f(f(x))\to x, \{(x\mapsto z'),(z\mapsto f(z'))\}]} g(z', f(z')) \\ &\dots \end{aligned}$$

which shows that even if the term rewriting system terminates, the narrowing derivation may not be so.

Exercice 6 — Use the system BasicArithmetic on page 51 to narrow the terms $succ(succ(0)) + pred(0)$, $succ(succ(x)) + pred(0)$, $succ(succ(x)) + pred(y)$.

Definition 12. A *constrained term* $(\exists W, t \parallel c)$ is a couple made of a term t and a system of constraints c together with a set of existentially quantified variables W. It schematizes the set of all instances of t by a solution of c with no assumption on W, i.e.

$$\{\exists W, \sigma(t) \mid \sigma \in \mathcal{S}ol(\exists W, c)\}.$$

The set of free variables of a constrained term $(\exists W, t \parallel c)$ is the union of the set of free variables of t and the free variables set of c minus W: $\mathcal{V}ar((\exists W, t \parallel c)) = \mathcal{V}ar(t) \cup \mathcal{V}ar(c) \setminus W$.

We consider in this section only constraint consisting of system of syntactic equations. This can be extended to more general constraint languages and domains as proposed for example in [KK89, KKR90, Cha94].

Example 7. The formula

$$\tau = (f(x, f(a, x)) \parallel \exists z, f(x, z) =^? f(g(a, y), f(a, y)) \wedge y =^? g(u, b))$$

is a constrained term. It schematizes the terms:

$$f(g(a, g(u, b)), f(a, g(a, g(u, b)))),$$
$$f(g(a, g(a, b)), f(a, g(a, g(a, b)))),$$
$$f(g(a, g(b, b)), f(a, g(a, g(b, b)))),$$
$$\ldots$$

and $\mathcal{V}ar(\tau) = \{x, y, u\}$.

Definition 13. (Constrained narrowing) A constrained term $(\exists W, t[u]_p \parallel c)$ *c-narrows* (narrows with constraints) into the constrained term

$$(\exists W \cup \mathcal{V}ar(l), t[r]_p \parallel c \wedge u =^?_\emptyset l)$$

at the non-variable position $p \in \mathcal{G}rd(t)$, using the rewrite rule $l \to r$ of the rewrite system R, if the system $c \wedge u =^?_\emptyset l$ is satisfiable and provided that the variables of the rule and the constrained terms are disjoint: $\mathcal{V}ar(l, r) \cap \mathcal{V}ar((t \parallel c)) = \emptyset$. This is denoted:

$$(\exists W, t[u]_p \parallel c) \overset{c}{\leadsto}^R_{[p, l \to r]} (\exists W \cup \mathcal{V}ar(l), t[r]_p \parallel c \wedge u =^?_\emptyset l).$$

Example 8. If we consider as previously the rule $f(f(x)) \to x$, then the term $(f(y) \parallel \mathbf{T})$ c-narrows at position Λ:

$$(\exists, f(y) \parallel \mathbf{T}) \overset{c}{\leadsto}_{[\Lambda, f(f(x)) \to x]} (\exists\{x\}, x \parallel f(y) =^?_\emptyset f(f(x))),$$

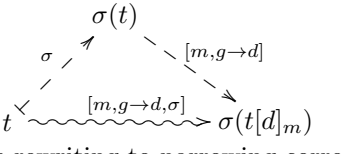
and similarly:

$$(\exists, g(y, f(y)) \parallel \mathbf{T}) \overset{c}{\leadsto}_{[2, f(f(x)) \to x,]} (\exists\{x\}, g(y, x) \parallel f(y) =^? f(f(x))).$$

The relation between rewriting and narrowing can be made more precise than just the trivial relationship $\longrightarrow^R \subseteq \leadsto^R$:

Lemma 4. For any term t and term rewriting system R, if $t \leadsto^R_{[m, g \to d, \sigma]} t'$ then $\sigma(t) \longrightarrow^R_{[m, g \to d]} t'$.

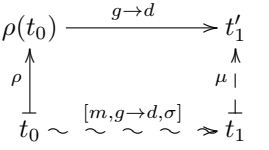
This can be pictured as follows:



The dual of this property, i.e. the rewriting to narrowing correspondence schema is more subtle and has been exhibited first by J.-M. Hullot [Hul80a, Hul80b].

Proposition 6. Let t_0 be a term and ρ be a R-normalized substitution such that $\rho(t_0) \longrightarrow^R_{[m,g\to d]} t'_1$. Then there exist substitutions σ et μ such that:

1. $t_0 \rightsquigarrow^R_{[m,g\to d,\sigma]} t_1$,
2. $\mu(t_1) = t'_1$,
3. $\rho =^{\mathcal{V}ar(t_0)} \mu\sigma$,
4. μ is R-normalized.



This result can be easily extended by induction on the number of steps to any rewriting derivation.

In order to apply narrowing to equational unification, it is convenient to introduce, for any rewrite system R on $\mathcal{T}(\mathcal{F},\mathcal{X})$, a new rule $x =^?_R x \to \mathbf{T}$ in which the symbols $=^?_R$ and $\mathbf{T}$ are considered as new symbols of the signature (i.e. $\mathbf{T}, =^?_R \notin \mathcal{F}$). This rule matches the equation $s =^?_R t$ if and only if s and t are identical; it narrows $s =^?_R t$ iff s and t are syntactically unifiable. Note that this could be also viewed as rewriting propositions instead of terms, an approach developed in [DHK98]. This leads to an easy characterization of R-unifiers:

Lemma 5. Let σ be a substitution, s and t be terms and R be any confluent rewrite system. Let $\bar{R} = R \cup \{x =^?_R x \to \mathbf{T}\}$. σ is a R-unifier of s and t if and only if $\sigma(s =^?_R t) \overset{*}{\longrightarrow}^{\bar{R}} \mathbf{T}$.

Since we also consider constrained equation systems, let us now define what a solution of such an entity is.

Definition 14. Let R be a term rewriting system on $\mathcal{T}(\mathcal{F},\mathcal{X})$. A *constrained system* is a constrained term $(\exists W, P \parallel c)$ where $P = \bigwedge_{i=1,\dots,n} s_i =^?_R t_i$ is a system of R-equations, i.e. a term in $\mathcal{T}(\mathcal{F} \cup \{\wedge, =^?_R\}, \mathcal{X})$ and c is a system of equations in the empty theory: $c = \exists W' \bigwedge_{i=1,\dots,n} s_i =^?_\emptyset t_i$. A *$R$-unifier* of a constrained system $(\exists W, P \parallel c)$ is a substitution σ which is a $\emptyset$-unifier of $(\exists W, c)$ and an R-unifier of $(\exists W, P)$.

For example $(x =^?_R y \parallel \mathbf{F})$ has no R-unifier and the R-solutions of $(s =^?_R t \parallel \mathbf{T})$ are the same as the R-unifiers of $s =^?_R t$.

Proposition 7. (Correctness) Let R be a term rewriting system. Let

$$G = (\exists W, s =^?_R t \parallel c)$$
$$G' = (\exists W \cup \mathcal{V}ar(l), s[r]_p =^?_R t \parallel c \wedge s|_p =^?_\emptyset l)$$

If $G \rightsquigarrow^R_{[1.p,l\to r]} G'$ then $\mathcal{U}_R(G') \subseteq \mathcal{U}_R(G)$.

Proving completeness could be achieved using generalizations of Lemma 4 and Property 6:

Narrow $(\exists W, s =^?_R t \parallel c)$
$\quad \mapsto\!\!\!\!\rightarrow\!\!\!\!\rightarrow$
$\quad (\exists W \cup \mathcal{V}ar(l), s[r]_p =^?_R t \parallel c \wedge s|_p =^?_\emptyset l)$
$\quad$ if $(c \wedge (s|_p =^?_\emptyset l))$ is satisfiable and $s|_p$ is not a variable

Block $(\exists W, s =^?_R t \parallel c)$
$\quad \mapsto\!\!\!\!\rightarrow\!\!\!\!\rightarrow$
$\quad (\exists W, \mathbf{T} \parallel c \wedge s =^?_\emptyset t)$
$\quad$ if $(c \wedge (s =^?_\emptyset t))$ is satisfiable

Fig. 2.7. *Narrowing*: Unification via constrained narrowing

Theorem 3. *Let R be a terminating and confluent term rewriting system and $\bar{R} = R \cup \{x =^? x \to \mathbf{T}\}$. If the substitution σ is a R-unifier of the terms s and t, then there exists a constrained narrowing derivation:*

$$(\exists \emptyset, s =^?_R t \parallel \mathbf{T}) \overset{c}{\leadsto}^{\bar{R}} \dots \overset{c}{\leadsto}^{\bar{R}} (\exists W_n, \mathbf{T} \parallel c_n),$$

such that $\sigma \in \mathcal{U}_\emptyset(c_n)$.

Let us consider now the *constrained narrowing tree* whose root is labeled with the constrained term $(\exists \emptyset, s =^?_R t \parallel \mathbf{T})$ and whose edges are all possible constrained narrowing derivations issued from a given node. In this tree, which is in general infinite, a *successful leave* is by definition a node labeled by a constrained term of the form: $(\exists W, \mathbf{T} \parallel c)$ with c satisfiable. For a given equation $s =^?_R t$, we denote $\mathcal{SNT}(s =^?_R t)$ the set of all successful nodes of the constrained narrowing tree issued from $(\exists \emptyset, s =^?_R t \parallel \mathbf{T})$.

Thanks to Theorem 3, we have:

$$\mathcal{U}_R(s =^?_R t) \subseteq \bigcup_{(\exists W, \mathbf{T} \parallel c) \in \mathcal{SNT}(s =^?_R t)} \mathcal{U}_\emptyset(c),$$

and since constrained narrowing is correct (by Property 7) we get the equality:

$$\mathcal{U}_R(s =^?_R t) = \bigcup_{(\exists W, \mathbf{T} \parallel c) \in \mathcal{SNT}(s =^?_R t)} \mathcal{U}_\emptyset(c).$$

This justifies the following main result about constrained narrowing:

Corollary 3. The transformation rules described in Figure 2.7, applied in a non deterministic and fair way to the constrained equation $(\exists \emptyset, s =^?_R t \parallel \mathbf{T})$, yield constrained equations of the form $(\exists W, \mathbf{T} \parallel c)$ such that the most general -unifiers of the c's form altogether a complete set of R-unifiers of $s =^?_R t$.

Note that in the set of rules *Narrowing*, the *Block* rule mimics exactly the application of the rule $x =^?_R x \to \mathbf{T}$.

Example 9. If we consider the rewrite system R reduced to the rule $f(f(y)) \to y$, then the constrained equation $(\exists\emptyset, f(x) =^?_R x \parallel \mathbf{T})$ is rewritten, using the rules *Narrowing* as follows:

$$\begin{array}{ll} & (\exists\emptyset, f(x) =^?_R x \parallel \mathbf{T}) \\ \mapsto\!\!\!\!\rightarrow_{\mathbf{Narrow}} & (\exists\{y\}, y =^?_R x \parallel f(f(y)) =^?_\emptyset f(x)) \\ \mapsto\!\!\!\!\rightarrow_{\mathbf{Block}} & (\exists\{y\}, \mathbf{T} \parallel y =^?_\emptyset x \wedge f(f(y)) =^?_\emptyset f(x)) \\ \mapsto\!\!\!\!\rightarrow_{\mathbf{SyntacticUnification}} & (\exists\{y\}, \mathbf{T} \parallel \mathbf{F}) \end{array}$$

See exercise 8 to conclude about the solution set of this equation.

Exercice 7 — Use the system BasicArithmetic on page 51 to solve the equation $x * x =^? x + x$ using *constrained* narrowing.

Exercice 8 — Let $R = \{f(f(x)) \to x\}$. Show that the standard narrowing is not terminating on the equation $f(y) =^?_R y$, as on the contrary basic or constrained narrowing does. What is the complete set of R-unifiers of this equation?

