Centrum für Informations-
und Sprachverarbeitung
Universität München

96-91

UNIF'96

Extended Abstracts of the Tenth
International Workshop on Unification

Klaus U. Schulz & Stephan Kepser (Eds.)

CIS—Universität München
Oettingenstr. 67 80538 München

Universität München
Centrum für Informations- und Sprachverarbeitung

UNIF'96

Extended Abstracts of the Tenth International Workshop on Unification

Klaus U. Schulz & Stephan Kepser

CIS-Bericht-96-91

Table of Contents

Towards Fuzzy Unification [*]

Francesca Arcelli, Ferrante Formato
DIIIE– Universitá di Salerno
84084 Fisciano (Salerno), Italy.
Fax:+39-89-964284, Ph.: +39-89-964254
e-mail: arcelli,formato@ponza.dia.unisa.it

1 Introduction

Various approaches to fuzzy Prolog has been proposed in the literature since Zadeh ([8])has proposed fuzzy set theory (see for example [4],[1] and [5]). Fuzziness has been introduced at several levels and from different points of views. A fuzzy degree associated to the facts in a program and to the rules has been introduced, notions of fuzzy inferences have been defined, based on different notions of logical consequence. Resolution techniques based on fuzzy logic have been used, employing fuzzy unification methods carried out in several ways. In particular this last aspect is not a completly accepted and syntactically and semantically defined concept.

For example Baldwin introduced inference methods for processing a knowledge base to answer queries based on semantic unification; this notion with probability theory and probabilistic fuzzy rules are discussed in [1], while Mukaidono in [6]defines a fuzzy matching between fuzzy predicates by using linguistic hedges.

In this work we describe our ongoing research towards the definition of a fuzzy unification. We start by giving a degree of unification based on "a soft" version of the usual definition of terms unifier. Then we show some important properties of such a degree, seen as a fuzzy subset and finally we propose a degree of unification associated to a complete set of transformation rules for a set of term equations.

2 A degree of unification

Let F be a signature, let X be a finite set of variables, and let $M(F,X)$ the free algebra generated by F and X. Let $\theta : X \rightarrow M(F,X)$ be a first-order substitution. Let $eq1 : M(F) \times M(F) \rightarrow L$ be a fuzzy relation over $M(F)$ with the following properties:

- i) $eq1(t,t) = 1$ for any closed term $t \in M(F)$

[*]This work has been partially supported by MURST 60%.

- ii) $eq1(t, t') = eq1(t', t)$ for any pair of closed terms $t, t' \in M(F)$
- iii) $eq1(t, t') \geq eq1(t, t'') \wedge eq1(t'', t')$ for any $t, t', t'' \in M(F)$

we call this relation a *fuzzy similitude relation.*

Alternatively, the similitude relation could be built by starting from a similitude relation eq_0 defined on the set of constants C and then extended to the set $M(F)$ by setting:

$$\begin{aligned} eq1(f(t_1, \ldots, t_n), f(t'_1, \ldots, t'_n)) &= \inf_{i=1..n} eq_0(t_i, t'_i) \\ eq1(a, b) &= eq_0(a, b)\ \forall a, b \in C \\ 0\ elsewhere. \end{aligned}$$

Given the set of first-order substitutions Θ, we can define a "degree of unification" between two terms as a fuzzy relation that "softens" the classical relation of unification. Recall that the classical unification relation between two terms t and t' could be stated as follows:

> a pair of terms (t, t') is unifiable if there exists a first-order substitution θ such that $\theta(t) = \theta(t')$

Along this lines, we can soften the meta-syntactical connectives, and define a "degree of unification" in the following manner:

$$U(t, t') = \sup_{\theta \in \Theta}\{eq1(\theta(t), \theta(t'))\} \text{ for any } t, t' \in M(F, X) \tag{1}$$

Formula (1) is the natural softening of the classical unification relation. The following properties follow:

Proposition 1 *For any variable $x \in X$, for any term $t \in M(F, X)$, $U(x, t) = 1$*

Proof: It suffices to consider a substitution θ such that $\theta(x) = t$. Then by property (i) of relation $eq1$ the thesis follows. Q.e.D

Proposition 2 *Let t be a term in $M(F, X)$. Then $U(t, t) = 1$*

Proof: If $t \in M(F)$ then $eq1(t, t) = 1$. If $t \in M(F, X)$, for any substitution θ, $eq1(\theta(t), \theta(t)) = 1$. Q.e.D

Proposition 3 *For any term $t, t' \in M(F, X)$ $U(t, t') = U(t', t)$*

proof: Since, for any $\theta \in \Theta$ it is $eq1(\theta(t), \theta(t')) = eq1(\theta(t'), \theta(t))$. The thesis follows immediately, passing to the supremum over Θ. Q.e.D

The degree of unification given in 1 is not transitive; indeed such property would entail that the following disequality:

$$U(t, t') \geq U(t, t'') \wedge U(t'', t')$$

holds for any $t, t', t'' \in M(F,X)$. If we consider two terms a and b in $M(F)$ and a variable $x \in X$, and take a similitude relation $eq1$ such that $eq1(a,b) < 1$, since by Proposition 2 it is $U(x,a) = U(x,b) = 1$, then $U(x,a) \wedge U(x,b) = 1 > U(a,b)$.

The definition of degree of unification given in (1) is quite natural. Many features of the classical unification relation could be found in this fuzzy counterpart; as an example, consider the aspects of unification connnected to computability theory. Assume that L and $M(F,X)$ are a computable lattice and a computable algebra, respectively. Then we can apply the concepts of L-computability theory (see [3]) and state whether or not the fuzzy relation U, seen as a fuzzy subset of $M(F,X) \times M(F,X)$, is decidable or less. We have the following remarkable result:

Theorem 1 *Consider the fuzzy unification degree U as an L-subset of $M(F,X) \times M(F,X)$. Then, if eq_0 is an L-computable subset of C, U is L-computable.*

proof: To show this it is sufficient to prove that U is a computable function from $M(F,X) \times M(F,X)$ to L, provided that eq_0 is a computable function from $C \times C$ to L. It is immediate that eq_1 is a computable function from $M(F) \times M(F)$ to L, since it is defined from eq_0 using structural induction. We now prove that U is a computable function using the structural induction over the terms of $M(F,X)$. We say that a $eq(t,t')$ is computable if it can be calculated in a finite number of steps.

- (i) Suppose t and t' are constants: let $t = a$ and $t' = b$. Then $U(a,b) = eq1(a,b)$ which is a computable expression.

- (ii) We have to prove that $U(f(t'_1, \ldots, t'_n), g(t_1, \ldots, t_n))$ is a computable expression, provided that $U(t_i, t'_i)$ is computable.

If $f \neq g$ or $n = m$ then, by definition of U, $U(f(t'_1, \ldots, t'_n), g(t_1, \ldots, t_n)) = 0$ Otherwise, we have that

$$U(f(t'_1, \ldots, t'_n), f(t_1, \ldots, t'_n)) = \sup_{\theta \in \Theta} \left\{ eq1(\theta(f(t'_1, \ldots, t'_n)), \theta(g(t_1, \ldots, t_n)) \right\}$$

By construction of θ, it is

$$U(f(t'_1, \ldots, t'_n), f(t_1, \ldots, t_n)) = \sup_{\theta \in \Theta} \left\{ eq1(f(\theta(t'_1), \ldots, \theta(t'_n)), g(\theta(t_1), \ldots, \theta(t_n)) \right\}$$

i.e. , by definition of eq_1:

$$U(f(t'_1, \ldots, t'_n), f(t_1, \ldots, t_n)) = \sup_{\theta \in \Theta} \left\{ \bigwedge_{i=1}^{n} eq_1(\theta(t_i), \theta(t'_i)) \right\}$$

finally, by distributivity

$$U(f(t'_1, \ldots, t'_n), f(t_1, \ldots, t_n)) = \bigwedge_{i=1}^{n} \left\{ \sup_{\theta \in \Theta} eq_1(\theta(t_i), \theta(t'_i)) \right\}$$

By induction hypotheses

$$\sup_{\theta\in\Theta} eq_1(\theta(t_i),\theta(t_i'))$$

is a computable expression, so, $U(f(t_1',\dots,t_n'),g(t_1,\dots,t_n'))$ is a computable expression since it is the infimum over a finite number of computable expressions.

Since we proved that U is a computable function from $M(F,X)\times M(F,X)$ it follows that U is an L-subset which is L-computable.

Q.e.D

We now extend the definition of unification degree to a finite set of first-order terms. Just like the first-order case, given a set $X=\{t_1,\dots,t_n\}$ of terms in $M(F,X)$, we call *degree of unification* of X the following fuzzy L-subset $\mathcal{U}$ of $P(M(F,X))$, where $P(M(F,X))$ is the powerset of $M(F,X)$

$$:\mathcal{U}(X)=\inf_{t,t'\in X}(U(t,t')) \tag{2}$$

where U is the degree of unification of a pair of terms. A simplified definition of degree of unification for a set of first-order terms is given by the following formula. Let $D(X,2)$ be the set of binary dispositions of elements of X without redundancies. Then we set $\mathcal{U}_1(X)$ as follows:

$$\mathcal{U}_1(X)=\inf_{t,t'\in D(X,2)}(U(t,t')) \tag{3}$$

The following propositions hold:

Proposition 4 *Let X be a set of first-order terms. Then $\mathcal{U}(X)=\mathcal{U}_1(X)$*

Proof: Since $D(X,2)\subseteq M(F,X)^2$ it is $\mathcal{U}_1(X)\le\mathcal{U}(X)$. Besides, since $U(t,t')=U(t',t)$ for any $t,t'\in M(F,X)$, then $\mathcal{U}(X)\le\mathcal{U}_1(X)$

Q.e.D

Proposition 5 *U is an L-computable subset of $P(M(F,X))$.*

Proof: Immediate from the finiteness of X.

3 Fuzzy Equational Unification

Given a free algebra of terms $M(F,X)$, where F is a finite signature and X is a finite set of variables, we can set a fuzzy derivational system based on the classic derivation rules for a set of equations (see [7]). A fuzzy derivation system is a finite set of ordered pairs $\langle\mathcal{R},\Lambda\rangle=\{\langle r_i,\lambda_i\langle\}$ where r_i is a rule of the kind $X\Rightarrow X'$, where X and X' are sets of term equations and λ_i is an element of the lattice L expressing the degree of derivability of X' from X. We just give some sketchy ideas on how to define the pair $\langle\mathcal{R},\Lambda\rangle$. Given a complete set of transformation rules as described in [7], we assign a derivability degree equal to 1 to all transformation rules, except for the following one:

$$X\cup\{u\doteq u'\}\overset{\lambda}{\Rightarrow}X$$

where $u \in M(F, X)$ and λ is $eq(u, u')$ if $u, u' \in M(F)$ and is 0 otherwise. We write $D(X \vdash Y) = \lambda$ if X and Y are sets of term equations and $X \overset{\lambda}{\Rightarrow} Y$ for some fuzzy rule $\langle r, \lambda$. Let X and X' be a set of first-order term equations. We call *derivation of X' from X* the sequence $X_1, \ldots, X_n$ such that $X_1 = X$ and $X_n = Y$ and, for any $i = 2..n$ there exists a rule R_k in $\mathcal{R}$ such that $X_i \Rightarrow X_{i+1}$. We say that a set of first-order term equations is *solved* if it is in "diagonal form", i.e. its equations are in the form $x \doteq t$ where $x \in X$ and $t \in M(F, X)$, x does not occurr elsewhere into the equations of X and x does not occurr in t. Finally, a set of first-order equations X is called *solvable* if there exists a solved set of term equations $\bar{X}$ that is derivable from X. We can set a degree of unification for a set of term equations in the following way:

$$D(X) = \sup_{X_1,\ldots,X_n} \left\{ \inf_{i=1..n-1} (D(X_i, X_{i+1}))X_1 \vdash \ldots \vdash X_n \text{ where } X_n \text{ is solved} \right\}$$

In this way we have obtained a degree of unification associated to a complete set of transformation rules. Such degree could be used in a computable model of fuzzy unification for a low-level implementation of a fuzzy resolution-based logic programming language.

References

[1] J.F.Baldwin, T.P.Martin and B.W. Pilsworth. "The implementation of FPROLOG - A Fuzzy Prolog Interpreter", *Fuzzy Sets and Systems*, 23:119-129, 1987.

[2] M.K.Chakraborty, "Graded Consequence: Further Studies", *Journal Studies of Applied non-Classical Logics*, 21, 1995.

[3] L.Biacino and G.Gerla , "Decidability and recursive enumerability for fuzzy subsets", in: B. Bouchon, L. Saitta, R.R. Yager (eds), *Uncertainty and Intelligent Systems*, LNCS Berlin, 55-62, 1988.

[4] R.C.T.Lee, " Fuzzy Logic and the resolution principle", *Journal of the ACM*, 19:109-119, 1972.

[5] M.Mukaidono, Z.Shen and L.Ding, " Fundamentals of Fuzzy Prolog", *Int.Jornal of Approximated Reasoning*, 3,2, 1989.

[6] H.Yasui, Y.Hamada and M.Mukaidono, "Fuzzy Prolog based on Lukasievitcz implication and bounded product, *Proc.IEEE Intern.Conf. on Fuzzy Sets*,2:949-954, 1995.

[7] W.Snyder, *A Proof Theory for General Unification*, Birkhauser ed.,1995.

[8] L.Zadeh. "Fuzzy Sets", *Information and Control*, 8, 1965.

Unification and Matching modulo Nilpotence

Qing Guo, Paliath Narendran[*] and D.A. Wolfram [†]

Abstract

We consider *nilpotence*: the simple theory $f(x,x) = 0$ where 0 is a constant. We show that elementary unification and matching modulo this theory are *NP*-complete. We also consider the case where the function f additionally satisfies associativity and commutativity. We found that the problems are still *NP*-complete. However, when 0 is also the unity of f, i.e., $f(x,0) = x$, unification and matching problems can be solved in polynomial time. Furthermore, we show that unification and matching problems remain in polynomial time when a homomorphism is added to the theory. This polynomial time algorithm can be used to solve a subclass of set constraints. Second-order matching modulo nilpotence is shown to be undecidable.

[*]Institute of Programming and Logics, Department of Computer Science, State University of New York at Albany, Albany, NY 12222, U.S.A.

[†]Department of Computer Science The Australian National University Canberra, ACT 0200 Australia

Uniform Representation of Recursively Enumerable Sets with Simultaneous Rigid E-Unification (Extended Abstract)

Margus Veanes

Computing Science Department, Uppsala University
Box 311, S-751 05 Uppsala, Sweden
email: margus@csd.uu.se

1 Introduction

Recently it was proved that the problem of simultaneous rigid E-unification (SREU) is undecidable [8]. Here we perform an in-depth investigation of this matter and obtain that one can use SREU to uniformly represent any recursively enumerable set. From this representation follows that SREU is undecidable already for 6 rigid equations with ground left hand sides and 2 variables.

There is a close correspondence between solvability of SREU problems and provability of the corresponding formulas in intuitionistic first order logic with equality. Due to this correspondence we obtain representation of the recursively enumerable sets in intuitionistic first order logic with equality with one binary functionsymbol and a countable set of constants. From this result follows the undecidability of the $\exists\exists$-fragment of intuitionistic logic with equality. This is an improvement of the recent result regarding the undecidability of the $\exists^*$-fragment in general [10].

1.1 Background of SREU

Simultaneous rigid E-unification was proposed by Gallier, Raatz and Snyder [15] as a method for automated theorem proving in classical logics with equality. It can be used in automatic proof methods, like semantic tableaux [12], the connection method [3] or the mating method [1],model elimination [22], and others that are based on the Herbrand theorem, and use the property that a formula is valid (i.e., its negation is unsatisfiable) iff all paths through its matrix are inconsistent. This property was first recognized by Prawitz [27] (for first order logic without equality) and later by Kanger [19] (for first order logic with equality).In first order logic with equality, the problem of checking the inconsistency of the paths results in SREU. Before SREU was proved to be undecidable, there were several faulty proofs of its decidability, e.g. [13, 17].