Notice that the previous proof of completeness of constrained narrowing can be extended to equational unification. This allows dealing in particular with narrowing modulo associativity and commutativity. Using normalization during the narrowing process could be achieved in a complete way. The latest paper on the subject, linking the problematic to the redundancy notions used in automated theorem proving, is [Nie95].

2.4 Dis-Unification Problems

As the name suggests, dis-unification is concerned with the generalization of unification to formulas where negation and arbitrary quantification are allowed. Many problems can be formalized in this setting and dis-unification has many useful applications from logic to computer science and theorem proving. Solving arbitrary first-order formulas whose only symbol is equality in an algebra $\mathcal{A}$ shows the decidability of the theory of $\mathcal{A}$ and provides a complete axiomatization of $\mathcal{A}$ [Mah88]. Dealing with negation in logic programming or when automating inductive theorem proving leads to solve dis-unification problems.

For a detailed motivation and presentation of dis-unification, see the surveys [CL89, Com91].

2.4.1 Equational Formulas, Semantics and Solved Forms

We call *equational formula* any first-order formula built on the logical connectives $\vee, \wedge, \neg$, the quantifiers $\forall$ and $\exists$ and which atomic formulas are equations or the constant predicate $\mathbf{T}$.

The set of $\mathcal{A}$-solutions is defined as previously for constraint systems.

The most common domains that have been considered in the literature are the following:

$\mathcal{A} = \mathcal{T}(\mathcal{F})$ **or** $\mathcal{T}(\mathcal{F}, \mathcal{X})$ This is the interpretation used when interested for example in complement problems (see for example [Com86]). The main point when dealing with such interpretation is the finiteness of $\mathcal{F}$ as we will see later. The case of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ will be considered as a special case of $\mathcal{T}(\mathcal{F})$ where the (infinite many) variables are considered as (infinitely many) constants.

$\mathcal{A} = RF(\mathcal{F})$ the algebra of rational terms is considered for example in constrained logic programming [Mah88]. When dealing with $IT(\mathcal{F})$, the algebra of infinite terms (not necessarily rational), [Mah88] has proved that a formula holds in $IT(\mathcal{F})$ if and only if it holds in $RT(\mathcal{F})$.

$\mathcal{A} = NF_{\mathcal{R}}(\mathcal{F})$ the subset of terms in $\mathcal{T}(\mathcal{F})$ that are in normal form with respect to the term rewrite system $\mathcal{R}$. Such interpretation are considered in particular in [Com89].

$\mathcal{A} = \mathcal{T}(\mathcal{F})/E$ where E is a finite set of equational axioms. This has been studied for specific set of axioms (with associativity-commutativity as an important theory from the application point of view) as well as in the general case using narrowing based techniques.

2.4.2 Solving Equational Formulas in $\mathcal{T}(\mathcal{F})$

Following the approach proposed in the section 2.2, we now define the solved forms under consideration when dealing with equational formulas when the interpretation domain is $\mathcal{A} = \mathcal{T}(\mathcal{F})$.

Definition 15. A basic formula is either **F**, **T** or

$$\exists \overrightarrow{z} : \; x_1 = t_1 \wedge \ldots \wedge x_n = t_n \wedge z_1 \neq u_1 \wedge \ldots \wedge z_m \neq u_m$$

where

- $x_1, \ldots, x_n$ are free variables which occur only once
- $z_1, \ldots, z_m$ are variables s.t. $\forall i, z_i \notin Var(u_i)$

Lemma 6. [Independence of dis-equations] If $\mathcal{T}(\mathcal{F})$ is infinite, and if E is a finite conjunction of equations and D is a finite conjunction of dis-equations then $E \wedge D$ has a solution in $\mathcal{T}(\mathcal{F})$ iff for each $s \neq t \in D$, $E \wedge s \neq t$ has a solution in $\mathcal{T}(\mathcal{F})$.

Lemma 7. The basic formulas which are distinct from **F** are solvable.

And indeed the solved forms are finite disjunction of basic formulas:

Theorem 4. *[Com91] If $\mathcal{A}$ is either $\mathcal{T}(\mathcal{F})$, $IT(\mathcal{F})$ or $RT(\mathcal{F})$ then any equational formula is equivalent to a finite disjunction of basic formulas.*

Let us now show how to compute the solved form of any equational formula in $\mathcal{T}(\mathcal{F})$.

$$\mathsf{SUP} - \mathit{DistribDoC}$$

$$\cup$$

Dis-equation	$\neg(s =^? t)$	$\to s \neq^? t$
TrivialDis	$s \neq^? s$	$\to \mathbf{F}$
NegT	$\neg\mathbf{T}$	$\to \mathbf{F}$
NegF	$\neg\mathbf{F}$	$\to \mathbf{T}$
NegNeg	$\neg\neg e$	$\to e$
NegDis	$\neg(e \vee e')$	$\to \neg e \wedge \neg e'$
NegConj	$\neg(e \wedge e')$	$\to \neg e \vee \neg e'$
DistribDoC	$e \vee (e' \wedge e'')$	$\to (e \vee e') \wedge (e \vee e'')$

Fig. 2.8. SEF: Rules for connectors Simplification in Equational Formulas

2.4.3 Solving Dis-Equation on Finite Terms

Our goal is here to transform any equational problem into its solved form assuming that the solving domain is $\mathcal{A} = \mathcal{T}(\mathcal{F})$. Indeed it is enough to transform the so called elementary formulas into basic formulas since, by successive application of the transformation rules and the appropriate use of double negation, this will allow to solve any equational formula. This is explained in [CL89][Section 6.1], presented by transformation rules is [Com91].

Definition 16. We call *elementary formulas* any disjunction of formulas of the form

$$\exists \overrightarrow{z}\ \forall \overrightarrow{y}\ P$$

where P is a quantifier-free equational formula.

The rules for connectors simplification (SUP) described in Figure 2.2 are extended and modified in order to cope with dis-equations. In particular, we orient this time distributivity (*DistribCoD*) in order to get conjunctive normal forms.

We give now some of the rules reducing elementary formulas to basic ones. A full description of the rules could be find in [Com91].

The replacement rule of unification should be extended to deal with dis-equations. In particular we get:

$$\mathit{Replacement2}\ z \neq^? t \vee P[z] \twoheadrightarrow z \neq^? t \vee \{z \mapsto t\}P[z]$$

if z is a free variable,
t does not contain any occurrence of a universally quantified variable,
$z \notin \mathcal{V}ar(t)$ and,
if t is a variable, then it occurs in P.

We have to deal directly with quantifiers that should be eliminated as far as possible either universal;

$$\begin{array}{lll} \textit{ElimUQ1} & \forall y:\ P & \twoheadrightarrow P \\ & & \quad \text{if } y \text{ does not occur free in } P \\ \textit{ElimUQ2} & \forall y:\ y \neq^? u \wedge P & \twoheadrightarrow \mathbf{F} \\ & & \quad \text{if } y \notin \mathcal{V}ar(u) \\ \textit{ElimUQ3} & \forall y:\ y \neq^? u \vee d & \twoheadrightarrow \{y \mapsto u\}d \\ & & \quad \text{if } y \notin \mathcal{V}ar(u) \end{array}$$

or existential:

$$\begin{array}{lll} \textit{ElimEQ1} & \exists x:\ P & \twoheadrightarrow P \\ & & \quad \text{if } x \text{ does not occur free in } P \\ \textit{ElimEQ2} & \exists z:\ z =^? t \wedge P & \twoheadrightarrow P \\ & & \quad \text{if } z \notin \mathcal{V}ar(t, P) \end{array}$$

With respect to unification, a new clash rule appears:

$$\textit{Clash} \quad f(t_1, \ldots, t_m) \neq^? g(u_1, \ldots, u_n) \twoheadrightarrow \mathbf{T} \quad \text{if } f \neq g$$

and the decomposition rule should take care of dis-equalities:

$$\begin{array}{ll} \textit{Decomp2} & f(t_1, \ldots, t_m) \neq^? f(u_1, \ldots, u_m) \\ & \twoheadrightarrow \\ & t_1 \neq^? u_1 \vee \ldots \vee t_m \neq^? u_m \\ \textit{Decomp3} & (f(t_1, \ldots, t_m) =^? f(u_1, \ldots, u_m) \vee d) \wedge P \\ & \twoheadrightarrow \\ & (t_1 =^? u_1 \vee d) \wedge \ldots \wedge (t_m =^? u_m \vee d) \wedge P \\ & \text{if one of the terms } t_1, \ldots, t_m, u_1, \ldots, u_m \text{ contains a} \\ & \quad \text{universally quantified variable, or else } d \text{ does not con-} \\ & \quad \text{tain any universally quantified variable} \end{array}$$

Of course the occur check may now also generate positive information:

$$\textit{OccurC2} \quad s \neq^? u[s] \twoheadrightarrow \mathbf{T} \quad \text{if } s \text{ and } u[s] \text{ are not syntactically equal}$$

When we assume that the domain consists in finite trees over a finite set of symbols, more quantifier eliminations can be performed like in:

$$\begin{array}{ll} \textit{ElimEQ3} & \exists \overrightarrow{w}:\ (d_1 \vee z_1 \neq^? u_1) \wedge \ldots \wedge (d_n \vee z_n \neq^? u_n) \wedge P \\ & \twoheadrightarrow \\ & \exists \overrightarrow{w}:\ P \\ & \text{if there exists a variable } w \in \overrightarrow{w} \cap Var(z_1, u_1) \cap \ldots \cap \\ & \quad Var(z_n, u_n) \text{ which does not occur in } P. \end{array}$$

and finite search could be used, when everything else have failed:

Explosion $\exists \overrightarrow{w_1} : P \quad \mapsto\!\!\!\!\!\rightarrow \bigvee_{f \in F} \exists \overrightarrow{w} \exists \overrightarrow{w_1} : P \wedge z =^? f(\overrightarrow{w})$

if $\overrightarrow{w} \cap \mathcal{V}ar(P) = \emptyset$,
no other rule can be applied and
there exists in P an equation or a dis-equation $z = u$ where u contains an occurrence of a universally quantified variable.

All these rules are correct and are consequences of the axiom system for finite trees over a finite alphabet. The full set of rules as given in [Com91] is terminating for any reduction strategy and the solved forms that are reached are basic formulas. As a consequence:

Theorem 5. *[Com88, Mah88] The first-order theory of finite trees is decidable.*

The above rules provide a way to reduce elementary formulas to basic ones. Of course obtaining solved form consisting only of equalities, and therefore giving a basis of the dis-unifiers set under the form of a complete set of substitutions is more appealing and as been addressed in [LM87, LMM88], [MS90, CF92], also in some equational cases [Fer98].

The complexity of dis-unification, even in the syntactic case where no equational theory is involved, can be high. In particular deciding satisfiability of elementary formulas is NP-complete [Pic99].

When extending dis-unification to equational theories, narrowing-based procedures could be designed [Fer92]. Unfortunately, even for theories as simple as associativity and commutativity, dis-unifiability becomes undecidable for general problems [Tre92] but is still decidable when considering the existential fragment i.e. elementary formulas without universal quantifiers [Com93]. It is also interesting to note that when restricting to shallow theories (where the equational axioms involve only variable at depth less than one), dis-unification is still decidable [CHJ94].

2.5 Ordering Constraints

We now consider the case where the symbolic constraints under consideration are based on an ordering predicate interpreted as a recursive path ordering on terms. We are presenting only the basic results and main references on the literature. A more detailed account of results could be found in the survey paper [CT94].

As this will be detailed in other lectures of this school, this kind of ordering constraints is extremely useful in theorem proving [KKR90, HR91, NR95], [BGLS95] as well as in programming language design since they allow to prune

the search space by keeping into account simplification as well as deletion strategies.

This section is devoted to ordering constraints where the ordering predicate is interpreted as a path ordering, but several other possibilities have been already explored:

Subterm ordering are studied in [Ven87].
Encompassment ordering [CCD93] play a central role in the ground reducibility problem [CJ94].
Matching ordering is surveyed in Section 2.6.

A *reduction ordering* $>$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is a well-founded ordering closed under context and substitution, that is such that for any context $C[\_]$ and any substitution σ, if $t > s$ then $C[t] > C[s]$ and $\sigma(t) > \sigma(s)$.