1.2 Outline of this Paper

This paper constitutes a summary of the main results presented in Veanes [30], where one can find detailed proofs to all the statements that are made here. In Section 2 we explain the main notations.

In Section 3 we introduce words (as representations of strings) and sentences (as representations of sequences of strings), as the basic components needed to formally represent Turing machine computations. The main result of this section is called the Sentence Lemma.

In Section 4 we describe a technique that can be used toexpress (roughly) that one sequence of strings is an encoding of pairwise adjacent strings of another sequence. This technique is formally stated as the Shifted Pairing Lemma.

The Sentence Lemma and the Shifted Pairing Lemma are then used in Section 5 to show that one can use SREU to represent(uniformly) any r.e. set, here we identify r.e. sets with languages accepted by Turing machines. We call this result the SREU Normal Form Theorem for RE. The undecidability of SREU follows immediately from this theorem (2 variables and six rigid equations is enough).

From the SREU Normal Form Theorem for RE we obtain a uniform characterization of all r.e. sets with certain simple formulas in intuitionistic first order logic with equality, using just one function symbol (of arity 2) and some number of constants. In particular, the undecidability of the $\exists\exists$-fragment of intuitionistic logic follows from this result.

We conclude the paper in Section 7 by summarizing the current status and stating some open problems regarding decidability of fragments of SREU.

2 Preliminaries

Throughout the paper, the first order language that we are working with is designated by L. L has one binary function symbol $\bullet$ and a countable set of constants L_0. We will use infix notation for $\bullet$ and assume that it associates to the right, so $t_1 \bullet t_2 \bullet t_3$ stands for the term $\bullet(t_1, \bullet(t_2, t_3))$. In general we will use the letters t and s to stand for terms in L. We write $X \subset Y$ to say that X is a *nonempty finite* subset of Y. A

- ▷ *rigid equation* is an expression of the form $E \vDash_\forall s = t$ where E is a finite set of equations and s and t are arbitrary terms. A *system* of rigid equations is a finite set of rigid equations.

A substitution θ (θ is assumed to map variables to *ground* terms) is a

- ▷ *solution of* or *solves* a rigid equation $E \vDash_\forall s = t$ if

$$\vdash \quad (\bigwedge_{e \in E} e\theta) \Rightarrow s\theta = t\theta,$$

  θ solves a system of rigid equations if it solves each member of the system.

Here $\vdash$ is either classical or intuitionistic provability (for this class of formulas they coincide). The problem of solvability of systems of rigid equations is called *simultaneous rigid E-unification* or SREU for short. Solvability of a single rigid equation is called *rigid E-unification.*

3 Words and Sentences

Words are certain terms of L that represent strings, and sentencesare certain terms that represent sequences of strings. We will use the letters v and w to stand for strings of constants. Formally, we say that a gound term t of L is a

- ▷ *q-word* or simply a *word* when it has the form $a_1 \bullet a_2 \bullet \cdots \bullet a_n \bullet q$ for some $n \geq 0$ where all the a_i and q are constants. If $n = 0$ then t is said to be *empty.*

If t is a word $a_1 \bullet a_2 \bullet \cdots \bullet a_n \bullet q$ we mostly use the shorthand $v \bullet q$ for t where v is the string $a_1 a_2 \cdots a_n$. We also say that t

▷ *represents* the string v, in symbols $\widehat{t} = v$.

Note that any constant q is an empty q-word and represents ϵ. We want to be more specific and talk about strings in certain regular sets. Let R be a regular set over some set of constants in L. We say a ground term t of L is a

▷ *word in* R if t is a word and it represents a string in R.

Sentences are just representations of sequences of strings. Let us first choose a fixed constant $[]$ ("nil") of L. Formally, a ground term t of L is called a

▷ *sentence* if it has the form $t_1 \bullet t_2 \bullet \cdots \bullet t_n \bullet []$ for some $n \geq 0$ where each t_i is a word. If $n = 0$ then t is said to be *empty*.

We use $[t_1, t_2, \ldots, t_n]$ as a shorthand for the corresponding sentence. We say that a sentence $t = [t_1, t_2, \ldots, t_n]$

▷ *represents* the sequence of strings $\widehat{t} = (\widehat{t_1}, \widehat{t_2}, \ldots, \widehat{t_n})$.

Our aim is to represent sequences of strings, where each string belongs to some member of a given family of regular sets, such that the sequence has some given regular pattern. For that purpose we introduce the following notion.

Let $\Sigma, \Gamma \subset L_0$ and let $\{R_q\}_{q \in \Gamma}$ be a family of regular sets over Σ and let R be a regular set over Γ. We say that a sentence $t = [t_1, t_2, \ldots, t_n]$ is a

▷ *sentence in* $\{R_q\}^R$ if each t_i is a q_i-word in R_{q_i} for some $q_i \in \Gamma$, $q_1 q_2 \cdots q_n \in R$.

In other words, t is a sentence in $\{R_q\}^R$ iff any q-word of t is a word in R_q, and if we replace all the words of t with the corresponding empty words then the resulting term is a $[]$-word in R. When all the members of the family are the same regular set then we drop the index in our notation.

For example, t is a sentence in $\{R_a, R_b, R_c\}^{ab^*c}$ means that the first word of t is an a-word in R_a, the last word of t is a c-word in R_c and the middle ones (if any) are b-words in R_b.

Theorem 3.1 *Let $\Sigma, \Gamma \subset L_0 \setminus \{[]\}$ be disjoint. Let $\{R_c\}_{c \in \Gamma}$ be a family of reqular sets over Σ. Let R be a regular set over Γ. There exists a system $S(x)$ ofrigid equations such that θ solves $S(x)$ iff $x\theta$ is a sentence in $\{R_c\}^R$.*

We will refer to this theorem as the Sentence Lemma. More precisely,the system $S(x)$ consists of two rigid equations, both of which haveground left hand sides and one variable x. Furthermore, the system is obtained effectively from the regular sets.

4 Shifted Pairing

The purpose of this section is to describe a technique, called shifted pairing, that can be used to construct a system of rigid equations, the solutions of which aresentences with certain interesting properties. This technique was first used by Plaisted [26].

4.1 Encoding Pairs of Strings

Given a set of constants $\Sigma \subset L_0$, we want to encode pairs of strings over Σ in a simple manner.Let $\flat$ be a fixed constant in L_0 called a *blank*. We can assume without loss of generality that $\flat \in \Sigma$, $\Sigma \setminus \{\flat\}$ is nonempty and that we only wish to encode pairs of strings in Σ^* that don't end with a blank (otherwise just expand Σ with $\flat$). We say that a function $\langle\rangle : \Sigma \times \Sigma \to L_0$ is a

▷ *pairing function* for Σ if $\langle\rangle$ is injective and $\langle\Sigma\rangle = \{\,\langle a,b\rangle \mid a,b\in\Sigma\,\}$ is disjoint from Σ, and we associate the following sets of equations with $\langle\rangle$:

$$\begin{array}{rcl}\Pi_1^{\langle\rangle} & = & \{\,\langle a,b\rangle = a \mid a,b\in\Sigma\,\},\\ \Pi_2^{\langle\rangle} & = & \{\,\langle a,b\rangle = b \mid a,b\in\Sigma\,\}.\end{array}$$

We will abbreviate $\Pi_1^{\langle\rangle}$ and $\Pi_2^{\langle\rangle}$ by Π_1 and Π_2, respectively.

Let $\langle\rangle$ be a pairing function for Σ. Consider

$$v = \langle a_1,b_1\rangle\langle a_2,b_2\rangle\cdots\langle a_k,b_k\rangle \in \langle\Sigma\rangle^*$$

for some $k\geq 0$ and let $n,m\geq 0$ be least such that $a_{n+1},\ldots,a_k$ and $b_{m+1},\ldots,b_k$ are blanks. We say that

▷ v *encodes* the pair $(a_1a_2\cdots a_n, b_1b_2\cdots b_m)$ of strings.

We will write $\langle v,w\rangle$ for any string that encodes the pair (v,w) of strings in $\Sigma^*\setminus\Sigma^*\flat$.

4.2 Shifted Pairing

We want to encode adjacent pairs of strings in a given sequence of strings. Let $\Sigma\subset L_0$. Assume $\flat\in\Sigma$, $\Sigma\setminus\{\flat\}$ is nonempty and let $\langle\rangle$ be a pairing function for Σ. Let $\vec{w}=(w_1,w_2,\ldots,w_n)$ be a *nonempty* sequence of strings in $\Sigma^*\setminus\Sigma^*\flat$. We say that a sequence $\vec{v}=(v_1,v_2,\ldots,v_n)$ of strings in $\langle\Sigma\rangle^*$ is a

▷ *shifted pairing* of $\vec{w}$ if v_i encodes the pair (w_i,w_{i+1}) for $1\leq i<n$ and v_n encodes the pair (w_n,ϵ), i.e., $\vec{v}=(\langle w_1,w_2\rangle,\langle w_2,w_3\rangle,\ldots,\langle w_{n-1},w_n\rangle,\langle w_n,\epsilon\rangle)$.

We refer to the following theorem as the Shifted Pairing Lemma. It constitutes the kernel of the proof the SREU Normal Form Theorem.

Theorem 4.1 *Let $\Sigma\subset L_0$ be such that $\flat\in\Sigma$, $\Sigma\setminus\{\flat\}$ is nonempty, and let $\langle\rangle$ be a pairing function for Σ. Let also $\Gamma\subset L_0$. Assume that Σ, $\langle\Sigma\rangle$, Γ and $\{[]\}$ are all pairwise disjoint.*

Let $q\in\Sigma\setminus\{\flat\}$. There is a system $\mathrm{SP}_q(z,x,y)$ of rigid equations such that

- *θ solves $\mathrm{SP}_q(z,x,y)$ iff*
- *$\widehat{y\theta}$ is a shifted pairing of $\widehat{x\theta}$ and the first string of $\widehat{x\theta}$ is $\widehat{qz\theta}$,*

for any substitution θ such that $z\theta$ is a c-word in $(\Sigma\setminus\{\flat\})^*$ for some $c\in\Gamma$, $x\theta$ is a sentence in $\{\Sigma^+\setminus\Sigma^*\flat\}^{\Gamma^+}$ and $y\theta$ is a sentence in $\{\langle\Sigma\rangle^+\}^{\Gamma^+}$.

Let ε be fixed element of Γ. The system $\mathrm{SP}_q(z,x,y)$ is defined as follows:

$$\mathrm{SP}_q(z,x,y) = \left\{\begin{array}{lcl}\Pi_1\cup\{\,\varepsilon=c\mid c\in\Gamma\,\}\cup\{\flat\bullet\varepsilon=\varepsilon\} & \vdash_{\forall} & x=y,\\ \Pi_2\cup\{\,\varepsilon=c\mid c\in\Gamma\,\}\cup\{\flat\bullet\varepsilon=\varepsilon,\ \varepsilon\bullet[]=[]\} & \vdash_{\forall} & x=(q\bullet z)\bullet y\end{array}\right.$$

Assuming that $x\theta$ represents the sequence $\vec{w}$ as above thenthe first rigid equation has the effect that $y\theta$ must have the form

$$(\langle w_1,\_\rangle,\langle w_2,\_\rangle,\ldots,\langle w_{n-1},\_\rangle,\langle w_n,\_\rangle),$$

and from the second rigid equation follows that $y\theta$ must have the form

$$(\langle\_,w_2\rangle,\langle\_,w_3\rangle,\ldots,\langle\_,w_n\rangle,\langle\_,\epsilon\rangle).$$

Together they give the desired effect.

5 Uniform Characterization of RE with SREU

Here we show that any r.e. set can be represented by a system of simultaneous rigid equations which is obtained uniformly in the r.e. index of that set. In particular, this theorem and the way the system in the theorem is constructed imply that given any r.e. set W over some alphabet Σ and a string w over Σ, one can effectively construct a system of rigid equations, having ground left hand sides and only two variables, which has a solution iff $w \in W$. So SREU is undecidable already with ground left hand sides (which was also shown by Plaisted [26]) and only two variables (that is a new result).

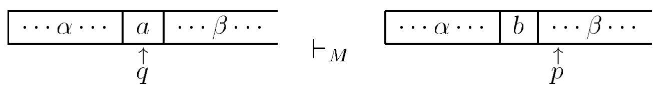
5.1 The Turing Machine Model

We follow Hopcroft and Ullmann [18]. Formally, a *Turing machine* M is a 7-tuple $(Q, \Sigma_0, \Sigma_1, \delta, q_0, \flat, F)$, where Q is the set of all *states* of M, Σ_0 is the *input alphabet* not including $\flat$, $\Sigma_1 = \Sigma_0 \cup \{\flat\}$, $\delta : Q \times \Sigma_1 \to Q \times \Sigma_1 \times \{\mathrm{L}, \mathrm{R}\}$ is the *transition function*, $q_0 \in Q$ is the *initial* state, and $F \subseteq Q$ is the set of *final* states. We also assume that Q and Σ_1 are disjoint subsets of L_0. An

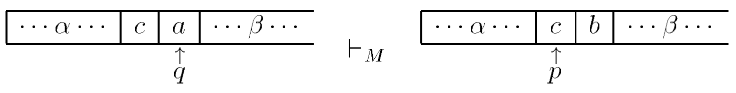
- ▷ *instantaneous description* (ID) of M is any string $\alpha q \beta$ where $q \in Q$ and $\alpha \in \Sigma_1^*$ and β is a string in Σ_1^* not ending with a blank.

Let $\vec{\flat}$ stand for a string of 0 or more blanks. A

- ▷ *move* is a pair (v, w) of ID's such that if $v\vec{\flat} = \alpha q a \beta$ and $\delta(q, a) = (p, b, \mathrm{R})$ then $w\vec{\flat} = \alpha b p \beta$,



  and if $v\vec{\flat} = \alpha c q a \beta$ and $\delta(q, a) = (p, b, \mathrm{L})$ then $w\vec{\flat} = \alpha p c b \beta$,



  i.e., w is obtained from v according to the next move function.

The binary relation of all moves of M is denoted by $\vdash_M$, as shown in the figures above, and its transitive and reflexive closure by $\vdash_M^*$. The

- ▷ *language accepted by* M, $L(M)$, is the following set

$$L(M) = \{\, w \in \Sigma_0^* \mid q_0 w \vdash_M^* \alpha p \beta \text{ where } p \in F \text{ and } \alpha p \beta \text{ is an ID} \,\}.$$

- ▷ A *valid computation* of M is a nonempty sequence $(w_1, w_2, \ldots, w_n)$ such that
  - each w_i is an ID of M, i.e., $w_i \in \Sigma_1^* Q(\Sigma_1^* \setminus \Sigma_1^* \flat)$ for $1 \le i \le n$,
  - w_1 is the *initial* ID, one of the form $q_0 v$ where $v \in \Sigma_0^*$,
  - w_n is a *final* ID, $w_n \in \Sigma_1^* F(\Sigma_1^* \setminus \Sigma_1^* \flat)$,
  - $w_i \vdash_M w_{i+1}$ for $1 \le i < n$, i.e., each pair (w_i, w_{i+1}) is a move of M.

We will use the following relationship between valid computations and the language of M without further notice: there is a valid computation of M with initial ID $q_0 v$ iff $v \in L(M)$.

5.2 The SREU Normal Form Theorem for RE

Theorem 5.1 *Let $M = (Q, \Sigma_0, \Sigma_1, \delta, q_0, \mathfrak{b}, F)$ be a TM and let ε be constant not in Q or Σ_1. There is a system $S_M(z, x, y)$ of rigid equations such that for any substitution θ that solves $S_M(z, x, y)$, $z\theta$ is an ε-word and $L(M) = \{\, \widehat{z\theta} \mid \theta$ solves $S_M(z, x, y)\,\}$.*

The auxiliary variables x and y are such that, for any θ that solves $S_M(z, x, y)$, $\widehat{x\theta}$ is a valid computation of M with initial ID $q_0\widehat{z\theta}$ and $\widehat{y\theta}$ is a shifted pairing of $\widehat{x\theta}$.