Reduction orderings are exactly what is needed when dealing with termination of term rewrite systems since a rewrite system R over the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is terminating iff there exists a reduction ordering $>$ such that each rule $l \to r \in R$ satisfies $l > r$.

Two most commonly used methods for building reduction orderings are polynomial interpretations and path orderings. Even if a very general notion of general path ordering can be defined [DH95], we restrict here to the *recursive path ordering* possibly with status.

Definition 17. Let us assume that each symbol f in the signature has a status, $Stat(f)$ which can be either lexicographic (lex) or multiset ($mult$).

The equality up to multisets, $=^{mult}$, is defined on terms as equality up to permutation of direct arguments of function symbols with multiset status.

Let $>_{\mathcal{F}}$ be a precedence on $\mathcal{F}$. The *recursive path ordering with status* $>_{rpos}$ is defined on terms by $s = f(s_1, .., s_n) >_{rpos} t = g(t_1, \ldots, t_m)$ if one at least of the following conditions holds:

1. $s_i >_{rpos} t$ or $s_i =^{mult} t$, for some i with $1 \leq i \leq n$, or
2. $f >_{\mathcal{F}} g$ and $\forall j \in \{1, \ldots, m\}, s >_{rpos} t_j$, or
3. $f = g$, $Stat(f) = lex$, $(s_1, \ldots, s_n) >^{lex}_{rpos} (t_1, \ldots, t_m)$ and $\forall i \in \{1, \ldots, n\}, s >_{rpos} t_i$, or
4. $f = g$, $Stat(f) = mult$ and $\{s_1, \ldots, s_n\} >^{mult}_{rpos} \{t_1, \ldots, t_m\}$.

where $>^{mult}_{rpos}$ and $^{lex}_{rpos}$ are the multiset and lexicographic extensions of $>_{rpos}$ respectively (the reader is referred to [KK99, BS99] for a definition of these extensions).

When all symbols have the lexicographic status, we speak of lexicographic path ordering (LPO for short). When conversely they all have the multiset status, we speak of the multiset or recursive path ordering (RPO for short). Such orderings are called generically *path orderings*.

A *term ordering constraint* is a quantifier-free formula built over the binary predicate symbols $>$ and $=$ which are interpreted respectively as a

given path ordering $\succ$ and a congruence on terms. We denote an inequation by $s \succ^? t$. A solution to an ordering constraint c is a substitution σ such that $\sigma(c)$ evaluates to true.

When solving such ordering constraints, a first main concern is about the introduction of new constants in order to express the solutions. This rises the distinction between fixed signature semantics and extended signature ones [NR95]. The satisfiability problem for ordering constraints has been shown to be decidable for fixed signature either when $>$ is a total LPO [Com90], or when it is a recursive path ordering with status [JO91]. In the case of extended signature, the decidability has been shown for RPO by [Nie93] and for LPO in [NR95]. Concerning complexity, NP algorithms have been given for LPO (in both fixed and extended signatures), RPO (for extended signatures) [Nie93] and for RPO with fixed signatures [NRV98]. NP-hardness is known, in all the cases and even for a single inequation [CT94].

Following the latest results of [Nie99] to which we refer for the full details, the satisfiability of a path ordering constraint can be achieved by first reducing it to a solved form which is then mainly checked to be occur-check free. The reduction to solved form follows the principle we have already seen for other constraint systems. The set of rules are of course using the definition of RPOS in order to decompose problem in simpler ones. For example, right decomposition is achieved via the rule:

$$\textit{decompR} \quad S \wedge s \succ^? f(t_1, \ldots, t_n) \twoheadrightarrow S \wedge s \succ^? t_1 \wedge \ldots \wedge s \succ^? t_n \quad \text{if } top(s) >_{\mathcal{F}} f$$

Left decomposition is achieved with some nondeterminism handled here by a disjunction:

$$\textit{decompL} \quad S \wedge f(s_1, \ldots, s_n) \succ^? t \twoheadrightarrow \bigvee_{1 \le i \le n} S \wedge s_i \succ^? t \vee \bigvee_{1 \le i \le n} S \wedge s_i =^? t \quad \text{if } top(t) >_{\mathcal{F}} f$$

where the introduced equations are solved modulo multiset equality.

2.6 Matching Constraints

The matching process is a symbolic computation of main interest for programming languages and in automated deduction. For applying a rewrite rule $l \to r$ to a (ground) term s, we have to decide if the left hand-side l of the rule "matches" a *ground* subterm t of s. This matching problem, denoted by $l \le^? t$ consists to unify l with a ground term t. According to this definition, matching is a very specific case of unification (unification with a *ground* term) and can be solved by reusing a unification algorithm (or strategy) if such an algorithm exists like in the empty theory. But there exist also equational theories where matching is decidable but unification is not, and in many situations, matching is much less complex than unification.

2.6.1 Syntactic Matching

The matching substitution from t to t', when it exists, is unique and can be computed by a simple recursive algorithm given for example by G. Huet [Hue76] and that we are describing now.

Definition 18. A *match-equation* is any formula of the form $t \ll^? t'$, where t and t' are terms. A substitution σ is solution of the match-equation $t \ll^? t'$ if $\sigma t = t'$. A *matching system* is a conjunction of match-equations. A substitution is solution of a matching system P if it is solution of all the match-equations in P. We denote by $\mathbf{F}$ a matching system without solution.

We are now ready to describe the computation of matches by the following set of transformation rules *Match* where the symbol $\wedge$ is assumed to satisfy the rules and axioms of figure 2.2.

Delete	$t \ll^? t \wedge P$	$\mapsto\!\!\!\!\rightarrow$	P
Decomposition	$f(t_1, \ldots, t_n) \ll^? f(t'_1, \ldots, t'_n) \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\bigwedge_{i=1,\ldots,n} t_i \ll^? t'_i \wedge P$
SymbolClash	$f(t_1, \ldots, t_n) \ll^? g(t'_1, \ldots, t'_m) \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ if $f \neq g$
MergingClash	$x \ll^? t \wedge x \ll^? t' \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ if $t \neq t'$
SymbVarClash	$f(t_1, \ldots, t_n) \ll^? x \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ if $x \in \mathcal{X}$

Match: Rules for syntactic matching

Theorem 6. *[KK99] The normal form by the rules in Match, of any matching problem $t \ll^? t'$ such that $\mathcal{V}ar(t) \cap \mathcal{V}ar(t') = \emptyset$, exists and is unique.*

1. *If it is $\mathbf{F}$, then there is no match from t to t'.*
2. *If it is of the form $\bigwedge_{i \in I} x_i \ll^? t_i$ with $I \neq \emptyset$, the substitution $\sigma = \{x_i \mapsto t_i\}_{i \in I}$ is the unique match from t to t'.*
3. *If it is $\mathbf{T}$ then t and t' are identical: $t = t'$.*

Exercice 9 — Write a program, in the language of your choice, implementing a matching algorithm derived from the *Match* set of rules.

Exercice 10 — Compute the match from the term $f(g(z), f(y, z))$ to the term $f(g(f(a, x)), f(g(c), f(a, x)))$.

It should be noted that the rule *Delete* in the previous set of rules is not correct when the two members of the match equation $t \ll^? t'$ share variables. This can be seen on the following example. The matching problem $f(x, x) \ll^? f(x, a)$ has no solution but the unrestricted use of the transformations leads

to: $f(x,x) \ll^? f(x,a) \mapsto\!\!\!\!\rightarrow_{\mathbf{Decomposition}} \{x \ll^? x, x \ll^? a\} \mapsto\!\!\!\!\rightarrow_{\mathbf{Delete}} \{x \ll^? a\}$ which has an obvious solution.

The above transformation rules could be easily generalized to deal with commutative matching. But the extension to associative *and* commutative matching is not straitforward and needs either to use semantical arguments like solving systems of linear Diophantine equations [Hul79, Mza86, Eke95] or to use the concept of syntactic theories [Kir85, KK90] to find the appropriate transformation rules [AK92, KR98].

2.7 Principles of Automata Based Constraint Solving

In the *syntactic methods* which have been presented previously, the aim of the constraint solving procedure is to reach a *solved form* which is an equivalent (yet simpler) representation of the set of solutions of the original constraint. Here, we consider another representation of the set of solutions: an automaton.

The relationship between logic and automata goes back to Büchi, Elgot and Church in the early sixties [Büc60, Elg61, Chu62]. The basic idea is to associate with each atomic formula a device (an automaton) which accepts all the models of the formula. Then, using the closure properties of the recognized languages, we can build an automaton accepting all the models of an arbitrary given formula. This is also the basis of optimal decision techniques (resp. model-checking techniques) for propositional temporal logic (see e.g. [Var96, BVW94]).

In this lecture, we illustrate the method with three main examples, using three different notions of automata:

Classical word automata. They allow to represent the set of solutions for arbitrary Presburger formulas, yielding a decision procedure for Presburger arithmetic. This technique is known since Büchi, but has been re-discovered recently with applications in constraint solving. This constraint solving method has been implemented in several places and competes with other arithmetic solvers as shown in [BC96, SKR98, Kla99]. We survey this very simple method in section 2.8.

Automata on finite trees. They allow to represent sets of terms, hence solutions of typing (membership) constraints on terms. We introduce such constraints as well as tree automata in section 2.9. There are several other applications of tree automata in constraint solving. For instance we will sketch applications in unification theory (sections 2.11.1, 2.11.2). They are also used e.g. in type reconstruction, in which case the interpretation domain is a set of trees representing types themselves (see e.g. [TW93]).

Tree set automata. They recognize sets of sets of terms. We define such a device in section 2.10 and show how it can be used in solving the so-called *set constraints* which were already surveyed at the CCL conference in 1994 [Koz94].

A significant part of these lecture notes is developed in [CDG+97]. There was already a survey of these techniques at the CCL conference in 1994 [Dau94].

2.8 Presburger Arithmetic and Classical Word Automata

We recall briefly the standard definitions in sections 2.8.1 and 2.8.2. Then we show how automata recognize sets of integers and we give a construction for the decision of arithmetic constraints.

2.8.1 Presburger Arithmetic

The *basic terms* in Presburger arithmetic consist of first-order variables, which we write $x, x_1, y, z' ...$, the constants 0 and 1 and sums of basic terms. For instance $x + x + 1 + 1 + 1$ is a basic term, which we also write $2x + 3$.

The *atomic formulas* are equalities and inequalities between basic terms. For instance $x + 2y = 3z + 1$ is an atomic formula.

The *formulas* of the logic are first-order formulas built on the atomic formulas. We use the connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\exists x$ (existential quantification), $\forall x$ (universal quantification). For instance, $\forall x, \exists y.(x = 2y \vee x = 2y + 1)$ is a formula. $\top$ is the empty conjunction (valid formula) and $\bot$ is the empty disjunction (unsatisfiable formula).

The *free variables* of a formula ϕ are defined as usual: for instance $FV(\phi_1 \vee \phi_2) = FV(\phi_1) \cup FV(\phi_2)$ and $FV(\exists x.\phi) = FV(\phi) \setminus \{x\}$.

The interpretation domain of formulas is the set of natural numbers $\mathbb{N}$, in which $0, 1, +, =, \leq$ have their usual meaning. A *solution* of a formula $\phi(x_1, \ldots, x_n)$ is an assignment of $x_1, \ldots, x_n$ in $\mathbb{N}$ which satisfies the formula. For instance $\{x \mapsto 0; y \mapsto 2; z \mapsto 1\}$ is a solution of $x + 2y = 3z + 1$ and every assignment $\{x \mapsto n\}$ is a solution of $\exists y.(x = 2y \vee x = 2y + 1)$.

2.8.2 Finite Automata

For every word $w \in A^*$, if i is a natural number which is smaller or equal to the length of w, we write $w(i)$ the ith letter of w.

A *finite automaton* on the alphabet A is a tuple (Q, Q_f, q_0, δ) where Q is a finite set of *states*, $Q_f \subseteq Q$ is the set of *final states*, $q_0 \in Q$ is the *initial state* and $\delta \subseteq Q \times Q \times Q$ is the *transition relation*.

Given a word $w \in A^*$ a *run* of the automaton (Q, Q_f, q_0, δ) on w is a word $\rho \in Q^*$ such that $\rho(1) = q_0$ and, if $\rho(i) = q$, $\rho(i+1) = q'$, then $(q, w(i), q') \in \delta$. A *successful run* ρ on w is a run such that $\rho(n+1) \in Q_f$ where n is the length of w. w is *accepted by the automaton* if there is a successful run on w. The *language* accepted (or recognized) by an automaton is the set of words on which there is a successful run.

The class of languages which are accepted by some finite automaton is closed by union, intersection and complement. Such constructions can be found in most textbooks. (See e.g. [Per90] for references).

2.8.3 Sets of Integers Recognized by a Finite Automaton

Every natural number can be seen as a word written in base k ($k \geq 1$) over the alphabet $A = \{0, ..., k-1\}$. For convenience, we write here its representation in a somehow unusual way from right to left. For instance thirteen can be written in base two as: 1011. There are more than one representation for each number, since we may complete the word with any number of 0's on the right: 101100 is another representation of thirteen in base two. Conversely, there is a mapping denoted $\widetilde{\ }^k$ from $\{0, ..., k-1\}^*$ into $\mathbb{N}$ which associates with each word a natural number: $\widetilde{1011\underline{0}}^2$ is thirteen.

n-uples of natural numbers can be represented in base k as a single word over the alphabet $\{0, ..., k-1\}^n$ by stacking the representations of each individual numbers. For instance the pair $(13, 6)$ (in base 10) can be represented in base 2 as a word over the alphabet $\{0,1\}^2$ as: $\begin{smallmatrix}1011\\0110\end{smallmatrix}$. Again there are more than one representation since the word can be completed by any number of $\begin{smallmatrix}0\\0\end{smallmatrix}$'s on the right.

2.8.4 A Translation from Presburger Formulas to Finite Automata

In the logic versus automata spirit we start by associating an automaton with each atomic formula.

Automata associated with equalities. For every basic formula

$$a_1x_1 + \ldots + a_nx_n = b \text{ (where } a_1, \ldots, a_n, b \in \mathbf{Z})$$

we construct the automaton by saturating the set of rules and the set of states, originally set to $\{q_b\}$ using the inference rule:

$$\frac{q_c \in Q}{q_d \in Q, \quad (q_c, \theta, q_d) \in \delta} \text{ if } \begin{cases} a_1\theta_1 + \ldots + a_n\theta_n =_2 c \\ d = \frac{c - a_1\theta_1 \ldots - a_n\theta_n}{2} \\ \theta \in \{0,1\}^n \text{encodes } (\theta_1, \ldots, \theta_n) \end{cases}$$

in other words: for every state $q_c \in Q$, compute the solutions $(\theta_1, \ldots, \theta_n)$ of $a_1x_1 + \ldots + a_nx_n = c$ modulo 2 and add the state q_d and the rule $q_c \xrightarrow{\theta} q_d$ where $d = \frac{c - a_1\theta_1 \ldots - a_n\theta_n}{2}$.

The initial state is q_b and, if $q_0 \in Q$, then this is the only final state. Otherwise there is no final state.

Example 10. Consider the equation $x + 2y = 3z + 1$.

Since $b = 1$, we have $q_1 \in Q$. We compute the solutions modulo 2 of $x+2y = 3z+1$: we get $\{(0,0,1),(0,1,1),(1,0,0),(1,1,0)\}$. Then we compute the new states: $\frac{1-0-0+3}{2} = 2$, $\frac{1-0-2+3}{2} = 1$, $\frac{1-1-0+0}{2} = 0$, $\frac{1-1-2+0}{2} = -1$, yielding $q_2, q_0, q_{-1} \in Q$, and the new transitions:

$$q_1 \xrightarrow{\substack{0\\0\\1}} q_2, q_1 \quad q_1 \xrightarrow{\substack{0\\1\\1}} q_1, q_1 \quad q_1 \xrightarrow{\substack{1\\0\\0}} q_0, q_1 \quad q_1 \xrightarrow{\substack{1\\1\\0}} q_{-1}.$$

Going on with states q_2, q_0, q_{-1}, we get the automaton of figure 2.9; we use circles for the states and double circles for final states.

For instance, consider the path from the initial state q_1 to the final state q_0 going successively through $q_{-1}q_{-2}q_0q_1q_2$. The word $\begin{smallmatrix}111100\\110001\\001110\end{smallmatrix}$ is accepted by the automaton, which corresponds to the triple (in base ten): $x = 15, y = 35, z = 28$ and one can verify $15 + 2 \times 35 = 3 \times 28 + 1$.

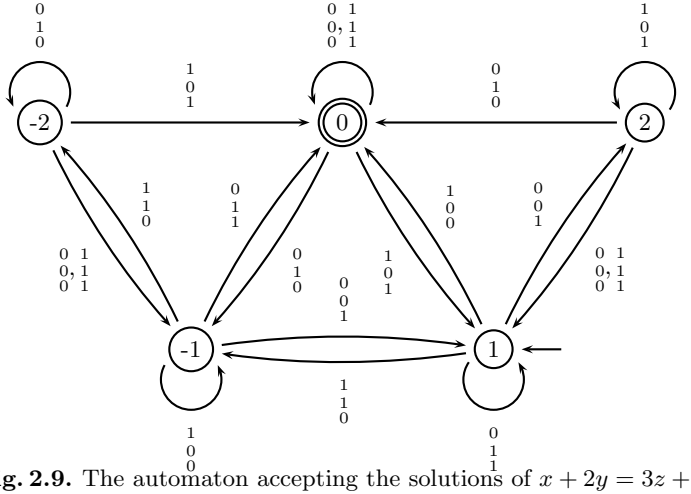


Fig. 2.9. The automaton accepting the solutions of $x + 2y = 3z + 1$

Proposition 8. *The saturation terminates yielding an (deterministic) automaton whose number of states is bounded by* $\max(|b|, |a_1| + \ldots + |a_n|)$ *and which accepts the solutions of* $a_1x_1 + \ldots + a_nx_n = b$.

Indeed, it is not difficult to check that the absolute values of the states are bounded by $\max(|b|, |a_1| + \ldots + |a_n|)$ (this is an invariant of the inference rule, the proof is left to the reader). The correctness and completeness are also easy: it suffices to show by induction on the length of the word (resp. on the length of the transition sequence) that, any word accepted starting from any state q_c is a solution of $a_1x_1 + \ldots + a_nx_n = c$.

The algorithm can be optimized in several respects, for instance precomputing the solutions modulo 2 or using Binary Decision Diagrams (BDDs) to represent the transitions of the automaton.

Exercise 1. If we fix $a_1, \ldots a_n$, what is the size of the automaton, w.r.t. b?

Automata associated with inequalities. Next, we compute an automaton for inequalities $a_1x_1 + \ldots + a_nx_n \leq b$. The computation is similar: we start with q_b and, from a state q_c we compute the transitions and states as follows. For every bit vector $(\theta_1, \ldots \theta_n)$, $q_c \xrightarrow{\theta_1 \ldots \theta_n} q_d$ with

$$d = \lfloor \frac{c - \sum_{i=1}^{n} a_i\theta_i}{2} \rfloor$$

$Q_f = \{q_c \mid c \geq 0\}$.

Example 11. Consider the inequality $2x - y \leq -1$. The automaton which we get using the algorithm is displayed on figure 2.10.

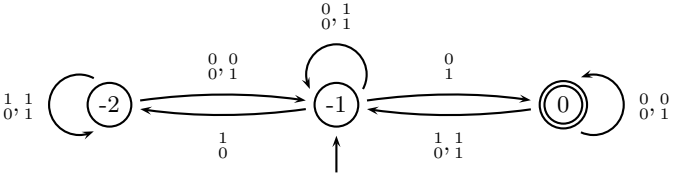


Fig. 2.10. The automaton accepting the solutions of $2x - y \leq -1$

Exercise 2. What is the largest number of reachable states in the automaton for $a_1x_1 + \ldots + a_nx_n \leq b$?

Closure properties and automata for arbitrary formulas.

Let $\mathcal{A}_\phi(x_1, \ldots, x_n)$ be an automaton over the alphabet $\{0,1\}^n$ which accepts the solutions of the formula ϕ whose free variables are contained in $\{x_1, \ldots, x_n\}$. If a variable x_i does not occur free in ϕ, then accepting/not accepting a word will not depend on what we have on the ith track.

Exercise 3. Show how to compute $\mathcal{A}_\phi(x_1, \ldots, x_n, y)$, given $\mathcal{A}_\phi(x_1, \ldots, x_n)$.

Given two automata $\mathcal{A}_{\phi_1}(\overrightarrow{x})$ and $\mathcal{A}_{\phi_2}(\overrightarrow{x})$ we can compute $\mathcal{A}_{\phi_1 \wedge \phi_2}(\overrightarrow{x})$ and $\mathcal{A}_{\phi_1 \vee \phi_2}(\overrightarrow{x})$ by the classical constructions for intersection and union of finite automata. We only have to be sure that $\overrightarrow{x}$ contains the free variables of both ϕ_1 and ϕ_2, which can be easily be satisfied, thanks to the above exercise.

Then negation corresponds to the complement, a classical construction which is linear when the automaton is deterministic (and may be exponential otherwise). Since the automata which were constructed for atomic formulas are deterministic, computing the automaton for $a_1x_1 + \ldots + a_nx_n \neq b$ is easy.

Now, it remains to handle the quantifiers. Since $\forall x.\phi$ is logically equivalent to $\neg\exists x.\neg\phi$, we only have to consider existential quantification. $\mathcal{A}_{\exists x.\phi}$ is computed from $\mathcal{A}_\phi$ by *projection*: the states, final states, initial state and transitions of $\mathcal{A}_{\exists x.\phi}$ are identical to those of $\mathcal{A}_\phi$, except that we forget the track corresponding to x in the labels of the transitions.

Example 12. Consider the formula $\exists z.x + 2y = 3z + 1$. The automaton is obtained from the automaton of figure 2.9, forgetting about z. This yields the automaton of figure 2.11. By chance, this automaton is still deterministic.

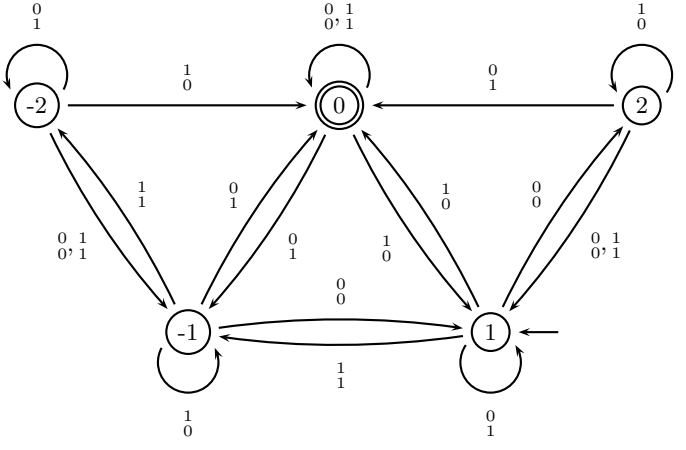


Fig. 2.11. The automaton accepting the solutions of $\exists z.x + 2y = 3z + 1$

But this is not the case in general.

Finally, by induction on the Presburger formula ϕ, we can compute an automaton which accepts the set of solutions of ϕ.

Deciding the satisfiability of a formula ϕ then reduces to decide the emptiness of $\mathcal{A}_\phi$, which can be done in linear time w.r.t. the number of states.

Consider however that $\mathcal{A}_\phi$ could have, in principle, a number of states which is a tower of exponentials whose height is the number of quantifier alternations in ϕ. This is because each quantifier alternation may require a projection followed by a complement and the projection may yield a non-deterministic automaton, requiring an exponential blow-up for the complement. Whether this multi-exponential blow-up may occur in the above construction is an open question. (Though the complexity of Presburger arithmetic is known and elementary, using other methods. See [FR79].)