The main steps in the proof of the theorem are as follows. First the regular sets R_{id}, R_{fin} and R_{mv} are defined so that R_{id}, R_{fin} and R_{mv} are all the IDs, the final IDs and the encodings of moves of M, respectively. Based on these, the Sentence Lemma gives us the systems $S_{\mathrm{id}}(x)$ and $S_{\mathrm{mv}}(y)$ and $S_{\mathrm{in}}(z)$, such that

1. θ solves $S_{\mathrm{id}}(x)$ iff $x\theta$ is a sentence in $\{\varepsilon \mapsto R_{\mathrm{id}}, \varepsilon_1 \mapsto R_{\mathrm{fin}}\}^{\varepsilon^*\varepsilon_1}$,
2. θ solves $S_{\mathrm{mv}}(y)$ iff $y\theta$ is a sentence in $\{\varepsilon \mapsto R_{\mathrm{mv}}, \varepsilon_1 \mapsto \langle\Sigma\rangle^+\}^{\varepsilon^*\varepsilon_1}$, and
3. θ solves $S_{\mathrm{in}}(z)$ iff $z\theta$ is an ε-word in Σ_0^*.

The Shifted Pairing Lemma gives us the system $\mathrm{SP}_{q_0}(z, x, y)$. Finally S_M is given by the union of all those systems,

$$S_M(z, x, y) = S_{\mathrm{id}}(x) \cup S_{\mathrm{mv}}(y) \cup S_{\mathrm{in}}(z) \cup \mathrm{SP}_{q_0}(z, x, y).$$

Given a word t and a TM M, the construction of $S_M(t, x, y)$ is effective. Also, all the left hand sides are ground. We get the following corollary.

Corollary 5.2 *SREU is undecidable even when restricted to ground equations on the left hand side and allowing only two variables, in any first order language with at least one binary function symbol and one constant.*

The first proof of the udecidability of SREU [8] was by reduction of the monadic semi-unification [2] to SREU.This proof was followed by two alternative (more transparent) proofsby the same authors, first by reducing second order unification to SREU [7, 10], and then by reducing Hilberts 10'th to SREU [9]. The undecidability of second order unification was proved by Goldfarb [16]. Reduction of second order unification to SREU is very simple, showing how close these problem are to each other.

Plaisted took the Post's Correspondence Problem and reduced it to SREU [26]. From his proof follows that SREU is undecidable already with ground left hand sides and three variables. He uses several function symbols of arity 1 and 2. The basic technique used by Plaisted is the same as the one used here.

A technique similar to the one used in the reduction of Hilberts 10'th to SREU, is used by Voda and Komara [31] to argue for (we did not check the details) the undecidability of the problem of Herbrand skeletons, i.e., given n and a formula $\psi = \exists\vec{x}\varphi(\vec{x})$ where φ is quantifier free, if the Herbrand skeleton of size n of ψ is solvable. (The Herbrand skeleton of size n of ψ is the disjunction of n variants of φ.) For $n = 1$ this problem comprises also SREU.

6 Uniform Representation of RE in Intuitionistic Logic with Equality

From the SREU Normal Form Theorem for RE we get the corresponding result for intuitionistic logic with equality.

Theorem 6.1 *Let M be a Turing machine. There is a formula $\varphi_M(z, x, y)$ and a constant ε in L such that, for all ground terms t in L: t is an ε-word in $L(M)$ iff $\vdash_{\mathrm{i}} \exists x \exists y \varphi(t, x, y)$.*

Since the construction of the formula φ_M is effective, we obtain that the $\exists\exists$-fragment of intuitionistic logic isundecidable. This is an improvement of the undecidability result of the $\exists^*$-fragment in general shown recently by Degtyarev and Voronkov [9, Theorem 10] (or [10, Theorem 3]).

A closely related problem is the skeleton instantiation problem, i.e., the problem of existence of a derivation with a given skeleton. Voronkov shows that SREU is polynomially reducible to this problem [32, Theorem 3.12]. Moreover, the basic structure of the skeleton is determined by the number of variables in the SREU problem and the number of rigid equations in it. Our result implies that this problem is undecidable already for a very restricted class of skeletons.

Decidabilty problems for some other fragments of intuitionistic logicwith and without equality were studied by Orevkov [24, 25], Mints [23] and Lifschitz [21]. More recently some new results have been obtained by Degtyarev and Voronkov [33, 32, 11, 6], and Tammet [29].

7 Current Status and Open Problems

Decidability of rigid E-unification has been known for some time now, for a clear proof see De Kogel [4]. The current status about what is known about SREU and rigid E-unification is summarized below.

1. Rigid E-unification with ground lefthand side is NP-complete [20]. Rigid E-unification in general is NP-complete and there exist finite complete sets of unfiers [14, 13].

2. If all function symbols have arity ≤ 1 then SREU is PSPACE-hard [17]. If only one unary function symbol is allowed then the problem is decidable [6, 5]. If only constants are allowed then the problem is NP-complete [6] if there are at least two constants. If there are more than one unary function symbol then the decidabilityis still an open question.

3. In general SREU is undecidable [8], already with ground left hand sides [26] and two variables [30].

Some other decidable cases of SREU are also described by Plaisted [26]. It should also be noted that the decidability of SREU with just one variable is an open question and thus also the decidability of the $\exists$-fragment of intuitionistic logic with equality. Note that SREU is decidable when there are no variables, then each rigid equation can be decided for example by using the Shostak congruence closure algorithm [28, 4].

References

[1] P.B. Andrews. Theorem proving via general matings. *Journal of the Association for Computing Machinery*, 28(2):193–214, 1981.

[2] M. Baaz. Note on the existence of most general semi-unifiers. In *Arithmetic, Proof Theory and Computation Complexity*, volume 23 of *Oxford Logic Guides*, pages 20–29. Oxford University Press, 1993.

[3] W. Bibel. *Deduction. Automated Logic.* Academic Press, 1993.

[4] E. De Kogel. Rigid E-unification simplified. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, number 918 in Lecture Notes in Artificial Intelligence, pages 17–30, Schloß Rheinfels, St. Goar, Germany, May 1995.

[5] A. Degtyarev, Yu. Matiyasevich, and A. Voronkov. Simultaneous rigid E-unification is not so simple. UPMAIL Technical Report 104, Uppsala University, Computing Science Department, April 1995.

[6] A. Degtyarev, Yu. Matiyasevich, and A. Voronkov. Simultaneous rigid E-unification and related algorithmic problems. In *LICS'96*, pages 1–11, 1996.

[7] A. Degtyarev and A. Voronkov. Reduction of second-order unification to simultaneous rigid E-unification. UPMAIL Technical Report 109, Uppsala University, Computing Science Department, June 1995.

[8] A. Degtyarev and A. Voronkov. Simultaneous rigid E-unification is undecidable. UPMAIL Technical Report 105, Uppsala University, Computing Science Department, May 1995.

[9] A. Degtyarev and A. Voronkov. Simultaneous rigid E-unification is undecidable. In *Computer Science Logic Workshop*, pages 1–12, 1995.

[10] A. Degtyarev and A. Voronkov. The undecidability of simultaneous rigid e-unification (note). *Theoretical Computer Science*, pages 1–10, 1995.

[11] A. Degtyarev and A. Voronkov. Skolemization and decidability problems for fragments of intuitionistic logic. In *LICS'96*, pages 1–10, 1996.

[12] M. Fitting. First-order modal tableaux. *Journal of Automated Reasoning*, 4:191–213, 1988.

[13] J. Gallier, P. Narendran, D. Plaisted, and W. Snyder. Rigid E-unification: NP-completeness and applications to equational matings. *Information and Computation*, 87(1/2):129–195, 1990.

[14] J.H. Gallier, P. Narendran, D. Plaisted, and W. Snyder. Rigid E-unification is NP-complete. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 338–346. IEEE Computer Society Press, July 1988.

[15] J.H. Gallier, S. Raatz, and W. Snyder. Theorem proving using rigid E-unification: Equational matings. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 338–346. IEEE Computer Society Press, 1987.

[16] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[17] J. Goubault. Rigid $\vec{E}$-unifiability is DEXPTIME-complete. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 1994.

[18] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Co., 1979.

[19] S. Kanger. A simplified proof method for elementary logic. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning. Classical Papers on Computational Logic*, volume 1, pages 364–371. Springer Verlag, 1983. Originally appeared in 1963.

[20] D. Kozen. Positive first-order logic is NP-complete. *IBM J. of Research and Development*, 25(4):327–332, 1981.

[21] V. Lifschitz. The decidability problem for some constructive theories of equality (in Russian). *Zapiski Nauchnyh Seminarov LOMI*, 4:78–85, 1967. English translation in: Seminars in Mathematics: Steklov Math.Inst.4, Consultants Bureau, NY-London, 1969, pp.29–31.

[22] D.W. Loveland. Mechanical theorem proving by model elimination. *Journal of the Association for Computing Machinery*, 15:236–251, 1968.

[23] G.E. Mints. Collecting terms in the quantifier rules of the constructive predicate calculus (in Russian). *Zapiski Nauchnyh Seminarov LOMI*, 4:78–85, 1967. English Translation in: Seminars in Mathematics: Steklov Math.Inst.4, Consultants Bureau, NY-London, 1969, pp.43–46.

[24] V.P. Orevkov. The undecidability in the constructive predicate calculus of the class of formulas of the form $\neg\neg\forall\exists$ (in Russian). *Soviet Mathematical Doklady*, 163(3):581–583, 1965. The journal is cover-to-cover translated to English.

[25] V.P. Orevkov. Solvable classes of pseudo-prenex formulas (in Russian). *Zapiski Nauchnyh Seminarov LOMI*, 60:109–170, 1976. English translation in: Journal of Soviet Mathematics.

[26] D.A. Plaisted. Special cases and substitutes for rigid E-unification. Technical Report MPI-I-95-2-010, Max-Planck-Institut für Informatik, November 1995.

[27] D. Prawitz. An improved proof procedure. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning. Classical Papers on Computational Logic*, volume 1, pages 162–201. Springer Verlag, 1983. Originally appeared in 1960.

[28] R. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21:583–585, July 1978.

[29] T. Tammet. A resolution theorem prover for intuitionistic logic. Unpublished manuscript, 1996.

[30] M. Veanes. Uniform representation of recursively enumerable sets with simultaneous rigid E-unification. UPMAIL Technical Report 126, Uppsala University, Computing Science Department, May 1996.

[31] P.J. Voda and J. Komara. On Herbrand skeletons. Technical report, Institute of Informatics, Comenius University Bratislava, July 1995.

[32] A. Voronkov. On proof-search in intuitionistic logic with equality, or back to simultaneous rigid E-Unification. UPMAIL Technical Report 121, Uppsala University, Computing Science Department, January 1996.

[33] A. Voronkov. Proof-search in intuitionistic logic based on the constraint satisfaction. UPMAIL Technical Report 120, Uppsala University, Computing Science Department, January 1996.

Unification in Sort Theories
(Extended Abstract)

Christoph Weidenbach
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken, Germany
email: weidenb@mpi-sb.mpg.de
phone: +49-681-9325221
fax: +49-681-9325299

May 23, 1996

1 Introduction

In this paper we investigate unification in different sort theories. The starting point is the approach of Schmidt-Schauß [6] to order-sorted unification. He considered sort theories that consist of subsort declarations and arbitrary term declarations. A sort corresponds to a monadic predicate that is a priori assumed to be non-empty. An example for such a sort theory would be:

$$\Sigma = \{Q \sqsubseteq T, f{:}\,T \to T, a{:}\,T, f{:}\,S \to S, a{:}\,S\}$$

where Q is declared to be a subsort of T, f a one-place function that maps terms of sort T and sort S to terms of sort T and sort S, respectively and a is constant contained in both sorts T and S. The sort theory Σ is called *elementary*: All declarations are either subsort or function declarations, i.e., there is no real term declaration. Now given the unification problem

$$\Gamma = (x_S \approx y_T)$$

the unification algorithm presented by Schmidt-Schauß yields infinitely many well-sorted (ground) mgus with respect to Σ:

$$\begin{array}{rcl} \sigma_1 & = & \{x_S/a, y_T/a\} \\ \sigma_2 & = & \{x_S/f(a), y_T/f(a)\} \\ \sigma_3 & = & \{x_S/f(f(a)), y_T/f(f(a))\} \\ & & \vdots \end{array}$$

In general, Schmidt-Schauß showed that unification in elementary sort theories is decidable, *NP*-complete and of unification type [7] infinitary. If arbitrary term declarations are allowed, e.g., declarations of the form $f(g(x_T, f(x_T))) : S$, sorted unification becomes undecidable, but is still of unification type infinitary [6]. Until now, Uribe [8] has shown that unification in semi-linear sort theories is decidable and *NP*-complete. A sort theory is called semi-linear if, roughly speaking, non-linear occurrences of variables in term declarations occur in identical subterms.

In this paper we will generalize the approach of Schmidt-Schauß:

(i) Sorts are extended to be sets of monadic predicates, denoting their respective intersection.

(ii) Sorts may denote the empty set.

Considering sorts to be sets of monadic predicates (also called *base sorts*), meaning their intersection has many advantages: It naturally introduces a top-sort, the empty set denoted by $\top$, and therefore avoids such concepts like subterm-closedness. In addition, we can now always derive a unique unifier for two variables that is a new variable with the union of the sorts of the variables to be unified. For the above example we obtain the single mgu

$$\sigma = \{x_S/z_{\{S,T\}}, y_T/z_{\{S,T\}}\}$$

where we write S for the singleton set $\{S\}$ as a shorthand. We also use a different notation for sort theories that is closer to first-order logic. We write

$$\mathcal{L} = \{T(u_Q), T(f(x_T)), T(a), S(f(y_S)), S(a)\}$$

for the above sort theory Σ, where all variables in $\mathcal{L}$ are universally quantified. The reason for the different notation is closely related to our motivation for extension (ii): We want to increase the applicability of sorted unification to automated theorem proving. In automated theorem proving, a problem is usually given by a set of clauses, without any explicitly given sort information. The contained monadic predicates that are the natural candidates for sorts, may denote the empty set and these predicates may occur in an arbitrary way in clauses.

Nevertheless, it is possible to a posteriori extract sort information. For example, using the relativization [6, 11] rules for sorted formulae we know that the standard[1] clause

$$\neg T(x) \vee \neg S(x) \vee \neg R(x, y) \vee S(f(x))$$

is logically equivalent to the clause

$$\neg R(x_{\{S,T\}}, y) \vee S(f(x_{\{S,T\}}))$$

where sorts are attached to variables. Negative monadic literals with a variable as their argument code sort restrictions on the variable. The variable y has top-sort, i.e., y is

[1]With the term "standard" we refer to notions without sorts.

interpreted like a standard variable, ranging over the whole domain. The problem remains how the sort theory is extracted from the clause set. We have shown that for the tableau calculus, it is sufficient to consider the sort theory built from the monadic atoms on the branch of the tableau we want to close [11]. For the resolution calculus [10], it is sufficient to *dynamically* choose from each clause consisting of positive monadic literals only, one positive literal for the sort theory. Hence, it is sufficient to consider sort theories of the form of $\mathcal{L}$ where sorts may denote the empty set. The application of sorted unification is not a topic of this paper, however, it is the motivation for our approach to sorted unification.

In the following we will present the rules of sorted unification together with some complexity results:

1. Sorted unification for weak-elementary (see Section 2) sort theories is decidable, *NP*-complete and of unification type finitary.

2. Sorted unification in linear and semi-linear sort theories is decidable, *NP*-complete and of unification type finitary.

3. Sorted unification is pseudo-linear sort theories is decidable and of unification type infinitary.

4. Sorted unification in non-linear sort theories where the depth difference of non-linear variable occurrences is at most one, is undecidable.