2.9 Typing Constraints and Tree Automata

If we want to solve constraints on terms, we need another device, which accepts not only words, but also trees. We introduce tree automata on a very simple constraint system. Some other applications are sketched in sections 2.11.1, 2.11.2, 2.11.4. We refer the interested reader to [CDG+97], a book freely available on the web, which contains much more material, both on theory and on applications to tree automata to constraint solving.

2.9.1 Tree Automata

Recognizing a tree is not very different from recognizing a word: a tree t is accepted if there is a successful run of the automaton on t. The main difference is that transitions send some (possibly different) states to all the successors of a node instead of sending a single state to the next position.

Here, we will only consider automata on finite trees, though there are very interesting developments and applications in the case of infinite trees (see e.g. [Tho90]).

We assume that $\mathcal{F}$ is a finite ranked alphabet (each symbol has a fixed arity) and $\mathcal{T}(\mathcal{F})$ is the set of (ground) terms built on $\mathcal{F}$. A finite (bottom-up) tree automaton is a triple (Q, Q_f, δ) where Q is a finite set of states, Q_f is a subset of final states and δ is a set of rules of the form $f(q_1, \ldots, q_n) \to q$ where $q_1, \ldots, q_n, q \in Q$ and $f \in \mathcal{F}$ has arity n. A *run* ρ of $\mathcal{A}$ on a term (or tree) $t \in \mathcal{T}(\mathcal{F})$ is a tree which has the same set of positions as t, whose labels are in Q and such that, for every position p of t, if $t(p) = f$ has arity n, $\rho(p) = q$ and $\rho(p \cdot 1) = q_1, \ldots \rho(p \cdot n) = q_n$, then there is a transition $f(q_1, \ldots, q_n) \to q$ in δ. A run is *successful* if its root is labeled with a finite state. A term t is *recognized* by a tree automaton $\mathcal{A}$ if there is a successful run of $\mathcal{A}$ on t.

Example 13. Using a notation which is more common in algebraic specifications, we define below an order-sorted signature:

SORTS:$\mathsf{nat}, \mathsf{int}$
SUBSORTS : $\mathsf{nat} \leq \mathsf{int}$
FUNCTION DECLARATIONS:

$$
\begin{array}{ll}
0 : & \to \mathsf{nat} \\
+ : & \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\
s : & \mathsf{nat} \to \mathsf{nat} \\
p : & \mathsf{nat} \to \mathsf{int} \\
+ : & \mathsf{int} \times \mathsf{int} \to \mathsf{int} \\
abs : & \mathsf{int} \to \mathsf{nat} \\
fact : & \mathsf{nat} \to \mathsf{nat} \\
\ldots &
\end{array}
$$

This can be seen as a finite tree automaton whose states are the sorts $\{\mathsf{nat}, \mathsf{int}\}$. Function declarations can easily be translated into automata rules, e.g. $+(\mathsf{nat}, \mathsf{nat}) \to \mathsf{nat}$. The subsort ordering corresponds to ϵ-transitions which can easily be eliminated, yielding

$$\begin{array}{rlrl} 0 & \to \mathsf{nat} & 0 & \to \mathsf{int} \\ +(\mathsf{nat},\mathsf{nat}) & \to \mathsf{nat} & +(\mathsf{nat},\mathsf{nat}) & \to \mathsf{int} \\ s(\mathsf{nat}) & \to \mathsf{nat} & s(\mathsf{nat}) & \to \mathsf{int} \\ p(\mathsf{nat}) & \to \mathsf{int} & +(\mathsf{int},\mathsf{int}) & \to \mathsf{int} \\ abs(\mathsf{int}) & \to \mathsf{nat} & abs(\mathsf{int}) & \to \mathsf{int} \\ fact(\mathsf{nat}) & \to \mathsf{nat} & fact(\mathsf{nat}) & \to \mathsf{int} \\ \multicolumn{4}{c}{\ldots} \end{array}$$

The set of terms accepted by the automaton in state nat (resp int) is the set of terms of *sort* nat.

Basically, recognizable tree languages have the same properties as recognizable word languages: closure by Boolean operations, decidability of emptiness, finiteness etc... See [CDG+97] for more details.

2.9.2 Membership Constraints

The set of *sort expressions* $\mathcal{SE}$ is the least set such that

- $\mathcal{SE}$ contains a finite set of *sort symbols* S, including the two particular symbols $\top_S$ and $\bot_S$.
- If $s_1, s_2 \in \mathcal{SE}$, then $s_1 \vee s_2$, $s_1 \wedge s_2$, $\neg s_1$ are in $\mathcal{SE}$
- If $s_1, \ldots, s_n$ are in $\mathcal{SE}$ and f is a function symbol of arity n, then $f(s_1, \ldots, s_n) \in \mathcal{SE}$.

The atomic formulas are expressions $t \in s$ where $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $s \in \mathcal{SE}$. The formulas are arbitrary first-order formulas built on these atomic formulas.

These formulas are interpreted as follows: we are giving an order-sorted signature (or a tree automaton) whose set of sorts is S. We define the interpretation$[\![\cdot]\!]_S$ of sort expressions as follows:

- if $s \in S$, $[\![s]\!]_S$ is the set of terms in $\mathcal{T}(\mathcal{F})$ that are accepted in state s.
- $[\![\top_S]\!]_S = \mathcal{T}(\mathcal{F})$ and $[\![\bot_S]\!]_S = \emptyset$
- $[\![s_1 \vee s_2]\!]_S = [\![s_1]\!]_S \cup [\![s_2]\!]_S$, $[\![s_1 \wedge s_2]\!]_S = [\![s_1]\!]_S \cap [\![s_2]\!]_S$, $[\![\neg s]\!]_S = \mathcal{T}(\mathcal{F}) \setminus [\![s]\!]_S$
- $[\![f(s_1, \ldots, s_n)]\!]_S = \{f(t_1, \ldots, t_n) \mid t_1 \in [\![s_1]\!]_S, \ldots t_n \in [\![s_n]\!]_S\}$

Example 14. Consider the specification:

$$\begin{array}{rlrl} 0 & \to \mathsf{even} & s(\mathsf{odd}) & \to \mathsf{even} \\ s(\mathsf{even}) & \to \mathsf{odd} & \mathsf{even} + \mathsf{odd} & \to \mathsf{odd} \\ \mathsf{odd} + \mathsf{even} & \to \mathsf{odd} & \mathsf{even} + \mathsf{even} & \to \mathsf{even} \\ \mathsf{odd} + \mathsf{odd} & \to \mathsf{even} & & \end{array}$$

$[\![\mathsf{odd} \wedge \mathsf{even}]\!]$ is empty: this can be shown by constructing the intersection automaton whose rules only yielding $\mathsf{odd} \wedge \mathsf{even}$ consist of

$$\begin{array}{rl} s(\mathsf{odd} \wedge \mathsf{even}) \to \mathsf{odd} \wedge \mathsf{even} & \mathsf{odd} \wedge \mathsf{even} + \mathsf{even} \to \mathsf{odd} \wedge \mathsf{even} \\ \mathsf{odd} \wedge \mathsf{even} + \mathsf{odd} \to \mathsf{odd} \wedge \mathsf{even} & \mathsf{odd} + \mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd} \wedge \mathsf{even} \\ \mathsf{even} + \mathsf{odd} \wedge \mathsf{even} \to \mathsf{odd} \wedge \mathsf{even} & \end{array}$$

which shows that $\mathsf{odd} \wedge \mathsf{even}$ is unreachable.

2.9.3 From the Constraints to the Automata

It is easy to associate with each membership constraint $x \in \zeta$ a tree automaton which accepts the solutions of the constraint: simply consider an automaton for $[\![\zeta]\!]$. Now, for arbitrary constraints, we need a decomposition lemma showing how $f(t_1, \ldots, t_n) \in \zeta$ can be decomposed into a finite disjunction of constraints $t_1 \in \zeta_1 \wedge \ldots \wedge t_n \in \zeta_n$. This is not difficult (though a bit long, see [CD94, Com98] for more details and extensions). Then every atomic constraint is a membership constraint $x \in \zeta$. And we use the closure properties of tree automata to get an automaton which accepts the solutions of an arbitrary constraint.

Note that, in this very simple example of application, we do not need to stack (or, more realistically, to overlap) the different tracks, since variables are "independent". In other words, the resulting automaton, which accepts tuples of terms, is actually a product of tree automata (each working on a single track). This is not the case for more complicated constraints such as rewriting constraints (see [CDG+97, DT90]). For instance the solutions of $x = y$ are recognized by a finite automaton on pairs (simply check that all labels are pairs of identical symbols) but is not recognized by a product of two automata.

2.9.4 The Monadic Second Order Logics

The constraint system of the previous section can be embedded in a well-known more powerful system: the weak monadic second-order logic. It is beyond the scope of these notes to study these logics. Let us simply briefly recall the constraint systems and the main results.

Atomic formulas are $x = y \cdot i$ where $i \in \{1, ...n\}$, $x = y$, $x = \epsilon$ or membership constraints $x \in X$ where x, y, X are variables. Upper case letters are used here for monadic second order variables, while lower case letters are first-order variables. The formulas are built over these atomic constraints using Boolean connectives and quantifications, both over the first-order and the second-order variables.

The formulas are interpreted as follows: first-order variables range over words in $\{1, ..., n\}^*$ (ϵ is the empty word) and second-order variables range over finite (resp. finite or infinite) subsets of $\{1, ..., n\}^*$. In the finite (resp.

finite or infinite) case, the constraint system is called *weak monadic second order logic with n successors* (resp. *monadic second order logic with n successors*), $WSnS$ for short (resp. SnS).

The main results state on one hand that the solutions of atomic formulas are recognized by appropriate tree automata and on the other hand that the class of automata is closed under Boolean operations and projections and emptiness is decidable. This yields the decidability of the constraint system.

More precisely, we use an encoding similar to the encoding which was presented in section 2.8: with each k-uple of subsets of $\{1, ..., n\}^*$, we associate an infinite tree of degree n labeled with vectors in $\{0, 1\}^k$. Assume for instance that there are three sets S_1, S_2, S_3. If a path $w \in \{1, ..., n\}^*$ of the tree yields a node labeled with e.g. $\begin{smallmatrix}1\\0\\1\end{smallmatrix}$, this means that $w \in S_1$, $w \notin S_2$ and $w \in S_3$. Hence solutions can be seen as trees labeled with $\{0, 1\}^k$. Then

Theorem 7 ([TW68]). *The set of solutions of a constraint in WSnS is accepted by a finite tree automaton.*

Theorem 8 ([Rab69]). *The set of solution of a constraint in SnS is accepted by a Rabin tree automaton.*

Both results are shown by first proving that the solutions of atomic constraints are recognized by an appropriate automaton and then proving the closure properties of the class of automata.

Such closure properties are easy, except in one case which is really non trivial: the complement for Rabin automata. We did not define so far Rabin automata (see e.g. [Tho97, Rab77] for surveys). They work on infinite trees, hence top-down. On the other hand, it is not possible to determinize top-down tree automata (this is left as an exercise). Hence the classical complementation construction does not work for automata on infinite trees.

Such general results can be applied to several constraint systems which can be translated into monadic second-order logics. A typical example is rewriting constraints (see [DHLT88]). A more detailed description of applications can be found in [CDG+97].

2.9.5 Extensions of the Constraint System and Further Results

The simple constraint system of section 2.9.2 can be extended in several directions. For instance, it is possible to add equations between first-order terms [CD94] or to consider (in some restricted way) second-order terms [Com98].

Solving constraints which combine equations and membership predicates is also known as *order-sorted unification* and deserved several works which use syntactic constraint solving methods [MGS90, HKK98].

The membership constraints have also been extended to other models with applications to automated deduction (see e.g. [GMW97, JMW98]). The

device (finite tree automata) has however to be extended, along the lines described in section 2.11.4.

Parametric specifications can also be handled, but we need then the set constraints of the next section, since we have also to find which values of the parametric types are solutions of the constraint.