5. Rigidly sorted unification is decidable and of unification type finitary.

In particular, the results 3 and 4 identify a new border between decidable and undecidable problems: If all non-linear occurrences of variables in the sort theory occur at the same depth sorted unification is decidable, if there is a difference in the depths of these variables of at most one, sorted unification is undecidable. A rigid sort theory is a sort theory where all variables are considered to be free variables, not universally quantified variables. Such sort theories play an important role when sorted unification is applied to free variable tableau [11].

2 Sorted Unification

We use the usual notation for terms, substitutions, etc. The set of all terms is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$ or $\mathcal{T}$ as a shorthand and the set of all ground terms is denoted by $\mathcal{T}(\mathcal{F})$. Let $\mathcal{S}$ be a finite set of monadic predicate symbols, also called *base sorts*. Then a *sort* T is a finite subset of $\mathcal{S}$, $T \in 2^{\mathcal{S}}$, denoting the intersection of the contained base sorts. For variables we assume a function $\mathit{sort}\colon \mathcal{V} \to 2^{\mathcal{S}}$ that attaches sorts to variables such that for each sort $T \in 2^{\mathcal{S}}$ there are countably many variables x with $\mathit{sort}(x) = T$. For $\emptyset \in 2^{\mathcal{S}}$, we write $\top$, the top sort. We write S for $\{S\}$ if the context causes no confusion. Usually, variables are annotated with their sort. We write x_S if $\mathit{sort}(x) = S$.

Syntactic properties of terms will play an important role for the complexity of sorted unification: A term t is called *elementary* if t is either a variable or a constant or of the form $f(x_1, \ldots, x_n)$ where all x_i are different. A term t is called *weak-elementary* if t is either a variable or a constant or of the form $f(t_1, \ldots, t_n)$ where each t_i is either a variable[2] or a ground term. The term t is called *linear* if any variable occurs at most once in t. It is called *semi-linear* if for any variable x occurring more than once in t, there is a subterm $f(s_1, \ldots, s_n)$ of t such that all occurrences of x are in $f(s_1, \ldots, s_n)$, each s_i has at most one occurrence of x and whenever x occurs in s_i and s_j we have $s_i = s_j$. A term t is called *pseudo-linear* if all non-linear variable occurrences in t occur at the same depth.

A *unification problem* Γ is either $\top$, $\bot$ or a conjunction of pairs $\Gamma = (t_1 \approx s_1 \wedge \ldots \wedge t_n \approx s_n)$. A substitution σ *solves* a unification problem $\Gamma = (t_1 \approx s_1 \wedge \ldots \wedge t_n \approx s_n)$, iff $t_1\sigma = s_1\sigma \wedge \ldots \wedge t_n\sigma = s_n\sigma$. A unification problem $\Gamma = (s_1 \approx t_1 \wedge \ldots \wedge s_n \approx t_n)$ is called *dag solved*, iff all s_i are variables, $s_i \neq s_j$ for every i and j with $i \neq j$ and $s_i \notin vars(t_j)$ for all i, $j \geq i$. If $\Gamma = (x_1 \approx t_1 \wedge \ldots \wedge x_n \approx t_n)$ is a dag solved unification problem, then $\Gamma^* = \sigma_1\sigma_2 \ldots \sigma_n$ with $\sigma_i = \{x_i/t_i\}$ is the unifier induced by Γ.

We start with a standard dag oriented unification algorithm, similar to the rule-based standard unification algorithms presented by Dershowitz and Jouannaud [4] or Jouannaud and Kirchner [5]. The rules are shown in Figure 1.

These rules are exhaustively applied (with respect to the commutativity and associativity of $\wedge$) to a unification problem Γ until Γ is dag solved or $\top$, $\bot$ is derived. The rule Orientation is not contained in other rule sets for standard unification [4, 5]. Therefore, the rules of these sets are also apply with respect to the commutativity of $\approx$. However, we will extend our standard unification to cope with the sort information of variables. Then it is necessary to distinguish between the pair $x \approx y$ and $y \approx x$, because these variables may have different sorts.

An atom $S(t)$ is called a *declaration* if $S \in \mathcal{S}$. It is called a *subsort declaration* if t is a variable and a *term declaration* otherwise. It is called a *function declaration* if t is elementary. A *sort theory* $\mathcal{L}$ is a finite set of declarations where all variables are implicitly assumed to be universally quantified. All following notions and definitions refer to a fixed, finite sort theory $\mathcal{L}$. The *size* of a sort theory $\mathcal{L}$ is $size(\mathcal{L}) = \sum_{S(t)\in\mathcal{L}}(size(t) + \sum_{x\in vars(t)} |sort(x)|)$.

Depending on the declarations occurring in $\mathcal{L}$, the following sort theories are distinguished: $\mathcal{L}$ is called *elementary* (*weak-elementary*) if every term in a term declaration in $\mathcal{L}$ is elementary (weak-elementary). $\mathcal{L}$ is called *linear* (*semi-linear*, *pseudo-linear*) if every term occurring in a term declaration is linear (semi-linear, pseudo-linear).

Definition 2.1 *The set of well-sorted terms $\mathcal{T}_S$ of sort S is recursively defined by:*

1. $x \in \mathcal{T}_S$ — *if* $S \subseteq sort(x)$.
2. $t \in \mathcal{T}_S$ — *if* S *is a base sort and* $S(t) \in \mathcal{L}$.
3. $t\sigma \in \mathcal{T}_S$ — *if* $t \in \mathcal{T}_S$ *and* $x\sigma \in \mathcal{T}_{sort(x)}$ *for all* $x \in dom(\sigma)$.
4. $t \in \mathcal{T}_{S_1\cup\ldots\cup S_n}$ — *if* $t \in \mathcal{T}_{S_i}$ *for all* i.

[2]Multiple occurrences of the same variable are allowed.

Tautology

$t \approx t \wedge \Gamma \to \Gamma$

Orientation

$t \approx x \wedge \Gamma \to x \approx t \wedge \Gamma$

if $t \notin \mathcal{V}$

Decomposition

$f(t_1, \ldots, t_n) \approx f(s_1, \ldots, s_n) \wedge \Gamma \to t_1 \approx s_1 \wedge \ldots \wedge t_n \approx s_n \wedge \Gamma$

Application

$x \approx y \wedge \Gamma \to x \approx y \wedge \Gamma\{x/y\}$

if $x \in \mathit{vars}(\Gamma)$

Clash

$f(t_1, \ldots, t_n) \approx g(s_1, \ldots, s_m) \wedge \Gamma \to \bot$

if $f \neq g$

Cycle

$x_1 \approx t_1[x_2]_{p_1} \wedge \ldots \wedge x_n \approx t_n[x_1]_{p_n} \wedge \Gamma \to \bot$

if there exists some i, $1 \leq i \leq n$, with $p_i \neq \epsilon$

Merge

$x \approx t \wedge x \approx s \wedge \Gamma \to x \approx t \wedge t \approx s \wedge \Gamma$

if $t, s \notin \mathcal{V}$ and $\mathit{depth}(t) \leq \mathit{depth}(s)$

Figure 1: The Rules of Dag Standard Unification

Sorted Fail

$x \approx f(t_1, \ldots, t_n) \wedge \Gamma \rightarrow \bot$

if $\top \not\sqsubseteq sort(x)$ and there is no sort $\{T_1, \ldots, T_m\} \sqsubseteq sort(x)$ with declarations $T_j(f(s_{j,1}, \ldots, s_{j,n})) \in \mathcal{L}$

Subsort

$x \approx y \wedge \Gamma \rightarrow x \approx z \wedge y \approx z \wedge \Gamma$

if $sort(x) \not\sqsubseteq sort(y)$, z new to Γ, and $sort(z) = sort(x) \cup sort(y)$

Weakening

$x \approx t \wedge \Gamma \rightarrow x \approx s_1 \wedge t \approx s_1 \wedge \ldots \wedge t \approx s_m \wedge \Gamma$

if $x \approx t \wedge \Gamma$ is dag solved, $t \notin \mathcal{V}$, $t\Gamma^* \notin \mathcal{T}_{sort(x)}$, $\{T_1, \ldots, T_m\} \sqsubseteq sort(x)$ and for each T_j there is a declaration $T_j(s_j) \in \mathcal{L}$ such that t and the s_j share the same top symbol

Figure 2: The Rules of Dag Sorted Unification

Note that there are only declarations for base sorts in $\mathcal{L}$. Case 4 also includes the case where $S = \top$, i.e., $n = 0$; then we have $\mathcal{T}_\top = \mathcal{T}$. $\mathcal{T}(\mathcal{F})_S$ is the restriction of $\mathcal{T}_S$ to ground terms. A sort S is called *empty* if there is no well-sorted ground term $t \in \mathcal{T}_S$, or equivalently if $\mathcal{T}(\mathcal{F})_S = \emptyset$. We always have $\mathcal{T}(\mathcal{F})_S \subset \mathcal{T}_S$. The binary relation $\sqsubseteq$ denotes the subsort relationship. If S and T are sorts, then we define $S \sqsubseteq T$ iff there exists a variable x_S with $x_S \in \mathcal{T}_T$. Note that if there exists one variable $x_S \in \mathcal{T}_T$, there are infinitely many variables of sort S in $\mathcal{T}_T$. The relation $S \sqsubseteq T$ implies $\mathcal{T}_S \subseteq \mathcal{T}_T$, but the relation $\mathcal{T}(\mathcal{F})_S \subseteq \mathcal{T}(\mathcal{F})_T$ neither implies $S \sqsubseteq T$ nor $\mathcal{T}_S \subseteq \mathcal{T}_T$. A substitution σ is *well-sorted* if for every $x \in dom(\sigma)$, $x\sigma \in \mathcal{T}_{sort(x)}$. It can be polynomially decided (in $size(\mathcal{L}) + size(\sigma)$) whether a substitution σ is well-sorted.

Now we extend standard unification (Figure 1) to sorted unification. A sorted unification problem $\Gamma = (x_1 \approx t_1 \wedge \ldots \wedge x_n \approx t_n)$ is called *dag sorted solved*, iff Γ is dag solved and $t_i\Gamma^* \in \mathcal{T}_{sort(x_i)}$ for all i. Since $dom(\Gamma^*) = \{x_1, \ldots, x_n\}$ the induced substitution Γ^* is well-sorted. The sorted unification algorithms consist both of the three don't care non-deterministic sorted rules Sorted Fail, Subsort and Weakening (see Figure 2) and the rules of standard unification (see Figure 1). These combined rule sets are applied to a unification problem Γ until it is solved or $\top$, $\bot$ is derived.

In rule Subsort the condition $sort(x) \not\sqsubseteq sort(y)$ does not imply $sort(y) \not\sqsubseteq sort(x)$. However, if $sort(y) \sqsubseteq sort(x)$ then $\mathcal{T}_{sort(x)} = \mathcal{T}_{sort(x) \cup sort(y)}$. Note that declarations must be well-sorted renamed, before they are used by the Weakening rule. The condition $t\Gamma^* \notin \mathcal{T}_{sort(x)}$ can be checked in polynomial time although Γ^* may increase exponentially in size of Γ. The idea is first to compute the sorts of all terms in Γ bottom up according to the variable dependencies in Γ.

Due to the rule Weakening, sorted unification does not terminate, in general. The

Weakening rule introduces terms from $\mathcal{L}$ with fresh variables. Therefore, the number of variables and the multiset of all term depths in the unification problem may increase. In addition, the number of sorted unsolved equations may increase, too. In general, sorted unification, as well as the problem whether a sort is empty, is undecidable. The Weakening rule, together with the other rules can simulate arbitrary computational processes.

It can be shown that for every unification problem Γ and finite sort theory $\mathcal{L}$, there exists a minimal, complete set of unifiers $\mu U_{\mathcal{L}}(\Gamma)$ and that sorted unification (Figure 1 and Figure 2) is correct and complete with respect to the usual semantics [9].

Concerning the complexity issues we start with weak-elementary sort theories. These properly include elementary sort theories where sorted unification is known to be decidable, *NP*-complete and of unification type finitary [6].

Theorem 2.2 *Unification in weak-elementary sort theories is decidable,* NP*-complete and of unification type finitary. The problem whether a sort is empty, is decidable and* NP*-complete, too.*

PROOF: Let $\mu(\Gamma)$ be the multiset of all term depths of pairs $x \approx t$, such that t is not ground and $t\Gamma^* \notin \mathcal{T}_{sort(x)}$. For any pair $x \approx t$ where t is ground, either $t \in \mathcal{T}_{sort(x)}$ or Γ is not solvable. Therefore these pairs can be disregarded. Now $\mu(\Gamma)$ always decreases after the sequence of a Weakening step followed by the exhaustive application of all other rules. The number of possible Weakening steps is bound by the number of function symbols in the initial unification problem. □

Theorem 2.3 *Unification in linear (semi-linear) theories is decidable,* NP*-complete and of unification type finitary. The problem whether a sort is empty, is decidable and* NP*-complete, too.*

PROOF: Linear and semi-linear sort theories can be transformed in linear time into weak-elementary sort theories. □

The following example shows a pseudo-linear sort theory with a unification problem leading to infinitely many mgus. Consider the pseudo-linear sort theory

$$\mathcal{L} = \{S(g(x_S)), S(h(x_S)), T(g(x_S)), S(f(g(x_S), h(x_S))), S(f(g(y_S), f(y_S, x_S)))\}$$

and the unification problem

$$\Gamma = (z_S \approx f(v_T, w_S))$$

Γ is solved but not sorted solved because $f(v_T, w_S) \notin \mathcal{T}_S$. The rule Weakening is applicable using the two declarations $S(f(g(x_S), h(x_S)))$ and $S(f(g(y_S), f(y_S, x_S)))$, respectively:

$$\begin{array}{lcl} \Gamma_1 & = & (z_S \approx f(g(x_S), h(x_S)) \wedge f(v_T, w_S) \approx f(g(x_S), h(x_S))) \\ \Delta_1 & = & (z_S \approx f(g(y_S), f(y_S, x_S)) \wedge f(v_T, w_S)) \approx f(g(y_S), f(y_S, x_S))) \end{array}$$

The two problems can be transformed into standard solved form:

$$\begin{array}{rcl}\Gamma_2 & = & (z_S \approx f(g(x_S), h(x_S)) \wedge v_T \approx g(x_S) \wedge w_S \approx h(x_S)) \\ \Delta_2 & = & (z_S \approx f(g(y_S), f(y_S, x_S)) \wedge v_T \approx g(y_S) \wedge w_S \approx f(y_S, x_S))\end{array}$$

The problem Γ_2 is already sorted solved. The problem Δ_2 contains the unsolved pair $w_S \approx f(y_S, x_S)$ to which we apply Weakening using the declarations $S(f(g(x'_S), h(x'_S)))$ and $S(f(g(y'_S), f(y'_S, x'_S)))$, respectively. We only show the pairs resulting from the Weakening step.

$$\begin{array}{rcl}\Delta_{31} & = & (w_S \approx f(g(x'_S), h(x'_S)) \wedge f(y_S, x_S) \approx f(g(x'_S), h(x'_S))) \\ \Delta_{41} & = & (w_S \approx f(g(y'_S), f(y'_S, x'_S)) \wedge f(y_S, x_S) \approx f(g(y'_S), f(y'_S, x'_S)))\end{array}$$

After standard unification we get

$$\begin{array}{rcl}\Delta_{32} & = & (w_S \approx f(g(x'_S), h(x'_S)) \wedge y_S \approx g(x'_S) \wedge x_S \approx h(x'_S)) \\ \Delta_{42} & = & (w_S \approx f(g(y'_S), f(y'_S, x'_S)) \wedge y_S \approx g(y'_S) \wedge x_S \approx f(y'_S, x'_S))\end{array}$$

Now Δ_{32} is sorted solved and Δ_{42} contains the sorted unsolved pair $x_S \approx f(y'_S, x'_S)$, a variant of the pair $w_S \approx f(y_S, x_S)$ in Δ_2. Hence, sorted unification runs through a "cycle" producing infinitely many mgus for the initial problem Γ:

$$\begin{array}{rcl}\sigma_1 & = & \{z_S/f(g(x_S), h(x_S)), v_T/g(x_S), w_S/h(x_S)\} \\ \sigma_2 & = & \{z_S/f(g(g(x'_S)), f(g(x'_S), h(x'_S))), v_T/g(g(x'_S)), w_S/f(g(x'_S), h(x'_S))\} \\ \sigma_3 & = & \{z_S/f(g(g(g(x''_S))), f(g(g(x''_S)), f(g(x''_S), h(x''_S)))), v_T/g(g(g(x''_S))), \\ & & \quad w_S/f(g(g(x''_S)), f(g(x''_S), h(x''_S)))\} \\ & & \vdots\end{array}$$

However, it can be shown that such "cycles" are not necessary to test unifiability, i.e., whenever a unification problem is solvable, there exists a solution without a "cycle".

Theorem 2.4 *Unification in pseudo-linear sort theories is decidable and of unification type infinitary.*

Theorem 2.5 *Unification in sort theories with non-linear term declarations, where the depth difference between different occurrences of the same variable is at most one, is undecidable.*

PROOF: Reduction to the Halting problem. □

So far, we assumed the variables in the sort theory $\mathcal{L}$ to be universally closed. This point of view is appropriate to extend resolution with sorts. For free variable tableau, however, this is not the case. Then the variables in $\mathcal{L}$ are free variables, i.e., they can only be instantiated once. These requirements lead to the following definitions and results.

Definition 2.6 *A substitution σ is called* rigidly well-sorted *with respect to $\mathcal{L}$ if σ is well-sorted and if there exists a substitution τ with $\sigma \subseteq \tau$, $dom(\tau) \subseteq (dom(\sigma) \cup vars(\mathcal{L}))$ and for all $x \in dom(\tau)$, $sort(x) \neq \top$:*

1. *If $x\tau \in \mathcal{V}$, then either $sort(x) \subseteq sort(x\tau)$ or for all $S \in (sort(x) \setminus sort(x\tau))$ we have $S(x\tau) \in \mathcal{L}\tau$*
2. *If $x\tau \notin \mathcal{V}$, then $S(x\tau) \in \mathcal{L}\tau$ for all $S \in sort(x)$*

Consider the sort theory $\mathcal{L} = \{S(a), S(f(x_S))\}$. The substitution $\sigma = \{u_S/f(f(v_S))\}$ is well-sorted but not rigidly well-sorted. It is not rigidly well-sorted because the declaration $S(f(x_S))$ is needed twice with different instantiations to establish the well-sortedness of σ. With respect to the sort theory $\mathcal{L}' = \{S(a), S(f(x_S)), S(f(y_S))\}$ the substitution σ is rigidly well-sorted where $\tau = \{x_S/f(v_S), y_S/v_S\}$ is the substitution which instantiates $\mathcal{L}'$ in the appropriate way.

The usual notions of a unifier and a most general unifier transfer from sorted unification to the case of rigidly well-sorted substitutions. Note that σ is a rigidly well-sorted unifier with respect to $\mathcal{L}$ iff σ is a rigidly well-sorted unifier with respect to $\mathcal{L}\sigma$. The sorted unification algorithm (Figure 2) can be modified for rigidly well-sorted substitutions. Instead of renaming declarations from $\mathcal{L}$ before they are used in the rule Weakening or in the computation of well-sortedness, only the declarations in $\mathcal{L}$ are used and instantiated accordingly. In addition, the rule Application is also applied to $\mathcal{L}$.

Theorem 2.7 *Rigidly sorted unification is decidable and of unification type finitary. The empty sort problem is decidable.*

3 Related Work

There is a close relationship between sorted unification and regular tree languages [3]. The ground terms induced by elementary sort theories are regular tree languages. The ground terms of weak-elementary sort theories can be described by tree automata with equality constraints on direct subterms, as shown by Bogaert and Tison [1]. Recently, from results on tree automata presented by Caron et al. [2], it can be shown that sorted unification is decidable for arbitrary term declarations if all declarations satisfy an ordering restriction. We assume a partial ordering on the sort symbols and require that whenever a declaration contains non-linear occurrences of variables, the sorts attached to the variables in the declaration are strictly below the sort symbol of the declaration itself. For example, for the declaration $S(f(x_T, g(x_T)))$ we would require that T is strictly below S in the ordering. However, there are operational differences between algorithms on tree automata and sorted unification. For tree automata two things usually exists: Algorithms computing boolean combinations and a non-emptiness test deciding whether the language of an automaton is empty. In general, the non-emptiness test of the automaton corresponds to the non-emptiness test for a sort. Boolean combinations of sorts cannot be expressed inside the framework of sorted unification, in general. On the other hand, sorted unification computes a complete set of unifiers for a specific unification problem of arbitrary terms that do not necessarily satisfy the syntactic restrictions on the sort theory.

References

[1] Bruno Bogaert and Sophie Tison. Equality and disequality constraints on direct subterms in tree automata. In A. Finkel and M. Jantzen, editors, *Proc. of 9th Annual Symposium on Theoretical Aspects of Computer Science, STACS92*, volume 577 of *LNCS*, pages 161–171. Springer, 1992.

[2] Anne-Cécile Caron, Hubert Comon, Jean-Luc Coquidé, Max Dauchet, and Florent Jacquemard. Pumping, cleaning and symbolic constraints solving. In Serge Abiteboul and Eli Shamir, editors, *Automata Languages and Programming. 21st International Colloquium, ICALP'94*, volume 820 of *LNCS*, pages 436–447. Springer, 1994.

[3] Hubert Comon. Inductive proofs by specification transformations. In *Rewriting Techniques and Applications, RTA-89*, volume 355 of *LNCS*, pages 76–91. Springer, 1989.

[4] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier Science Publishers, 1990.

[5] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, chapter 8, pages 257–321. MIT Press, 1991.

[6] Manfred Schmidt-Schauß. *Computational aspects of an order sorted logic with term declarations*, volume 395 of *LNAI*. Springer, 1989.

[7] Jörg Siekmann. Unification theory. *Journal of Symbolic Computation, Special Issue on Unification*, 7:207–274, 1989.

[8] Tomás E. Uribe. Sorted unification using set constraints. In *11th International Conference on Automated Deduction, CADE-11*, volume 607 of *LNCS*, pages 163–177. Springer, 1992.

[9] Christoph Weidenbach. Unification in sort theories and its applications. *Annals of Mathematics and Artificial Intelligence*. To appear.

[10] Christoph Weidenbach. Extending the resolution method with sorts. In *Proc. of 13th International Joint Conference on Artificial Intelligence, IJCAI-93*, pages 60–65. Morgan Kaufmann, 1993.

[11] Christoph Weidenbach. First-order tableaux with sorts. *Journal of the Interest Group in Pure and Applied Logics, IGPL*, 3(6):887–906, 1995.

Some Related Cases of Infinite Unification and Matching

Marisa Venturini Zilli

We show how infinite unification, infinite matching and unification of infinite sets of terms can be related.

Complexity of Term Schematizations

Gernot Salzer

We investigate the structural complexity of unifying term schematizations. Using the 1-in-3-SAT problem we show that unification is NP-hard even when no Diophantine equations have to be solved. To obtain an upper bound we give an algorithm for solving the word problem for term schematizations. Our aim is to prove unification to be in NP (assigning unit cost to the solution of linear Diophantine equations). Currently this claim has the status of a conjecture since parts of the proof are still missing.

Polynomial Non-AC-Unifiability and Matching Filters

Robert Nieuwenhuis and Jose Miguel Rivero
Technical University of Catalonia
Pau Gargallo 5, 08028 Barcelona, Spain
E-mail: {roberto,rivero}@lsi.upc.es

As a consequence of previous work by Nieuwenhuis and Rubio, it seems that efficient automated deduction strategies with built-in associativity and commutativity properties (AC) for some operators could be obtained by dealing with AC-unifiability constrained formulae. Here we address ongoing work on some of the practical aspects appearing in actual implementations of such strategies: efficient sound tests for detecting cases of non-AC-unifiability (the full decision problem is well-known to be NP-hard) as well as some ideas for practical methods for AC-matching. In particular, we provide benchmarks for several cases.

Matching and Unification with Compiled Substitution Trees

Robert Nieuwenhuis, Jose Miguel Rivero, and Miguel Angel Vallejo
Technical University of Catalonia
Pau Gargallo 5, 08028 Barcelona, Spain
E-mail: {roberto,rivero,vallejo}@lsi.upc.es

We provide a standard abstract architecture around which high-performance theorem provers for full clausal logic with equality can be built. A WAM-like heap structure for storing terms (as DAG's, with structure sharing) and several substitution trees [Graf RTA95] are central in the architecture. These two data structures turn out to be surprisingly well combinable due to conceptual similarities. Indexing techniques based on substitution trees outperform previous methods, and are integrated in such a way that e.g. no writing on the heap is needed during (many-to-one) term unification. Static clause (sub)sets can be compiled in this framework into efficient abstract machine code for inference computation and redundancy proving. We provide benchmarks for several typical operations.

Results about RUE Resolution

Ulf Dunker and Annette Müller
University of Paderborn
33095 Paderborn (Germany)
`dunker@uni-paderborn.de` **and** `nette@uni-paderborn.de`

1 Introduction

Based on an implementation of the first-order resolution calculus, which was realized within Paderborn's C library Logic*AL* [DFKBLL94], we were interested in extending the algorithm for processing equality. In [BlB92] we found the approach of Digricoli: RUE [Di79], which stands for Resolution by Unification and Equality. With general E-resolution a resolution step is possible only if the differences of terms in corresponding literals can be fully resolved. Principally RUE allows resolution steps at any time. The difference between unresolved terms, which defines a distance between them, is represented within the new derived clause by *disagreement sets*, a series of unequalities. So, unresolved terms can be handled later in order to reduce their distance. For unifying two terms in a literal $s \neq t$, instead of applying paramodulation Digricoli also suggests the idea of reducing their distance in a similar way. The non-deterministic RUE procedure is correct and complete.

2 RUE is Incomplete

When applying RUE, in each step a disagreement set and a substitution have to be chosen. In order to obtain a deterministic procedure which can be implemented, Digricoli defines some *strategies* for this.

At first he defines the *viability* of a disagreement set. The viability is a necessary condition that there is a sequence of equalities, so that the unequalities of the disagreement set can be resolved. Two terms may lead to more than one viable disagreement set. That one which terms have the lowest term depth is called the *topmost viable disagreement* set. For deriving new clauses, all disagreement sets, all viable disagreement sets, or just the topmost one are possible choices. The viability test is described in [DLS89].

Secondly, instead of trying each of the possibly infinite number of applicable substitutions, three special substitutions are suggested. The *most general partial unifier* (mgpu), the *RUE unifier* which applies the viability test, and the *2-scan unifier* which is another special partial unifier.

In [BoHa92] it is shown that deterministic RUE resolution using these strategies is not complete without applying the functional reflexivity axiom $f(X) = f(X)$. So RUE doesn't seem to be a reasonable theorem proving procedure for problems with equality.

3 Heuristics

Additionally to the strategies of chosing disagreement sets and substitutions, reducing the number of possible resolution steps can be made by restricting which parent clauses can be selected for resolution (N-, P, Set of Support resolution, ...) or by restricting the maximal clause length or the maximal term depth. We want to introduce another heuristic.

If two literals $t_1 = t_2$ und $t_3 \neq t_4$ should be resolved, the pairs t_1, t_3 and t_2, t_4 have to be tested whether they are unifiable. If this test fails, the pairs t_1, t_4 and t_2, t_3 also have to be tested because of the symmetry. The idea is to sort all equality arguments according to a given term order and to process only the first test without loosing too much information.

We chose a term oder known from term rewriting systems: $t_1 < t_2 \Leftrightarrow \omega(t_1) < \omega(t_2)$ where $\omega(t) = \omega_1$ if t is a constant, $\omega(t) = \omega_2$ if t is a variable, and if $t = f(t_1, \ldots, t_n)$, $\omega(t) = \omega_3 + \sum_{i=1}^{n} \omega(t_i)$.

The intention is that if the distance $|\omega(t_1) - \omega(t_2)|$ is large, the probability to unify t_1 and t_2 is small. It can be shown, that the order $\omega_1 < \omega_2 < \omega_3$ theoretically is the best choice, e.g. $\omega_1 = 1$, $\omega_2 = 3$, and $\omega_3 = 5$.

The viability test can also be equiped with several heuristics.

4 Results

For the implementation, the resolution algorithm of LogicAL was extended by paramodulation and RUE with the described strategies and heuristics. In order to evaluate the large number of possible settings for the resulting algorithm, we used the TPTP formula library [SS94] for runtime tests. The TPTP contains 1831 unsatisfiable CNF formulas with equality mainly contained in ten problem classes of the TPTP. We used N-resolution and a maximal runtime per formula of ten minutes, i.e. maximal twelve days per test.

4.1 Best Strategy

At first we wanted to find the best RUE strategy (the values are numbers of solved formulas or numbers of formulas decided as satisfiable (errors) absolute and in percent)

- Testing the choice of disagreement sets with RUE using mgpu:
  - Topmost viable disagreement sets: 79 (4%) solved, 89 (5%) wrong
  - All viable disagreement sets: 685 (37%) solved, 84 (5%) wrong
  - All disagreement sets: 763 (42%) solved, 3 (0.1%) wrong

  So Digricoli's viability test leads to a large amount of wrong decisions without improving the procedure. Trying all disagreement sets is the best strategy and though it's incomplete, 0.1% is a low error rate.

- Testing the choice of substitution with RUE trying all disagreement sets:

  - All three substitutions (mgpu, RUE unifier, and 2-scan unifier) lead to 3 (0.1%) wrong decisions.
  - Also the number of solved formulas is nearly the same: 763 (42%) mgpu, 742 (41%) RUE unifier, 765 (42%) 2-scan unifier
  - The best runtime was obtained with the 2-scan unifier

  Therefore the strategy of trying all disagreement sets with the 2-scan unifier leads to the best results.

Remark: Our paramodulation version solves 508 (28%) of the formulas, of course without errors because resolution with paramodulation is complete. But with only three wrong decisions RUE solves 50% more problems.

4.2 Best Heuristic

Secondly we tryed to find the best RUE heuristic. We found that all heuristics shows advantages in some problem classes and disadvantages in others. With restricting the clause length to five, the term depth to eight, and using the term order mentioned before, the number of solved formulas can be raised to 778 (42%) where 4 (0.2%) wrong decisions were made. The term oder which was theoretically shown to be the best also delivers the best runtime results.

4.3 Otter

In order to get a more global result about the efficiency of RUE, we compared the results with those obtained by OTTER, a famous resolution based theorem prover developed in C by W. McCune [McC94]. We tested three OTTER configurations: Standard (positive hyper resolution, paramodulation), Knuth-Bendix (positive hyper resolution, Knuth-Bendix completion), and Autonomous (the prover chooses the strategy for each formula).

The Autonomous configuration delivers the best runtime results. OTTER solved 618 formulas (34%). So RUE solved 24% more formulas than OTTER. When looking at the runtime, on the average, RUE solved the problems faster.

5 Conclusion

RUE was shown to be an efficient theorem proving procedure. The best strategy is to try all disagreement sets with the 2-scan unifier. The rate of wrong decisions is low (0.1%). Additional heuristics, especially the term order approach, can help to solve more formulas without producing much more wrong decisions. In relation to OTTER, with RUE better results with a better runtime behaviour can be obtained for the TPTP formulas. But remember that when RUE decided satisfiable, you cannot be sure that the decision is correct.