Tree automata (on infinite trees) are used in [TW93] in the context of type reconstruction. Here constraints are interpreted over the type structures, not over the term structure.

2.10 Set Constraints and Tree Set Automata

There has been recently a lot of work on set constraints and, in particular on applications of automata techniques. Let us cite among others [BGW93, GTT93b, GTT93a, GTT94, Tom94].

We do not intend to survey the results. Our purpose is to show how automata may help to understand and solve such constraints.

2.10.1 Set Constraints

We consider only the simplest version of set constraints. The *set expressions* are built from set variables, intersection, union, complement and application of a function symbol (out of a finite ranked alphabet) to set expressions. Then the constraints are conjunctions of inclusion constraints $e \subseteq e'$ where e, e' are set expressions.

Set expressions are interpreted as subsets of $\mathcal{T}(\mathcal{F})$: intersection, union and complement are interpreted as expected. If f is a function symbol, then $f(e_1, \ldots, e_n)$ is interpreted as the set of terms $f(t_1, \ldots, t_n)$ where, for every i, t_i is in the interpretation of e_i.

Example 15. In this example, we define the lists L_S of elements in S as a solution of the classical fixed point equations. Since S is arbitrary, we consider another instance of the definition, where S is replaced with L_S itself. Then we are looking for the sets of trees S such that the lists over S are also lists of lists of elements in S.

$$\begin{aligned} L_S &= nil \cup cons(S, L_S) \\ L_{L_S} &= nil \cup cons(L_S, L_{L_S}) \\ N &= 0 \cup s(N) \\ L_S &= L_{L_S} \end{aligned}$$

A solution of this constraint is the set $S = \{nil\}$ since L_S then consists of lists of any number of nil's which are also lists of lists of nil's. What are all the solutions?

2.10.2 Tree Set Automata

We follow here the definitions of [Tom94]. Extensions can be found in e.g. [CDG+97].

A *tree set automaton* (on the ranked alphabet $\mathcal{F}$) is a tuple (Q, A, Ω, δ) where Q is a finite set of states, A is a tuple $(A_1, \ldots, A_m)$ of final states, $\Omega \subseteq 2^Q$ is a set of accepting sets of states and δ is a set of rules $f(q_1, \ldots, q_n) \to q$ where $f \in \mathcal{F}$ has arity n and $q_1, \ldots, q_n, q$ are states.

A *run* of a tree set automaton is a mapping from $\mathcal{T}(\mathcal{F})$ into Q such that, if $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F})$, then there is a rule $f(\rho(t_1), \ldots, \rho(t_n)) \to \rho(f(t_1, \ldots, t_n))$ in δ. A run is *successful* if $\rho(\mathcal{T}(\mathcal{F})) \in \Omega$.

The *language recognized* by a tree set automaton is the set of tuples of sets $(L_1, \ldots, L_m)$ such that there is a successful run ρ such that, for every i, $L_i = \{t \in \mathcal{T}(\mathcal{F}) \mid \rho(t) \in F_i\}$.

We will see an example below.

2.10.3 From Constraints to Automata

We consider a single constraint $e = e'$ (the case $e \subseteq e'$ is similar). Let Q be the set of mappings from the set of subexpressions (with top symbol in $\mathcal{F}$ or $\mathcal{X}$) occurring in c into $\{0, 1\}$. Such mappings are extended to all subexpressions using the usual Boolean rules. They are represented as bit-vectors. For convenience (in the BDD style) we use an x instead of $0, 1$ when the value is not relevant (i.e. both rules are present, for each Boolean value). If there are n free variables $X_1, \ldots, X_n$, then $A = (A_1, \ldots, A_n)$ with $A_i = \{\phi \in Q \mid \phi(X_i) = 1\}$. δ consists of the rules $f(\phi_1, \ldots, \phi_k) \to \phi$ such that, for every subexpression e which is not a variable, $\phi(e) = 1$ iff $(e = f(e_1, \ldots, e_k)$ and $\forall i, \phi_i(e_i) = 1)$. Ω is the set of $\omega \subseteq Q$ such that, for every $\phi \in \omega$, $\phi(e) = 1$ if and only if $\phi(e') = 1$.

Example 16. Consider first the equation $L_S = nil \cup cons(S, L_S)$. We have $Q = 2^{\{S, L_S, cons(S,L_S), nil\}}$. The transitions are

$$\begin{aligned} nil &\to (\mathsf{x}, \mathsf{x}, 0, 1) \\ cons((1, \mathsf{x}, \mathsf{x}, \mathsf{x}), (\mathsf{x}, 1, \mathsf{x}, \mathsf{x})) &\to (\mathsf{x}, \mathsf{x}, 1, 0) \\ f(...) &\to (\mathsf{x}, \mathsf{x}, 0, 0) \text{ for any } f \notin \{cons, nil\} \end{aligned}$$

Ω consists of the subsets of

$$\bigcup_{x_1, x_2, x_3, x_4 \in \{0,1\}} \{(x_1, 0, 0, 0), (x_2, 1, 0, 1), (x_3, 1, 1, 0), (x_4, 1, 1, 1)\}.$$

Consider for instance the following mappings from $\mathcal{T}(\mathcal{F})$ to Q:

$\rho_1(t) = (0, 0, 0, 0)$ for all t
$\rho_2(t) = (0, 0, 0, 0)$ for all $t \neq nil$ and $\rho_2(nil) = (0, 0, 0, 1)$
$\rho_3(t) = (0, 0, 0, 0)$ for all $t \neq nil$ and $\rho_2(nil) = (0, 1, 0, 1)$

ρ_1 is not a run since it is not compatible with the first transition rule. ρ_2 is a run which is not successful since $(0,0,0,1) \in \rho_2(\mathcal{T}(\mathcal{F}))$ and $(0,0,0,1) \notin \omega$ for any $\omega \in \Omega$. ρ_3 is a successful run.

Let $A_1 = \{(1,\mathsf{x},\mathsf{x},\mathsf{x})\}$, $A_2 = \{(\mathsf{x},1,\mathsf{x},\mathsf{x})\}$. For instance $(\emptyset, \{nil\})$ is accepted by the automaton, as well as $(T(\{cons, nil\}), T(\{cons, nil\}))$.

Exercise 4. In the previous example, what are all the successful runs such that $\rho(\mathcal{T}(\mathcal{F}))$ is minimal (w.r.t. inclusion)?

Note that we may simplify the construction: we may only consider states which belong to some $\omega \in \Omega$: otherwise, we will not get a successful run. For instance in our example, the set of rules may be restricted to

$$\begin{aligned} nil &\to (\mathsf{x},1,0,1) \\ cons((1,\mathsf{x},\mathsf{x},\mathsf{x}),(\mathsf{x},1,\mathsf{x},\mathsf{x})) &\to (\mathsf{x},1,1,0) \\ f(...) &\to (\mathsf{x},0,0,0) \text{ for every } f \notin \{cons, nil\} \end{aligned}$$

Another remark is that Ω is subset closed, when it is derived from a (positive) set constraint: if $\omega \in \Omega$ and $\omega' \subseteq \omega$, then $\omega' \in \Omega$. Tree set automata which have this property are called *simple*. There is yet another particular property of tree set automata which can be derived from the construction: once we fix the components of the state which correspond to set variables, the rules becomes deterministic. In other words, if two rules applied to t yield states q_1, q_2, then q_1, q_2 differ only in the components corresponding to set variables. Such automata belong to a restricted class called *deterministic tree set automata*. The above construction shows that:

Proposition 9. *The solutions of an atomic set constraints are recognized by a deterministic simple tree set automaton.*

Now we may take advantage of closure of tree set automata:

Theorem 9 ([Tom94, GTT94]). *The class of languages recognized by tree set automata is closed by intersection, union, projection, inverse projection.*

The class of languages recognized by simple deterministic tree set automata is closed under all Boolean operations.

as well as decision properties:

Theorem 10 ([Tom94, GTT94]). *Emptiness decision is NP-complete for the class of (simple) tree set automata.*

Finally, if there is a solution (tuples of sets of terms) to a set constraint, then there is one which consists of regular sets only.

Example 17. Let us come back to example 15.
$Q = 2^{\{S, L_S, L_{L_S}, nil, cons(S,L_S), cons(L_S,L_{L_S})\}}$ and
$\Omega = 2^{\{(\mathsf{x},0,0,0,0,0),(\mathsf{x},1,1,0,1,1),(\mathsf{x},1,1,1,0,0)\}}$.

Since we may only consider rules which belong to some member of Ω, there are actually few rules:

$$\begin{aligned} nil &\to (\mathsf{x},1,1,1,0,0)\\ cons((1,1,\mathsf{x},\mathsf{x},\mathsf{x},\mathsf{x}),(\mathsf{x},1,1,\mathsf{x},\mathsf{x},\mathsf{x})) &\to (\mathsf{x},1,1,0,1,1)\\ cons((0,0,0,0,0,0),(\mathsf{x},\mathsf{x},\mathsf{x},\mathsf{x},\mathsf{x},\mathsf{x})) &\to (\mathsf{x},0,0,0,0,0)\\ cons((\mathsf{x},\mathsf{x},\mathsf{x},\mathsf{x},\mathsf{x},\mathsf{x}),(\mathsf{x},0,0,0,0,0)) &\to (\mathsf{x},0,0,0,0,0)\\ f(...) &\to (\mathsf{x},0,0,0,0,0) \end{aligned}$$

Let ρ be the mapping such that $\rho(t) = (1,1,1,0,1,1)$ if $t \in T(\{nil, cons\})$, $\rho(nil) = (0,1,1,1,0,0)$ and $\rho(t) = (0,0,0,0,0,0)$ otherwise. ρ is a successful run.

Exercise 5. What is the solution corresponding to the run ρ? Are there any other successful runs?

2.11 Examples of Other Constraint Systems Using Tree Automata

2.11.1 Rigid E-Unification

A unification problem is a conjunction of equations between terms (as previously seen). Given a set of equations E, a solution of the rigid E-unification problem P is a (ground) substitution σ such that $E\sigma \vdash P\sigma$. Note that, in contrast with E-unification, only one instance of each equation in E can be used in the proof. Such problems play crucial roles in automated deduction (see e.g. [GRS87, DMV96]).

The crucial lemma in [Vea97] is that the set of solutions of a single rigid E-unification problem with one variable is recognized by a finite tree automaton. The following result is then a consequence (at least the EXPTIME membership), computing the intersection of n tree automata:

Theorem 11 ([Vea97]). *The simultaneous rigid E-unification problem with one variable is EXPTIME-complete.*

There are other applications of tree automata and tree grammars to unification problems, see for instance [LR97] and there are also other applications of tree automata to related problems (e.g. [CGJV99])

2.11.2 Higher-Order Matching Problems

We consider here simply typed λ terms. A *higher-order matching problem* is a (conjunction of) equation(s) $s = t$ where s, t are simply-typed λ-terms built over an alphabet of typed constants and variables, such that t does not contain any free variables. A solution consists in an assignment σ to the free

variables of s such that $s\sigma$ and t are β-equivalent. This decidability of higher-order matching is still an open question. It has been shown decidable at order 3 [Dow93] and at order 4 [Pad96]; the *order* of a type is defined by $o(a) = 1$ if a is an atomic type and $o(\tau_1, \ldots, \tau_n \to \tau) = \max(o(\tau), 1+o(\tau_1), ..., 1+o(\tau_n))$. The order of a problem is the maximal order of the type of a free variable occurring in it.

An *interpolation equation* is a higher-order matching problem of the form $x(t_1, \ldots, t_n) = u$ where $t_1, \ldots, t_n, u$ do not contain any free variables. As shown in [Pad96], a matching problem of order n is equivalent to a Boolean combination of interpolation equations of order n.

The crucial property in [CJ97b], as before, is that the set of solutions of an interpolation equation $x(t_1, \ldots, t_n) = u$ where x is the only free variable, is recognized by a (slight variant of a) finite tree automaton, provided that the order of x is at most 4. It follows, thanks to the closure properties of tree automata that:

Theorem 12 ([CJ97b]). *The set of solutions of a 4th order matching problem is recognized by a (effectively computable) finite tree automaton.*

This implies of course the decidability of 4th order matching.