References

[BlB92] K.H. Bläsius, H.-J. Bürckert: *Deduktionssysteme*, R. Oldenbourg, 2. Edition, (1992) [BoHa92] M. P. Bonacina, J. Hsiang: *Incompleteness of RUE/NRF Inference Systems*, Scientific note, State University of New York, Stony Brook (1992)

[Di79] V. J. Digricoli: Resolution by Unification and Equality, *Proc. 4th Workshop on Automated Deduction*, Texas (1979), pp. 43-52

[Di79a] V. J. Digricoli: Automated Deduction and Equality, *Proc. National ACM Conf.*, 1979, pp. 240-249

[DI80] V. J. Digricoli: First Experiments with RUE automated Deduction, *Proc. 1st National Annual Conf. on Artificial Intelligence* (1980), pp. 96-98

[Di81] V. J. Digricoli: The Efficiency of RUE-Resolution Experimental Results and Heuristic Theory, *Proc. 7th IJCAI* (1981), pp. 539-547

[Di85] V. J. Digricoli: The Management of Heuristic Search in Boolean Experiments with RUE-Resolution, *Proc. 9th IJCAI* (1985), pp. 1154-1161

[DiHa86] V. J. Digricoli, M.C. Harrison: Equality-Based Binary Resolution, *JACM* **33** (1986), pp. 253-289

[DLS89] V. J. Digricoly, J.J. Lu, V.S. Subrahmanian: And-Or Graphs applied to RUE-Resolution, *Proc. 11th IJCAI* (1989), pp. 345-358

[DFKBLL94] U. Dunker, A. Flögel, H. Kleine Büning, J. Lehmann, Th. Lettmann: ILFA - A Project in Experimental Logic Computation, *Technical Report, University of Paderborn* **142** (1994)

[LS93] J. J. Lu, V. S. Subrahmanian: Completeness-Issues in RUE-NRF Deduction, *Journal Automated Reasoning* **10**, (1993), pp.371-388

[McC94] William W. McCune: *OTTER 3.0 Reference Manual and Guide*, Technical Report, Argonne National Laboratory, Argonne, Ill. , **6** (1994)

[SS94] C. B. Suttner, G. Suttcliffe: *The TPTP Problem Library*, Technical Report, Technische Universität München, (1994)

Goal–Directed Completion using SOUR Graphs[*]

Christopher Lynch [†]

May 29, 1996

Abstract

We give the first Goal-Directed version of the Knuth Bendix Completion Procedure. Our procedure is based on Basic Completion and SOUR Graphs. There are two phases to the procedure. The first phase, which runs in polynomial time, compiles the equations and the goal into a constrained tree automata representing the completed system, and a set of constraints representing goal solutions. The second phase starts with the goal solutions and works its way back to the original equations, solving constraints along the way.

1 Introduction

The Knuth Bendix Completion Procedure [7] is the best known procedure for solving word problems and equational unification problems. One reason that it works as well as it does is that it is based on a well-founded ordering. In many cases, this allows it to convert a set of equations into a decision procedure to solve the word problem for its equational theory. This cannot always work, because the word problem is undecidable. The Completion Procedure works by starting with the initial set of equations, continually generating new equations, at each point checking to see if it has discovered a solution to the goal. No matter what the goal is, it always generates the same equations. There has been much research to improve the efficiency of the Completion Procedure, but up until now, nobody has found a way to use the goal to direct the search for solutions of the goal. In this paper, we give the first method for doing that.

The fact that Knuth-Bendix completion is not goal directed means that if a goal is false, and the set of equations has an infinite canonical system, then Knuth-Bendix completion will run forever. A goal directed system could detect that certain critical pairs are not necessary to solve a given goal, and halt and say the goal is not true. It is possible to give heuristics for constructing critical pairs so that equations that are related to the goal are created first. But in order to have a complete system, all critical pairs much be eventually created. So this changes the order in which critical pairs are created, but all of them must be created eventually. It could be argued that the completion procedure can be combined with a narrowing (or rewriting) procedure so that new equations are immediately involved in inferences with the goal. In this sense the completion procedure is more goal sensitive. But it is not goal directed, because it is still the case that all critical pairs must be constructed for a false goal. Also, it is possible to encode the goal as an equation itself, and prove the goal in the completion procedure. However, this process is equivalent to the process of interleaving completion with narrowing, and therefore must construct all critical pairs. In fact, it is worse, because it does not give any priority to the goal. Until now, there have been no modifications to Knuth Bendix completion to allow the process to stop when a goal is false and the canonical set of equations is infinite.

Consider the equational theory $E = \{f(f(x)) = g(f(x))\}$. Suppose we want to know if a and b are equivalent modulo E. The Completion Procedure will generate infinitely many equations from E,

[*]This document was created using Xy-pic[13]. Thanks to Kristoffer Rose for his help with Xy-pic

[†]INRIA Lorraine et CRIN, Campus Scientifique, BP 101, 54602 Villers-lès-Nancy cedex, France, Phone: (33) 83 59 30 88, Fax : (33) 83 27 83 19, email: lynch@loria.fr

but never halt and say the goal is not true, although it is trivial to see that the goal does not follow. The General E-Unification Procedure [14] is an alternative to Completion. It starts from the goal, and creates new subgoals by using the equations. But it has a drawback that it cannot incorporate orderings and make things smaller at each point. Given the theory $E = \{f(a) = a\}$, and the goal $g(a) = h(a)$, it will replace $g(a)$ by $g(f(a))$ by $g(f(f(a)))$ etc. It never halts and says "no". Orderings are crucial for converting theories into decision procedures.

For logic programming, there is a procedure which solves goals by building up from the facts until a goal is reached. If those bottom-up proofs are reversed, we have SLD Resolution, a goal-directed procedure. This is possible because an initial rule is involved in every inference. But in the completion procedure, we cannot turn the proof around and work backwards, because inferences may not involve initial equations, and we have no idea what equations will be derived. In our approach, we handle this by schematizing the equations that will be derived, using constrained tree automata. Recently, there have been some papers showing how to schematize terms (see [6] for references). However these approaches do not show how the schematizations can be created.

Our approach is based on Basic Completion [1, 11], which works by saving unification problems in equational constraints instead of applying most general unifiers. It is even possible to save the ordering problems in constraints [12]. In [9], we defined a graphical theorem proving approach, based on Basic Paramodulation. This procedure was refined to be implemented as a Completion Procedure in [10]. The initial problem is stored in a graph with terms represented as dags. The edges forming the dag are called *subterm edges*, and the edges for the equations are called *rewrite edges*. Inferences are transformations on the graph, which create new subterm and rewrite edges, labelled with the constraint and renaming used in the inference. Inference do not increase the number of nodes in the graph. The graphs are called *SOUR graphs*, because the edges in the graph represent **S**ubterm, **O**rdering, **U**nification, and **R**ewrite relations.

SOUR Graphs can be adapted to a goal directed completion procedure. When we add a new edge to the graph, instead of inheriting the actual constraints and renamings, we just give a reference to the constraints and renamings it is built up of. So there are only finitely many edges added to the graph, in fact only polynomially many, and completion halts in polynomial time (we call this a compilation). It creates a schematization of the completed set of equations and a set of constraints representing goal solutions. After the compilation halts, the procedure starts with the goal solutions and words backwards from the goal to the initial equations, solving constraints along the way. If our technique is restricted to narrowing, we could consider the work of [5, 2] to be special cases. We consider our work to be an extension of [8], where it was first shown how to decide the word problem for ground terms in polynomial time, with a congruence closure algorithm. We show that it is possible to perform congruence closure in polynomial time, even for non-ground terms. The difference is that the result of the congruence closure is a constraint which now must be solved.

The paper is as follows. After some brief definitions we define how to compile a set of equations into a SOUR graph, in polynomial time. Next we give the semantics of SOUR graphs, to show that it truly represents the completion of a set of equations. Then we give inference rules to solve the constraints of the SOUR graph. In the last section, we discuss ways to use the procedure as a decision procedure. The proofs are missing in this abstract, but they are straightforward.

2 Preliminaries

We present necessary definitions briefly. We refer the reader to [4] for a more detailed exposition. Terms are defined inductively from a set of functions symbols. If f is a function symbol of *arity* p, and $t_1, \cdots, t_p$ are terms then $t = f(t_1, ...t_p)$ is a *term*, and we say $top(t) = f$. We consider *variables* and *constants* as arity 0 function symbols. Unifiers and substitutions are defined as usual. A *renaming* is an injective substitution ρ such that every variable is mapped to a variable. The renaming *id* is the identity substitution. A *fresh renaming* is a renaming which maps all variables to new variables. In

SOUR graphs, renamings will be written in the form ρ_i, where i is a positive integer. They can be understood to mean that $x\rho_{i_1}\cdots\rho_{i_m}$ is the same as $x\rho_{j_1}\cdots\rho_{j_m}$ if and only if $m = n$ and $i_k = j_k$ for all k, $1 \leq k \leq n$. For this, we use the concept of a *fresh number*, which means an index number that has not previously appeared.

The symbol $\approx$ (resp. $\not\approx$) is a binary symbol, written in infix notation, representing semantic equality (resp. disequality). The 0-ary symbol $\Box$ will represent solutions to equations. An object is *ground* if it contains no variables. We say $s\sigma \approx t\sigma$ is an *instance* of $s \approx t$. if $s\sigma$ and $t\sigma$ are ground. Let EQ be a set containing equations. We define EQ^* to be the reflexive, symmetric, transitive and congruence closure of all the instances of equations $s \approx t$ in EQ.

Let EQ be a set of equalities and disequalities. We define a function Sub so that $Sub(EQ)$ is the set of subterms in EQ. If $t = f(t_1, \cdots, t_n)$ with $n \geq 0$, then $Sub(t) = \{t\} \cup \bigcup_{1\leq i\leq n} Sub(t_i)$. We define $Sub(s \approx t) = Sub(s) \cup Sub(t)$. A disequation $s \not\approx t$ is viewed as a term with s the first argument of $\not\approx$ and t the second, so $Sub(s \not\approx t) = \{s \not\approx t\} \cup Sub(s) \cup Sub(t)$. We define $Sub(EQ) = \bigcup_{eq\in EQ} Sub(eq)$.

We assume that $<$ is a reduction ordering, total on ground terms. The symbol $\doteq$ is a binary symbol that represents syntactic equality, and $\prec$ is a symbol representing the ordering $<$. A constraint is a conjunction of $\doteq$ and $\prec$ expressions. A constraint $s \doteq t$ is true if $s = t$, $s \prec t$ is true if $s < t$, $E_1 \wedge E_2$ is true if E_1 and E_2 are true, and $\top$ is always true. We say that a substitution σ is a *solution* of a constraint φ if $\varphi\sigma$ is true. If there is a substitution σ such that $\varphi\sigma$ is true, then φ is satisfiable. Otherwise it is *unsatisfiable*. We say $\sigma = mgs(\varphi)$ (σ is a *most general solution* of φ) if σ is a solution of φ, and $\sigma \leq \theta$ for all solutions θ of φ. See [3, 12] for algorithms to find the mgs of a constraint, where $\prec$ represents a lexicographic path ordering.

Let EQ be a set of equations and disequations, where no two disequations in EQ have any variables in common. A set of constraints C is a *complete set of solutions* for EQ if (i) for every solution σ of some $\varphi \in C$, there is a disequation $s \not\approx t$ in EQ such that $s\sigma \approx t\sigma \in EQ^*$, and (ii) for every disequation $s \not\approx t$ in EQ and substitution σ such that $s\sigma \approx t\sigma \in EQ^*$, there is a constraint $\varphi \in C$ such that σ is a solution of φ.

A *constrained term*, (or equation or disequation) $t\,[\![\,\varphi\,]\!]$ is a pair of a term (or equation or disequation) t and a constraint φ. An equation or disequation eq is viewed as the constrained equation or disequation $eq\,[\![\,\top\,]\!]$. The meaning of $t_1\,[\![\,\varphi_1\,]\!] \doteq t_2\,[\![\,\varphi_2\,]\!]$, is $t_1 \doteq t_2 \wedge \varphi_1 \wedge \varphi_2$. The meaning of $f(t\,[\![\,\varphi\,]\!])\,[\![\,\Psi\,]\!]$ is $f(t)\,[\![\,\varphi \wedge \Psi\,]\!]$ The Basic Completion inference rules are the following.

Critical Pair

$$\frac{s \approx t\,[\![\,\varphi_1\,]\!] \qquad u[s'] \approx v\,[\![\,\varphi_2\,]\!]}{u[t\rho] \approx v\,[\![\,s\rho \doteq s' \,\wedge\, s \succ t \,\wedge\, u[s'] \succ v \,\wedge\, \varphi_1\rho \,\wedge\, \varphi_2\,]\!]}$$

where ρ is a renaming substitution and $s' \notin Var$.

Narrowing

$$\frac{s \approx t\,[\![\,\varphi_1\,]\!] \qquad u[s'] \not\approx v\,[\![\,\varphi_2\,]\!]}{u[t\rho] \not\approx v\,[\![\,s\rho \doteq s' \,\wedge\, s \succ t \,\wedge\, \varphi_1\rho \,\wedge\, \varphi_2\,]\!]}$$

where ρ is a renaming substitution and $s' \notin Var$.

Equation Resolution

$$\frac{u \not\approx v\,[\![\,\varphi\,]\!]}{\Box\,[\![\,u \doteq v \,\wedge\, \varphi\,]\!]}$$

If EQ is a set of equations[1] and disequations, let $Clos(EQ)$ represent the closure of EQ under the Basic Completion inference rules. We have the following soundness and completeness result from [12].

[1]Throughout the paper, we assume we are given a set of equations closed under symmetricity.

Theorem 1 *Let EQ be a set of equations and disequations. Let $C = \{\varphi \mid \Box [\![\varphi]\!] \in Clos(EQ)\}$. Then C is a complete set of solutions for EQ.*

3 SOUR Graphs

In this section we show how to create a SOUR graph. An initial SOUR graph is created as the dag representation of a set of equations. From the initial representation a compilation process is performed that creates a schematization of the completed system. The schematization adds new edges to the SOUR graph, but does not add nodes.

A SOUR graph contains nodes which are labelled uniquely by the subterms in the initial set of equations. Each node represents a set of constrained terms. There are two types of edges in a SOUR graph, *subterm edges* and *rewrite edges*. Subterm edges are directed edges labelled by an index number. Some subterm edges are also labelled with a set of pairs, each pair consisting of a subterm edge and a rewrite edge. Rewrite edges are directed edges, some of which are labelled by a set of pairs, each pair consisting of two rewrite edges. Edges that are not labelled with a set of pairs are called *initial* edges. Initially a graph only contains initial edges. As the compilation process is performed, new edges are created due to pairs of existing edges. The sets of pairs for an edge represents the different ways the edge could be created. Therefore, a compiled graph contains a history of its origins. Let u and v be two nodes in the graph. Then there is at most one rewrite edge from u to v. If a rewrite edge exists from u and v, we will call it (R, u, v). If u is labelled by a term with an n-ary top symbol, then for all i such that $1 \leq i \leq n$ there is at most one subterm edge from u to v labelled with index i. If $i < 1$ or $i > n$ then there are no subterm edges from u to v labelled with i. If a subterm edge exists from u to v labelled with index i, we will call it (S, i, u, v). Also, every graph has an additional set of pairs called *Goal* associated with it, each pair consisting of two subterm edges.

Given a set EQ of equations and disequations, we create an initial SOUR graph in the following way from the subterms of EQ. For every element t of $Sub(EQ)$, there is a node in the graph labelled with t. We create a set of subterm edges $S(EQ)$ such that if $f(t_1, \cdots, t_n) \in Sub(EQ)$, then, for each t_i, there is an initial subterm edge $e \in S(EQ)$ from the node labelled with $f(t_1, \cdots, t_n)$ to the node labelled with t_i. This edge is labelled with index i. We also create a set $R(EQ)$ of rewrite edges such that if $s \approx t \in EQ$, then there is an initial rewrite edge from the node labelled with s to the node labelled with t. There is also an initial rewrite edge from the node labelled with t to the node labelled with s. We add both edges, but in inferences, we will use ordering constraints, which will be unsatisfiable for edges ordered the wrong way. In figures, we will sometimes draw unoriented rewrite edges, since they always appear in both directions. We say that the SOUR graph composed of edges of $S(EQ)$ and $R(EQ)$ is the *initial SOUR graph* of EQ. In Figure 1, we give the initial SOUR graph of $ffx \approx gfx$. This gives us the usual dag representation of a set of equations. One point that is important to understand is that when we define the semantics, only the top symbol of a term labelling a node will be used. The rest of the term was only used to aide us in defining the graph, so in future figures, only the top symbol will be written.

After the initial SOUR graph, is created, we must compile it into a schematization of the completed system by adding edges corresponding to inferences. Before we define how to perform the compilation, we define what it means for two nodes to *top unify*. Two nodes top unify if they represent the same term, ignoring constraints and renamings (which we describe below). Formally, if u and v are nodes, then u and v *top unify* if (i) u or v is labelled by a variable symbol, or (ii) u and v are labelled by the same n-ary function symbol and for all i, $1 \leq i \leq n$, there is an edge e_i from u to some u_i and an edge $e_i{}'$ from v to some v_i such that both e_i and $e_i{}'$ are labelled with index i, and u_i and v_i top unify. In order for terms represented by nodes to unify, the nodes must top unify.

We will define an inference procedure that detects certain patterns in the graph and adds a new edge based on each pattern. We first define a function RUR so that if $e_1 = (R, v_1, v_2)$ and $e_2 = (R, v_3, v_4)$ are edges such that (i) v_1 and v_3 top unify, (ii) v_1 is not labelled by a variable, and (iii) $e_3 = (R, v_2, v_4)$

$$\begin{tikzcd}
\mathbf{f}fx \arrow[r, dash, "R"'] \arrow[d, "S"'] & \mathbf{g}fx \arrow[dl, "S"] \\
\mathbf{f}x \arrow[d, "S"'] & \\
x &
\end{tikzcd}$$

Figure 1: Initial SOUR graph

$$\begin{tikzcd}
a \arrow[r, "R"'] \arrow[d, "R"'] & b \arrow[dl, dotted, "R"] \\
c &
\end{tikzcd}$$

Figure 2: RUR edge

is not an initial edge, then $e_3 = RUR(e_1, e_2)$. We say that e_3 is created by an *RUR transformation*. Creating edge e_3 corresponds to performing a critical pair between the equation represented by e_1 and the equation represented by e_2, at the root of the left hand sides. See Figure 2 to see a critical pair between $a \approx b$ and $a \approx c$. The new edge is represented by a dotted line. After adding the new edge, the graph represents the original equations plus the equation added by performing the critical pair.

Let SUR be a function so that if $e_1 = (S, i, v_1, v_2)$ and $e_2 = (R, v_3, v_4)$ are edges such that (i) v_2 and v_3 top unify, (ii) v_2 is not labelled by a variable, and (iii) $e_3 = (S, i, v_1, v_4)$ is not an initial edge, then $e_3 = SUR(e_1, e_2)$. Then e_3 is created by an *SUR transformation*. Creating edge e_3 corresponds to performing a inference from the equation represented by e_2 into the term represented by e_1, below the root. Figure 4 shows a critical pair from $a \approx c$ into $fa \approx b$. Figure 3 is a more complicated example. In Figure 3, the curved edge going from left to right represents the following inference:

$$\frac{ffx \approx gfx \qquad ffx \approx gfx}{fgfx\rho_1 \approx gfx\, [\![\, ffx\rho_1 \doteq fx \,\wedge\, ffx \succ gfx \,]\!]}$$

More precisely, what we have shown in the figure represents the *skeleton* part of the conclusion of this inference, i.e., everything except for the renaming and the constraint. Later, we will see how the renaming and constraint will be represented, by virtue of the label on the new edge. The conclusion can be simplified to $fgfx\rho_1 \approx gfx\, [\![\, fx\rho_1 \doteq x \,]\!]$

The line that goes from the edge labelled by g to itself represents the following inference:

$$\frac{ffx \approx gfx \qquad fgfx\rho_1 \approx gfx\, [\![\, fx\rho_1 \doteq x \,]\!]}{fggfx\rho_2 \approx gfx\, [\![\, ffx\rho_2 \doteq fx\rho_1 \,\wedge\, ffx \succ gfx \,\wedge\, fx\rho_1 \doteq x \,]\!]}$$

With the addition of these two edges, we have completely schematized the completion of $ffx \approx gfx$, which is an infinite set of equations.

We also define a function *Goal* so that if $e_1 = (S, 1, v, v_1)$ and $e_2 = (S, 2, v, v_2)$ are edges such that (i) v_1 and v_2 top unify, and (ii) v is labelled by a disequation $s \not\approx t$, then $Goal(e_1, e_2)$ is called a *goal transformation*. This corresponds to a possible solution of a goal. When we define the semantics, we will see that this represents a constraint that must be solved to solve the goal.

Now we are ready to show how to compile the graph. If EQ is a set of equations, we define $COMP(EQ)$ to be the graph $G = (V, E, Goal)$ where V is the set of nodes labelled by $Sub(EQ)$, and E and *Goal* are the smallest set of edges and goal transformations such that (i) $S(EQ) \cup R(EQ) \subseteq E$, (ii) if there is an SUR transformation $e_3 = SUR(e_1, e_2)$ for edges $e_1, e_2 \in E$, then $e_3 \in E$ and (e_1, e_2) is in the set of pairs of e_3, (ii) if there is an RUR transformation $e_3 = RUR(e_1, e_2)$ for edges $e_1, e_2 \in E$, then $e_3 \in E$ and (e_1, e_2) is in the set of pairs of e_3, (ii) if there is a *Goal* transformation

$$\begin{tikzcd}
f \arrow[r, "R"'] \arrow[r, bend left=60, "S"] \arrow[d, "S"'] & g \arrow[loop right, "S"] \arrow[dl, "S"] \\
f \arrow[d, "S"'] & \\
x &
\end{tikzcd}$$

Figure 3: SUR edges

$$\begin{tikzcd}
f \arrow[r, "R"'] \arrow[d, "S"'] \arrow[dr, dotted, "S"] & b \\
a \arrow[r, "R"'] & c
\end{tikzcd}$$

Figure 4: SUR edge

$e_3 = Goal(e_1, e_2)$ for edges $e_1, e_2 \in E$, then $e_3 \in E$ and (e_1, e_2) is in the set of pairs of e_3.

This completely defines the SOUR graph. What is left to do is to define how to interpret the SOUR graph to represent the schematization of a completed set of equations and the solutions to the goal. We also need to say how to use this interpretation to solve the goal constraints in a goal directed fashion. But first we must point out that $COMP(EQ)$ can be computed in polynomial time in the size of the initial set of equations. Clearly the initial graph can be created in polynomial time. There is a simple algorithm to compute $COMP(EQ)$ from the initial SOUR graph. The algorithm is the following. At any point, if there is an SUR transformation $e_3 = SUR(e_1, e_2)$ that has not performed, then we perform it. We perform it by checking if e_3 exists. If it does not exist, we add it with set of term pairs $\{(e_1, e_2)\}$. If it does exist, we add (e_1, e_2) to the set of term pairs. We do the same thing for RUR and Goal transformations. We keep performing these transformations until we have done them all. This algorithm will halt in polynomial time. First, notice that it is possible to calculate if a transition exists and find it in polynomial time. We only need to check all the pairs of subterm edges, pairs of subterm and rewrite edges, and pairs of rewrite edges. The number of such pairs is quadratic in the size of the graph. For each pair, we check the top unification problem. But top unification can, in fact be performed on the whole graph in time quadratic in the size of the graph, if answers are saved as they are calculated, and no calculation is performed twice. This is true, because we only have to top unify all pairs of edges. Therefore, each step of the algorithm is quadratic. Now we must argue that the algorithm halts and only takes polynomially many steps. Note that each step in the algorithm either adds a new edge to the graph or increases the number of edges associated with a pair. Since the number of nodes of the graph does not grow, we cannot add more than quadratically many edges in the initial size of the graph, which is no bigger than the initial size of the problem. Furthermore, each edge can have at most one pair in its set for each pair of edges in the graph. Therefore the size of each set is $O(n^4)$ in the size of the original problem. That shows that the algorithm has only polynomially many steps, therefore it is polynomial. This is a big polynomial, but in practice it will be much smaller.

4 SOUR Graph Semantics

We need a semantics which shows that the initial SOUR graph of a set of equations and disequations represents the initial set of equations. We also need to show that a compiled SOUR graph of a set of equations represents a schematization of the completed set of equations, and a compiled SOUR graph of a set of equations and disequations represents the schematization of the completed set plus a set of recursive syntactic unification and ordering constraints representing possible solutions of the goal.

We define the semantics of nodes, and subterm and rewrite edges, inductively, in terms of each other. Each node and edge has a set of functions associated with it, which can be applied to certain types of arguments. Let e be an edge (or node). Then F_e will be the set of functions associated with e. Each function in F_e will have the same type of inputs and output. If the functions in F_e take inputs of type $T_1, \cdots, T_n$, then we define $N_e = \{F(n_1, \cdots, n_p) \,|\, n_i \in T_i \text{ for all } i,\ 1 \leq i \leq p\}$. Functions will be written in typed lambda calculus notation. Therefore, each $n \in N_e$ is a typed lambda calculus term, and $Nf(n)$ is the normal form reached by reducing all the radices in n. Also, let T_r be the set of all fresh renamings.

If v is a node, then each member of N_v will be a constrained term. If e is a subterm edge, then each member of N_e will be a triple (φ, ρ, t), where φ is a constraint, ρ is a renaming, and t is a constrained term. We consider *Con*, *Ren* and *Term* to be functions applied to members of N_e which select the corresponding field. If e is a rewrite edge, then each member of N_e will be of the form $(\varphi, \rho_1, \rho_2, t_1, t_2)$, where φ is a constraint, ρ_1 and ρ_2 are renamings, and t_1 and t_2 are constrained terms. We consider Con, Ren_1, Ren_2, $Term_1$ and $Term_2$ to be functions applied to members of N_e which select the corresponding fields.

To understand the complicated functions we are about to describe, we note that for each node v, the normal forms of all members of N_v represent terms whose dag representations are rooted at the node. For an edge e, the normal forms of the members of N_e represent the the constraints, renamings and terms that are built up by inferences. For the case of subterm edges, these apply to the final node of the edge. For rewrite edges, terms and renamings apply to both nodes. There is one constraint to apply to both nodes.

Let v be a node labelled by a p-ary function symbol f and $i_1, \cdots, i_p$ be integers.[2] Then $F_v = \{\lambda n_1 : N_{e_1}, \cdots, n_p : N_{e_p}.Term(v, i_1, \cdots, i_p) \mid \text{ for all } j,\ e_j \text{ is an edge leading from } v \text{ to some } v_j \text{ with index } j\}$, where $Term(v, i_1, \cdots, i_p) = f(Term(n_{i_1})Ren(n_{i_1}), \cdots, Term(n_{i_p})Ren(n_{i_p}))\,[\![\,\bigcup_{1 \leq j \leq p} Con(n_{i_j})\,]\!]$.

For the rest of the definitions, let i, j, k be integers. Let $e = (S, l, u, v)$ be an initial subterm edge. Then $F_e = \{\lambda n : N_v.List(e, i, j, k)\}$, where $List(e, i, j, k) = (\top, id, n)$. Let $e = (R, u, v)$ be an initial rewrite edge. Then $F_e = \{\lambda n_1 : N_u, n_2 : N_v.List(e, i, j, k)\}$, where $List(e, i, j, k) = (\top, id, id, n_1, n_3)$.

Let $e = (S, l, u, v)$ be a non-initial subterm edge. Then $F_e = \{\lambda n_1 : N_{e_1}, n_2 : N_{e_2}, \rho_k : T_r.List(e, i, j, k) \mid (e_1, e_2) \text{ is a pair in the pair set of } e\}$, where $List(e, i, j, k) = (Con(e, i, j, k), Ren(e, i, j, k), Term(e, i, j, k))$, such that $Con(e, i, j, k) = Con(n_i) \wedge Con(n_j)\rho_k \wedge Term_1(n_j)Ren_1(n_j)\rho_k \doteq Term(n_i)Ren(n_i) \wedge Term_1(n_j)Ren_1(n_j) \succ Term_2(n_j)Ren_2(n_j)$, and $Ren(e, i, j, k) = Ren_2(n_j)\rho_k$, and $Term(e, i, j, k) = Term_2(n_j)$.[3]

Let $e = (R, u, v)$ be a non-initial rewrite edge. Then $F_e = \{\lambda n_1 : N_{e_1}, n_2 : N_{e_2}, \rho_k : T_r.List(e) \mid (e_1, e_2)$ is a pair in the pair set of $e\}$, where $List(e, i, j, k) = (Con(e, i, j, k), Ren_1(e, i, j, k), Ren_2(e, i, j, k),$ $Term_1(e, i, j, k), Term_2(e, i, j, k))$, such that $Con(e, i, j, k) = Con(n_i) \wedge Con(n_j)\rho_k \wedge$ $Term_1(n_j)Ren_1(n_j)\rho_k \doteq Term_1(n_i)Ren_1(n_i) \wedge Term_1(n_i)Ren_1(n_i) \succ Term_1(n_j)Ren_1(n_j) \wedge$ $Term_1(n_j)Ren_1(n_j) \succ Term_j(n_j)Ren_2(n_j)$, and $Ren_1(e, i, j, k) = Ren_2(n_i)$, and $Ren_2(e, i, j, k) = Ren_2(n_j)\rho_k$, and $Term_1(e, i, j, k) = Term_2(n_i)$, and $Term_2(e, i, j, k) = Term_2(n_j)$.

If $e = Goal$, then $F_e = \{\lambda n_1 : N_{e_1}, n_2 : N_{e_2}.List(e, i, j, k) \mid (e_1, e_2) \text{ is a pair in the pair set of } e\}$, where $List(e, i, j, k) = Con(e, i, j, k)$, such that $Con(e, i, j, k) = Con(n_i) \wedge Con(n_j) \wedge Term(n_i)Ren(n_i)$ $\doteq Term(n_j)Ren(e_j)$.

For a graph $G = (V, E, Goal)$, let E_r be the set of all rewrite edges in R. We define $Eqs(G)$, $Deqs(G)$ and $Sols(G)$ to represent all equations, disequations and goal solutions given by G, according to the semantics. Formally, $Eqs(G) = \{Nf(s) \approx Nf(t) \,|\, (\varphi, \rho_1, \rho_2, s, t) \in N_e \text{ for some } e \in E_r\}$. Also, $Deqs(G) = \{Nf(n) \,|\, n \in N_v \text{ for some } v \in V \text{ where } top(v) \text{ is } \not\approx\}$. Additionally, $Sols(G) = \{Nf(n) \,|\, n \in N_e \text{ and } e = Goal\}$. The following are straightforward consequences of the definitions.

Theorem 2 *Let EQ be a set of equations and disequations. Let G be the initial graph of EQ. Then $Eqs(G) \cup Deqs(G) = EQ$.*

[2]It does not not matter which integers we pick for these definitions. They are only necessary so we can perform renamings in the inference rules in the next section.

[3]For this definition and the following two definitions, please refer to the inference rules at the end of section 2. We are only reproducing those constraints.

The next theorem shows that the compilation process schematizes the completed system and finds all the solutions to the goals. It is proved by the relationship between SOUR derivations and Basic Paramodulation derivations (also see [10]).

Theorem 3 *Let EQ be a set of equations and disequations. Let $G = COMP(EQ)$. Then $Sols(G)$ is a complete set of solutions for EQ.*

5 Goal Directed Completion

Now we describe how to solve the *Goal* constraint in $COMP(EQ)$. For this procedure, we save a constraint denoted by φ, a set of variable–type pairs denoted by VT, and a set of term equalities denoted by TE. The constraint is built up by the algorithm. Each member of VT is of the form $n : T$ where n is a variable and T is the type of n. Elements of TE are of the form $Term_1(n) = t$ or $Term_2(n) = t$, where t is a constrained term we are building.