2.11.3 Feature Constraints

We mention here an application to feature constraints [MN98]. Since feature constraints are interpreted over infinite trees, we need a device which works on infinite trees. It is beyond the scope of this course to introduce automata on infinite objects (see e.g. [Tho90]).

The constraints consist of conjunctions of atomic constraints which are of the form $x \leq x'$, $x[f]x'$ or $a(x)$.

A *feature tree* is a finite or infinite tree in which the order of the sons is irrelevant, but each son of a node is labeled with a *feature* and each feature labels at most one son of each node (see e.g. [Tre97] for more details). Feature trees have been introduced as a record-like data structure in constraint logic programming (see e.g. [ST94]) and are also used as models in computational linguistics. The reader is referred to [Tre97] for more references of the subject.

The semantics of the constraints is given by an interpretation over feature trees, inequalities mean roughly that a tree is "more defined" than another tree. For instance it may include new features. $x[f]x'$ is satisfied by a pair of trees (t, t') if t' is a son of t, at feature f. $a(x)$ is satisfied by a tree t if the root of t is labeled with a.

The crucial part of the result in [MN98] is the following:

Theorem 13 ([MN98]). *The set of solutions of an constraint over sufficiently labeled feature trees is recognized by a finite tree automaton.*

Actually, the authors show a translation into some monadic second order logic; this is an equivalent formulation thanks to Rabin's theorem [Rab69] (see also [Tho97] for a comprehensive proof), or Thatcher and Wright's theorem if one considers finite trees only [TW68] (see also [CDG+97]).

It follows for instance from this theorem that the entailment of existentially quantified feature constraints is decidable.

2.11.4 Encompassment Constraints

Encompassment constraints are built on arbitrarily many atomic predicates of the form $encomp_t(x)$. Where t is a term. Such a predicate is interpreted as the set of terms which *encompass* t. A term u encompasses v when an instance of v is a subterm of u.

It is known for a long time that, when t is a linear term (i.e. each variable occurs at most once in t), then the solutions $encomp_t(x)$ are recognized by a finite tree automaton, hence the first-order theory of such predicates is decidable, thanks to closure properties of tree automata.

When t is not linear, we need an extension of tree automata. Such an extension is defined and studied in [DCC95]: a *reduction automaton* is defined as a finite tree automaton, except that each transition may check equalities and/or disequalities between subtrees at fixed positions. For instance a rule $f(q_1, q_2) \xrightarrow{12=21 \wedge 11 \neq 22} q$ states that the transition can be applied to $f(t_1, t_2)$ only if t_1 can reach q_1, t_2 can reach q_2 and, moreover, the subterm of t_1 at position 2 equals the subterm of t_2 at position 1 and the subterm of t_1 at position 1 is distinct from the subterm of t_2 at position 2. Such a computation model is, in general, too expressive. Hence reduction automata also impose an ordering on the states such that, if there is at least one equality checked by the transition, then the state is decreasing. For instance in the above rule, we must have $q_1 > q$ and $q_2 > q$.

We follow our leitmotiv: first it is possible to associate each atomic constraint with a reduction automaton which accepts all its solutions, then we have the closure and decision properties for these automata:

Theorem 14 ([DCC95]).

1. *There is a (effectively computable) deterministic and complete reduction automaton which accepts the solutions of* $encomp_t(x)$.
2. *The class of languages accepted by deterministic reduction automata is effectively closed under Boolean operations.*
3. *The emptiness problem for reduction automata is decidable.*

It follows in particular that the first-order theory of encompassment predicates is decidable. This implies in particular the decidability of a well-known problem in rewriting theory: the ground reducibility, which can be expressed as a formula in this theory.

There are a lot of further works on automata with equality and disequality tests, e.g. [BT92,LM94,CCC+94,CJ97a,JMW98]. See [CDG+97] for a survey of the main results and applications.

Acknowledgments

This document beneficiated of works and papers written with several colleagues, in particular Jean-Pierre Jouannaud, Hélène Kirchner and Christophe Ringeissen.

References

[AK92] M. Adi and Claude Kirchner. Associative commutative matching based on the syntacticity of the ac theory. In Franz Baader, J. Siekmann, and W. Snyder, editors, *Proceedings 6th International Workshop on Unification, Dagstuhl (Germany)*. Dagstuhl seminar, 1992.

[Baa91] F. Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, April 1991.

[Büc60] J.R. Büchi. On a decision method in restricted second-order arithmetic. In E. Nagel et al., editor, *Proc. Int. Congr. on logic, methodology and philosophy of science*, Standford, 1960. Stanford Univ. Press.

[BC96] Alexandre Boudet and Hubert Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Proc. Coll. on Trees in Algebra and Programming (CAAP'96)*, Lecture Notes in Computer Science, 1996.

[BCD+98] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.3 User Manual*. LORIA, Nancy (France), third edition, December 1998.

[BGLS95] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[BGW93] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Set constraints are the monadic class. In *Proc. 8th IEEE Symp. Logic in Computer Science, Montréal*, 1993.

[BHSS89] H.-J. Bürckert, A. Herold, and M. Schmidt-Schauß. On equational theories, unification and decidability. *Journal of Symbolic Computation*, 8(1 & 2):3–50, 1989. Special issue on unification. Part two.

[BKK+98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume16.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[Boc87] A. Bockmayr. A note on a canonical theory with undecidable unification and matching problem. *Journal of Automated Reasoning*, 3(1):379–381, 1987.

[BS86] R. V. Book and J. Siekmann. On unification: Equational theories are not bounded. *Journal of Symbolic Computation*, 2:317–324, 1986.

[BS99] F. Baader and W. Snyder. Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. To appear.

[BT92] B. Bogaert and Sophie Tison. Equality and disequality constraints on brother terms in tree automata. In A. Finkel, editor, *Proc. 9th Symp. on Theoretical Aspects of Computer Science*, Paris, 1992. Springer-Verlag.

[Bür89] H.-J. Bürckert. Matching — A special case of unification? *Journal of Symbolic Computation*, 8(5):523–536, 1989.

[BVW94] O. Bernholtz, M. Vardi, and P. Wolper. Au automata-theoretic approach to branching-time model checking. In *Proc. 6th Int. Conf. on Computer Aided verification*, volume 818 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Cas98] Carlos Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.

[CCC+94] A.-C. Caron, H. Comon, J.-L. Coquidé, M. Dauchet, and F. Jacquemard. Pumping, cleaning and symbolic constraints solving. In *Proc. Int. Conference on Algorithms, Languages and Programming*, Lecture Notes in Computer Science, vol. 820, Jerusalem, July 1994. Springer-Verlag.

[CCD93] Anne-Cécile Caron, Jean-Luc Coquide, and Max Dauchet. Encompassment properties and automata with constraints. In Claude Kirchner, editor, *Rewriting Techniques and Applications, 5th International Conference, RTA-93*, LNCS 690, pages 328–342, Montreal, Canada, June 16–18, 1993. Springer-Verlag.

[CD94] Hubert Comon and Catherine Delor. Equational formulae with membership constraints. *Information and Computation*, 112(2):167–216, August 1994.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. A preliminary version of this unpublished book is available on `http://l3ux02.univ-lille3.fr/tata` , 1997.

[CDJK99] Hubert Comon, Mehmet Dincbas, Jean-Pierre Jouannaud, and Claude Kirchner. A methodological view of constraint solving. *Constraints*, 4(4):337–361, December 1999.

[CF92] H. Comon and M. Fernández. Negation elimination in equational formulae. In *Proc. 17th International Symposium on Mathematical Foundations of Computer Science (Praha)*, Lecture Notes in Computer Science, 1992.

[CGJV99] V. Cortier, H. Ganzinger, F. Jacquemard, and V Veanes. Decidable fragments of simultaneous rigid reachability. In *Proc. ICALP'99*, 1999. To appear in ICALP'99.

[Cha94] Jacques Chabin. *Unification Générale par Surréduction Ordonnée Contrainte et Surréduction Dirigée.* Thèse de Doctorat d'Université, Université d'Orléans, January 1994.

[CHJ94] Hubert Comon, Marianne Haberstrau, and Jean-Pierre Jouannaud. Syntacticness, cycle-syntacticness and shallow theories. *Information and Computation*, 111(1):154–191, May 1994.

[Chu62] A. Church. Logic, arithmetic, automata. In *Proc. International Mathematical Congress*, 1962.

[CJ94] H. Comon and F. Jacquemard. Ground reducibility and automata with disequality constraints. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen (France)*, volume 775 of *Lecture Notes in Computer Science*, pages 151–162. Springer-Verlag, February 1994.

[CJ97a] Hubert Comon and Florent Jacquemard. Ground reducibility is exptime-complete. In *Proc. IEEE Symp. on Logic in Computer Science*, Varsaw, June 1997. IEEE Comp. Soc. Press.

[CJ97b] Hubert Comon and Yan Jurski. Higher-order matching and tree automata. In M. Nielsen and W. Thomas, editors, *Proc. Conf. on Computer Science Logic*, volume 1414 of *LNCS*, pages 157–176, Aarhus, August 1997. Springer-Verlag.

[CL89] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3 & 4):371–426, 1989. Special issue on unification. Part one.

[Col82] A. Colmerauer. PROLOG II, manuel de référence et modèle théorique. Technical report, GIA, Université Aix-Marseille II, 1982.

[Com86] H. Comon. Sufficient completeness, term rewriting system and anti-unification. In J. Siekmann, editor, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer-Verlag, 1986.

[Com88] H. Comon. *Unification et disunification. Théories et applications.* Thèse de Doctorat d'Université, Institut Polytechnique de Grenoble (France), 1988.

[Com89] Hubert Comon. Inductive proofs by specification transformations. In Nachum Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 76–91, Chapel Hill, NC, April 1989. Vol. 355 of *Lecture Notes in Computer Science*, Springer, Berlin.

[Com90] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Sciences*, 1(4):387–411, 1990.

[Com91] H. Comon. Disunification: a survey. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9, pages 322–359. The MIT press, Cambridge (MA, USA), 1991.

[Com93] Hubert Comon. Complete axiomatizations of some quotient term algebras. *Theoretical Computer Science*, 118(2):167–191, September 1993.

[Com98] H. Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *Journal of Symb. Computation*, 25:421–453, 1998. This is the second part of a paper whose abstract appeared in Proc. ICALP 92, Vienna.

[CT94] Hubert Comon and Ralf Treinen. Ordering constraints on trees. In Sophie Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, Scotland, April 1994. Springer Verlag.

[Dau94] M. Dauchet. Symbolic constraints and tree automata. In Jouannaud [Jou94], pages 217–218.

[DCC95] Dauchet, Caron, and Coquidé. Automata for reduction properties solving. *JSCOMP: Journal of Symbolic Computation*, 20, 1995.

[DH95] N. Dershowitz and C. Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, May 1995.

[DHK98] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz`.

[DHLT88] Max Dauchet, Thierry Heuillard, Pierre Lescanne, and Sophie Tison. The confluence of ground term rewriting systems is decidable. In *Proc. 3rd IEEE Symp. Logic in Computer Science, Edinburgh*, 1988.

[DJ90a] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[DJ90b] D. Dougherty and P. Johann. An improved general E-unification method. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449 of *Lecture Notes in Computer Science*, pages 261–275. Springer-Verlag, July 1990.

[DKM84] C. Dwork, P. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[DMR76] M. Davis, Y. Matijasevič, and J. A. Robinson. Hilbert's tenth problem: Positive aspects of a negative solution. In *F. E. Browder, Editor, Mathematical Developments Arising from Hilbert Problems, American Mathematical Society*, pages 323–378, 1976.

[DMV96] Anatoli Degtyarev, Yuri Matiyasevich, and Andrei Voronkov. Simultaneous reigid e-unification and related algorithmic problems. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 494–502. IEEE Comp. Soc. Press, 1996.

[Dow93] Gilles Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 1993.

[DT90] Max Dauchet and Sophie Tison. The theory of ground rewrite systems is decidable. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, 1990.

[Eke95] Steven Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[Elg61] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–52, 1961.

[Fay79] M. Fay. First order unification in equational theories. In *Proceedings 4th Workshop on Automated Deduction, Austin (Tex., USA)*, pages 161–167, 1979.

[Fer92] M. Fernández. Narrowing based procedures for equational disunification. *Applicable Algebra in Engineering, Communication and Computation*, 3:1–26, 1992.

[Fer98] M. Fernández. Negation elimination in empty or permutative theories. *J. Symbolic Computation*, 26(1):97–133, July 1998.

[FH86] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(1):189–200, 1986.

[FR79] Jeanne Ferrante and Charles W. Rackoff. *The computational complexity of logical theories*. Number 718 in Lecture Notes in Mathematics. Springer Verlag, 1979.

[Fri85] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *IEEE Symposium on Logic Programming*, Boston (MA), 1985.

[GM86] J. A. Goguen and J. Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In Douglas De Groot and Gary Lindstrom, editors, *Functional and Logic Programming*, pages 295–363. Prentice Hall, Inc., 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984.

[GMW97] H. Ganzinger, C. Meyer, and C. Weidenbach. Soft typing for ordered resolution. In W. McCune, editor, *Proc. 14th Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1997.

[GRS87] Jean Gallier, S. Raatz, and Wayne Snyder. Theorem proving using rigid E-unification: Equational matings. In *Proc. 2nd IEEE Symp. Logic in Computer Science, Ithaca, NY*, June 1987.

[GS87] J. Gallier and W. Snyder. A general complete E-unification procedure. In P. Lescanne, editor, *Proceedings 2nd Conference on Rewriting Techniques and Applications, Bordeaux (France)*, volume 256 of *Lecture Notes in Computer Science*, pages 216–227, Bordeaux (France), 1987. Springer-Verlag.

[GS89] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2-3):203–260, October 1989.

[GTT93a] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In *Proc. 34th Symposium on Foundations of Computer Science*, pages 372–380, Palo Alto, CA, November 1993. IEEE Computer society press.

[GTT93b] Rémy Gilleron, Sophie Tison, and Marc Tommasi. Solving systems of set constraints using tree automata. In *Proc. 10th Symposium on Theoretical Aspects of Computer Science, Würzburg, LNCS*, 1993.

[GTT94] R. Gilleron, S. Tison, and M. Tommasi. Some new decidability results on positive and negative set constraints. In Jouannaud [Jou94], pages 336–351.

[HKK98] Claus Hintermeier, Claude Kirchner, and Hélène Kirchner. Dynamically-typed computations for order-sorted equational presentations. *Journal of Symbolic Computation*, 25(4):455–526, 98. Also report LORIA 98-R-157.

[Höl89] S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1989.

[HR91] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem proving strategies: The transfinite semantic tree method. *Journal of the ACM*, 38(3):559–587, July 1991.

[Hue76] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[Hul79] J.-M. Hullot. Associative-commutative pattern matching. In *Proceedings 9th International Joint Conference on Artificial Intelligence*, 1979.

[Hul80a] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, July 1980.

[Hul80b] J.-M. Hullot. *Compilation de Formes Canoniques dans les Théories équationelles*. Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France), 1980.

[JK91] J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[JKK83] J.-P. Jouannaud, Claude Kirchner, and Hélène Kirchner. Incremental construction of unification algorithms in equational theories. In *Proceedings International Colloquium on Automata, Languages and Programming, Barcelona (Spain)*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer-Verlag, 1983.

[JMW98] Florent Jacquemard, Christoph Meyer, and Christoph Weidenbach. Unification in extensions of shallow equational theories. In T. Nipkow, editor, *Porceedings of the 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 76–90, Tsukuba, Japan, 1998. Springer-Verlag.

[JO91] J.-P. Jouannaud and M. Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In J. Leach Albert, B. Monien, and M. Rodríguez Artalejo, editors, *Proceedings 18th ICALP Conference, Madrid (Spain)*, volume 510 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[Jou94] Jean-Pierre Jouannaud, editor. *First International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, München, Germany, September 1994. Springer Verlag.

[Kir85] Claude Kirchner. *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles*. Thèse de Doctorat d'Etat, Université Henri Poincaré – Nancy 1, 1985.

[KK89] Claude Kirchner and Hélène Kirchner. Constrained equational reasoning. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, Portland (Oregon)*, pages 382–389. ACM Press, July 1989. Report CRIN 89-R-220.

[KK90] Claude Kirchner and F. Klay. Syntactic theories and unification. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 270–277, June 1990.

[KK99] Claude Kirchner and Hélène Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKR90] Claude Kirchner, Hélène Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[KKV95] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[KL87] Claude Kirchner and P. Lescanne. Solving disequations. In D. Gries, editor, *Proceedings 2nd IEEE Symposium on Logic in Computer Science, Ithaca (N.Y., USA)*, pages 347–352. IEEE, 1987.

[Kla99] Nils Klarlund. A theory of restrictions for logics and automata. In *Proc. Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 406–417, 1999.

[Koz94] D. Kozen. Set constraints and logic programming. In Jouannaud [Jou94]. To appear in Information and Computation.

[KR98] Claude Kirchner and Christophe Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998.

[LM87] Jean-Louis Lassez and K. Marriot. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):301–318, 1987.

[LM94] D. Lugiez and J.L. Moysset. Tree automata help one to solve equational formulae in ac-theories. *Journal of symbolic computation*, 11(1), 1994.

[LMM88] Jean-Louis Lassez, M. J. Maher, and K. Marriot. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan-Kaufman, 1988.

[LR97] S. Limet and P. Réty. E-unification by means of tree tuple synchronized grammars:. *Discrete Mathematics and Theoretical Computer Science*, 1(1):69–98, 1997.

[Mah88] M. J. Maher. Complete axiomatization of the algebra of finite, rational trees and infinite trees. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*. COMPUTER SOCIETY PRESS, 1988.

[Mat70] Y. Matijasevič. Diophantine representation of recursively enumerable predicates. In *Actes du Congrès International des Mathématiciens*, volume 1, pages 235–238, Nice (France), 1970.

[MGS90] J. Meseguer, J. A. Goguen, and G. Smolka. Order-sorted unification. In Claude Kirchner, editor, *Unification*, pages 457–488. Academic Press, London, 1990.

[MN89] U. Martin and T. Nipkow. Boolean unification — the story so far. *Journal of Symbolic Computation*, 7(3 & 4):275–294, 1989. Special issue on unification. Part one.

[MN98] Martin Müller and Joachim Niehren. Ordering constraints over feature trees expressed in second-order monadic logic. In *Proc. Int. Conf. on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 196–210, Tsukuba, Japan, 1998.

[MNRA92] J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: the language BABEL. *Journal of Logic Programming*, 12(3):191–223, February 1992.

[MS90] Michael J. Maher and Peter J. Stuckey. On inductive inference of cyclic structures. In F. Hoffman, editor, *Annals of Mathematics and Artificial*

Intelligence, volume F. J.C. Baltzer Scientific Pub. Company, 1990. To appear.

[Mza86] J. Mzali. *Méthodes de filtrage équationnel et de preuve automatique de théorèmes*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, 1986.

[Nie93] R. Nieuwenhuis. Simple lpo constraint solving methods. *Information Processing Letters*, 47(2), 1993.

[Nie95] Robert Nieuwenhuis. On narrowing, refutation proofs and constraints. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95*, LNCS 914, pages 56–70, Kaiserslautern, Germany, April 5–7, 1995. Springer-Verlag.

[Nie99] Roberto Nieuwenhuis. Solved forms for path ordering constraints. In Paliath Narendran and Michael Rusinowitch, editors, *Porceedings of RTA'99*, volume 1631 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, July 1999.

[NO90] P. Narendran and F. Otto. Some results on equational unification. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449 of *Lecture Notes in Computer Science*, pages 276–291, July 1990.

[NR95] R. Nieuwenhuis and A. Rubio. Theorem Proving with Ordering and Equality Constrained Clauses. *J. Symbolic Computation*, 19(4):321–352, April 1995.

[NRS89] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7(3 & 4):295–318, 1989. Special issue on unification. Part one.

[NRV98] P. Narendran, M. Rusinowitch, and R. Verma. RPO constraint solving is in NP. In *Annual Conference of the European Association for Computer Science Logic*, Brno (Czech Republic), August 1998. Available as Technical Report 98-R-023, LORIA, Nancy (France).

[Nut89] W. Nutt. The unification hierarchy is undecidable. In H.-J. Bürckert and W. Nutt, editors, *Proceedings 3rd International Workshop on Unification, Lambrecht (Germany)*, June 1989.

[Pad96] Vincent Padovani. *Filtrage d'ordre supérieur*. PhD thesis, Université de Paris VII, 1996.

[Per90] Dominique Perrin. Finite automata. In *Handbook of Theoretical Computer Science*, volume Formal Models and Semantics, pages 1–58. Elsevier, 1990.

[Pic99] Reinhard Pichler. Solving equational problems efficiently. In Harald Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 97–111, Trento, Italy, July 7–10, 1999. Springer-Verlag.

[Plo72] G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[PP97] L. Pacholski and A. Podelski. Set constraints - a pearl in research on constraints. In Gert Smolka, editor, *Proc. 3rd Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–561, Linz, Austria, 1997. Springer Verlag. Invited Tutorial.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Rab69] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[Rab77] M. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.

[Rin97] Christophe Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997.

[SKR98] T. Shiple, J. Kukula, and R. Ranjan. A comparison of presburger engines for EFSM reachability. In *Proc. Computer Aided verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 280–292, 1998.

[Sny88] W. Snyder. *Complete sets of transformations for general unification.* PhD thesis, University of Pennsylvania, 1988.

[SS82] J. Siekmann and P. Szabó. Universal unification and classification of equational theories. In *Proceedings 6th International Conference on Automated Deduction, New York (N.Y., USA)*, volume 138 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[SS84] J. Siekmann and P. Szabó. Universal unification. In R. Shostak, editor, *Proceedings 7th International Conference on Automated Deduction, Napa Valley (Calif., USA)*, volume 170 of *Lecture Notes in Computer Science*, pages 1–42, Napa Valley (California, USA), 1984. Springer-Verlag.

[SS90] M. Schmidt-Schauß. Unification in permutative equational theories is undecidable. In Claude Kirchner, editor, *Unification*, pages 117–124. Academic Press inc., London, 1990.

[ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[Sza82] P. Szabó. *Unifikationstheorie erster Ordnung.* PhD thesis, Universität Karlsruhe, 1982.

[TA87] E. Tiden and S. Arnborg. Unification problems with one-sided distributivity. *Journal of Symbolic Computation*, 3(1 & 2):183–202, April 1987.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 134–191. Elsevier, 1990.

[Tho97] Wolfgang Thomas. Languages, automata and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–456. Springer-Verlag, 1997.

[Tom94] Marc Tommasi. Automates et contraintes ensemblistes. Thèse de l'Univ. de Lille, February 1994.

[Tre92] R. Treinen. A new method for undecidability proofs of first order theories. *J. Symbolic Computation*, 14(5):437–458, November 1992.

[Tre97] Ralf Treinen. Feature trees over arbitrary structures. In Patrick Blackburn and Maarten de Rijke, editors, *Specifying Syntactic Structures*, chapter 7, pages 185–211. CSLI Publications and FoLLI, 1997.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2:57–82, 1968.

[TW93] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *CAAP '93: 18th Colloquium on Trees in Algebra and Programming*, July 1993.

[Var96] M. Vardi. An automata-theoretic approach to linear time logic. In *Logic for concurrency: structure versus automata*, volume 1043 of *Lecture Notes in Comp. Science*. Springer Verlag, 1996.

[Vea97] Margus Veanes. *On simultaneous rigid E-unification*. PhD thesis, Computing Science Department, Uppsala University, Uppsala, Sweden, 1997.

[Ven87] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM*, 34(2):492–510, 1987.

[WBK94] A. Werner, A. Bockmayr, and S. Krischer. How to realize LSE narrowing. In *Proceedings Fourth International Conference on Algebraic and Logic Programming, Madrid (Spain)*, Lecture Notes in Computer Science. Springer-Verlag, September 1994.