Initially, $\varphi = Con(n)$, $VT = \{n : N_{Goal}\}$, and $TE = \emptyset$. The algorithm expands φ. At each point of the algorithm, we can perform an inference rule called *Definition Expansion* or an inference rule called *Term Expansion*, which modify φ, VT and TE. First we don't-care nondeterministically select an inference rule to perform and a variable from VT to expand. Then we don't-know nondeterministically select an expansion to perform. After the inference we deterministically perform some simplification steps. This is analogous to the SLD-Resolution rule in Logic Programming, where we don't-care nondeterministically select a subgoal to solve and then don't-know nondeterministically select a rule to resolve against the subgoal. As in SLD Resolution, Goal-Directed Completion may be implemented with a breadth-first search or with iterative deepening. A depth first search is not complete, because we could follow a single path and ignore the one that solves the goal.

The *Definition Expansion* rule replaces a variable by its definition:

Definition Expansion

$$\frac{\varphi[n] \qquad VT \cup \{n : N_e\} \qquad TE[n]}{\varphi[List(e,i,j,k)] \qquad VT \cup \{n : N_e, n_i : N_{e_1}, n_j : N_{e_2}\} \qquad TE[List(e,i,j,k)]}$$

where (e_1, e_2) is in the pair set of e, and i, j, and k are fresh numbers.

This mean that we select a variable n appearing in VT and replace n everywhere it appears in the constraint we are solving with renamed versions of the edges that create n. For example, this implies that we replace a field of the form $Con(n)$ by a renamed copy of its meaning. In the term equations, we do the same, effectively replacing $Term(e)$ by a renamed copy of its meaning.

The *Term Expansion* rule expands a term:

Term Expansion

$$\frac{\varphi[Term_m(n)] \qquad VT \cup \{n : N_e\} \qquad TE[Term_m(n)]}{\varphi[Term(v_m, i_1, \cdots, i_p)] \qquad VT \cup \{n : N_e, n_{i_1} : N_{e_1}, \cdots, n_{i_p} : N_{e_p}\} \qquad TE[Term(v_m, i_1, \cdots, i_p)]}$$

where (i), $m \in \{1, 2\}$, (ii) e goes from v_1 to v_2, (iii) for all j, e_j is an edge from v_m to some v_j with index j, and (iv) $i_1, \cdots, i_p$ are fresh numbers.

This says that if there is a term $Term_m(n)$ that appears in the constraint, we can find this node in the graph and expand the term, replacing the new constraints and renamings by renamed copies.

After each inference rule, we also need to perform simplification steps to solve the equational and ordering constraints which appear. For lack of space, we do not present the simplifications here (see Figure 3 in [3]). These transformations convert constraints to a normal form. We can then perform the algorithms given in [3, 12] to solve the constraints.[4] If the constraints are unsatisfiable we fail. If the constraints are satisfiable, we leave them in normal form and continue the goal solving procedure. Each branch of this procedure halts when no more inferences can be performed.

We want to show that these inference rules give us all the solutions to our goal. Let Φ_{EQ} be the set of all constraints φ such that (φ, VT, TE) is in the closure of the above inference rules over $COMP(EQ)$, and no inferences can be applied to (φ, VT, TE). We have the completeness theorem:

Theorem 4 *Let EQ be a set of equations and disequations. Then* Φ_{EQ} *is a complete set of solutions for EQ.*

The theorem is proved by showing that for each (φ, VT, TE), the set of all Definition Expansion inferences on a given variable, preserves the set of solutions. This is also true for Term Expansion.

We have defined an algorithm which starts with the potential goal solutions and works its way backwards to the original equations. But the inference rules can be modified so that we can start at any point and build from there in finding goal solutions. For instance, we could simulate the Knuth-Bendix algorithm by starting with the original equations and working toward the goal. Perhaps the most interesting algorithm is one that starts with the initial goal equation, instead of the solutions to the goal. This algorithm performs inferences to build up the goal solution, and it also works its way back to the original equations. It is also truly goal directed in that it starts from the goal and works its way backwards toward the original equations.

6 Decidability Results and Future Work

A natural question to show the usefulness of these ideas is to show how SOUR graphs are used to give a decision procedure in cases where Knuth-Bendix completion does not because the completed system is infinite. For instance, consider the equation $ffx \approx gfx$ and goal $a \not\approx b$ given in the introduction. Knuth-Bendix runs forever on this example. Our algorithm compiles the system in polynomial time. After the compilation is finished, there are no goal constraints. Therefore, the Definition Expansion and Term Expansion rules cannot be performed. And the system stops immediately and says that the goal is not true. So this is an example where our algorithm is superior to the Knuth-Bendix algorithm.

It would be even more interesting if we could show that there are some equational theories such that the Knuth-Bendix algorithm never halts for any false goals, but our system halts for all goals. For instance, consider the equational system $ffx \approx gfx$. First note that $\{fg^nfx \approx gg^nfx \mid n \geq 0\}$ is an infinite reduced system representing the same equational theory as $ffx \approx gfx$. Therefore Knuth-Bendix will not halt with the equation $ffx \approx gfx$ and a false goal. We would like to show that it does halt in our system

We believe we have given convincing evidence to show that SOUR graphs and the algorithms in this paper are a powerful technique for proving theories decidable. We are currently examining this, and we will give some results at the Unification Workshop. The constraints in $COMP(EQ)$ provide solutions to the goal that can be represented as recurrence equations on terms. We need to develop techniques to solve such recurrence equations. Other ways of developing this is to use recently developed constraint automata techniques, or possibly to use techniques developed in logic programming, since the constraints we give are similar to constraints that would be created by compiling a set of Horn clauses in a similar way.

References

[1] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic Paramodulation. *Information and Computation Vol. 121, No. 2* (1995) pp. 172–192.

[4]We are assuming the ordering is the lexicographic path ordering

[2] J. Chabin, and P. Réty. Narrowing directed by a graph of terms. In *Proc. 4th Int. Conf. on Rewriting Techniques and Applications*, Lect. Notes in Computer Science, vol. 488, pp. 112–123, Springer-Verlag.

[3] H. Comon. Solving Symbolic Ordering Constraints. *International Journal of Foundations of Computer Science* **1** (1990) pp. 387–411.

[4] N. Dershowitz and J. -P. Jouannaud. Rewrite Systems. In J. van Leeuwen editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pp. 243–320. North-Holland, Amsterdam, 1990.

[5] N. Dershowitz and G. Sivakumar. Solving goals in equational languages. In *Proc. of the First Intl. Workshop on Conditional and Typed Rewriting Systems*, Lect. Notes in Comput. Sci., vol. 308, (1987) pp. 45–55.

[6] M. Hermann and R. Galbavý. Unification of Infinite Sets of Terms Schematized by Primal Grammars. To appear in *Theoretical Computer Science.*

[7] D. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, Ed. *Computational Problems in Abstract Algebras*, Oxford, Pergamon Press.

[8] D. Kozen. Complexity of Finitely Presented Algebras. PhD Thesis. Cornell University. 1977.

[9] C. Lynch. Paramodulation without Duplication. In *10th Intl IEEE Symposium on Logic in Computer Science*, (1995) pp. 167-177.

[10] C. Lynch, and P. Strogova. SOUR Graphs for Efficient Completion. CRIN Technical Report (1995).

[11] R. Nieuwenhuis and A. Rubio. Basic Superposition is Complete. In *Proc. European Symposium on Programming*, Lect. Notes in Computer Science, vol. 582, pp. 371–390, (1992) Springer-Verlag.

[12] R. Nieuwenhuis and A. Rubio. Theorem Proving with Ordering and Equality Constrained Clauses. *Journal of Symbolic Computation Vol. 19 No. 4* (1995) pp. 321–352.

[13] K. Rose. Xy-pic Reference Manual 3.0 edition. DIKU, University of Copenhagen. Universitetsparken 1, DK-2100 København Ø. June, 1995.

[14] W. Snyder. *A Proof Theory for General Unification.* Birkhauser Boston, Inc., Boston MA (1991).

Termination Proofs with Efficient *gpo* Ordering Constraint Solving

Thomas Genet and Isabelle Gnaedig

INRIA Lorraine & CRIN CNRS - BP 101
54602 Villers-lès-Nancy CEDEX FRANCE
Phone: (+33) 83-59-30-18 - Fax: (+33) 83-27-83-19
E-mail: {Thomas.Genet, Isabelle.Gnaedig}@loria.fr

To prove termination of a Term Rewrite System (TRS for short), the most commonly used method is to define a well-founded ordering between terms and show that each rewrite step is a strictly decreasing step. In general, the proof is made "a posteriori": orderings are tested at random or using human expertise, until an appropriate one is found.

Our goal here, is to reduce human expertise by working "a priori": starting from constraints on a generic ordering, we will help the user to build an appropriate specific instance of this ordering by using semi-automatic constraint solving methods.

The generic ordering, we start from, is the general path ordering (*gpo*), designed for expressing, in a single notion, a large set of well known orderings: syntactic orderings such as RPO or LPO, as well as semantic orderings like SPO or polynomial orderings. It is based on a lexicographic combination of *termination functions*. Particular orderings, such as those cited above, are obtained by instantiating termination functions with particular values.

Starting from inequalities on a general path ordering, we will reduce the set of possibilities for instantiation of termination functions by constraint solving, until we get a particular ordering, when it is possible.

The *gpo* ordering constraint algorithm is presented through a sound, complete and terminating set of deduction rules, limiting the explosion of computations thanks to a shared term data structure.

Conditional Rewrite Systems under Signature Extensions: Some Counterexamples

Bernhard Gramlich
INRIA Lorraine, 615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy, France
gramlich@loria.fr

Extended Abstract

1 Introduction

Here we are interested in the question whether certain weak and strong termination and confluence properties of conditional term rewriting systems (CTRSs) are preserved under signature extensions. We shall focus on the most interesting type of CTRSs, namely join systems, where equality in the conditions of a rule

$$l \to r \Longleftarrow s_1 = t_1, \ldots, s_n = t_n$$

is (recursively) interpreted as joinability, i.e., as $\downarrow$. In the presentation which is based on some results of [5] we assume familiarity with the basic no(ta)tions and terminology of (conditional) term rewriting (cf. e.g. [1], [6]).

The termination and confluence properties considered include the following:

- (strong) termination
- weak termination
- weak innermost termination, i.e., weak termination of the innermost reduction relation
- (strong) innermost termination, i.e., termination of the innermost reduction relation
- confluence
- local confluence
- joinability of all (conditional) critical pairs

From a systematic point of view signature extensions constitute a very special type of combining (disjoint) systems. Starting with the pioneering work of Toyama ([10]), the general question which properties P of rewrite systems are preserved under (disjoint or less restrictive) combinations (and vice versa) has received a lot of attention. For surveys of achieved results in this field we refer to [7], [8], [5]. For instance, for unconditional term rewriting systems (TRSs) it is well-known that confluence is *modular*, i.e., preserved under disjoint unions (and vice versa), but termination is not modular ([9]). However, various restricted termination properties have turned out to be modular for TRSs: weak termination and weak innermost termination (cf. e.g. [2]) as well as (strong) innermost termination ([4]). This implies in particular that these properties of TRSs are also preserved under signature extensions. That termination of TRSs which is not modular in general is at least preserved under signature extensions is almost trivial ([3]). All confluence properties mentioned above are modular for TRSs (for local confluence which is equivalent to joinability of all critical pairs this is again almost trivial), hence also preserved under signature extensions. However, for CTRSs the analysis is considerably more complicated. Middeldorp ([7]) has shown that the modularity of confluence carries over to CTRSs. However, he also showed that local confluence (and joinability of critical pairs) is not modular:

Example 1.1 *([7]) Consider the disjoint CTRSs $\mathcal{R}_1^{\mathcal{F}_1}$, $\mathcal{R}_2^{\mathcal{F}_2}$ given by*

$$\mathcal{R}_1 = \left\{ \begin{array}{lcl} f(x,y) \to x & \Longleftarrow & x \downarrow z, z \downarrow y \\ f(x,y) \to y & \Longleftarrow & x \downarrow z, z \downarrow y \end{array} \right\}$$

and

$$\mathcal{R}_2 = \left\{ \begin{array}{l} b \to a \\ b \to c \\ c \to b \\ c \to d \end{array} \right\}$$

over the signatures $\mathcal{F}_1 = \{f\}$ and $\mathcal{F}_2 = \{a,b,c,d\}$, respectively. It is easy to see that both $\mathcal{R}_1^{\mathcal{F}_1}$ and $\mathcal{R}_2^{\mathcal{F}_2}$ have joinable (conditional) critical pairs and are locally confluent. However, their disjoint union $\mathcal{R}^{\mathcal{F}} = \mathcal{R}_1^{\mathcal{F}_1} \oplus \mathcal{R}_2^{\mathcal{F}_2} = (\mathcal{R}_1 \uplus \mathcal{R}_2)^{\mathcal{F}_1 \uplus \mathcal{F}_2}$ is not locally confluent: The (new) instance

$$a \;{}_{\mathcal{R}_1 \oplus \mathcal{R}_2}\!\leftarrow f(a,d) \to_{\mathcal{R}_1 \oplus \mathcal{R}_2} d$$

of the critical peak between the first two rules of $\mathcal{R}_1$ is not joinable, since both a and d are irreducible.

Similarly, the properties of weak termination, weak and strong innermost termination are not modular for CTRSs in general.[1]

Example 1.2 *Consider the CTRSs $\mathcal{R}_1^{\mathcal{F}_1}$, $\mathcal{R}_2^{\mathcal{F}_2}$*

$$\left\{ a \to a \quad \Longleftarrow \quad x \downarrow b \wedge x \downarrow c \right\}$$

and

$$\left\{ \begin{array}{l} G(x,y) \to x \\ G(x,y) \to y \end{array} \right\}$$

[1]For the first two properties this was shown in [7].

over $\mathcal{F}_1 = \{a, b, c\}$ and $\mathcal{F}_2 = \{G, A\}$, respectively. Here, we have $a \rightarrow_{\mathcal{R}_1 \oplus \mathcal{R}_2} a$ by applying the $\mathcal{R}_1$-rule (x is substituted by $G(b, c)$), but neither $a \rightarrow_{\mathcal{R}_1} a$ nor $a \rightarrow_{\mathcal{R}_2} a$. Hence, a is an $\mathcal{R}_i$-normal form (for $i = 1, 2$) but not a normal form w.r.t. $\mathcal{R}_1 \oplus \mathcal{R}_2$. Obviously, both $\mathcal{R}_1$ and $\mathcal{R}_2$ are strongly (hence also weakly and innermost) terminating but their disjoint union is not. For instance, $a \rightarrow_{\mathcal{R}_1 \oplus \mathcal{R}_2} a \rightarrow_{\mathcal{R}_1 \oplus \mathcal{R}_2} a \rightarrow_{\mathcal{R}_1 \oplus \mathcal{R}_2} \ldots$ is an infinite innermost derivation, and a does not have a normal form w.r.t. $\mathcal{R}_1 \oplus \mathcal{R}_2$.

Below we give examples showing that the above non-modularity results for weak termination and confluence properties of CTRSs also hold for the very special case of signature extensions.

At first glance, these counterexamples may seem to be very surprising. In fact, they indicate once more the inherent complexity and intricacy of conditional rewriting.

2 Counterexamples for Signature Extensions

Example 2.1 (weak, weak innermost and strong innermost termination are not preserved under signature extensions)
Consider the CTRSs $\mathcal{R}_1^{\mathcal{F}_1}$, $\mathcal{R}_2^{\mathcal{F}_2}$ given by

$$\mathcal{R}_1 = \left\{ \begin{array}{rcl} f(g(x, y)) \rightarrow f(g(x, y)) & \Longleftarrow & x \downarrow z \wedge z \downarrow y \\ g(x, x) \rightarrow x & & \\ g(x, a) \rightarrow c & & \\ g(x, b) \rightarrow c & & \\ g(x, f(y)) \rightarrow c & & \\ g(x, g(y, z)) \rightarrow c & & \\ g(a, x) \rightarrow c & & \\ g(b, x) \rightarrow c & & \\ g(f(y), x) \rightarrow c & & \\ g(g(y, z), x) \rightarrow c & & \\ c \rightarrow a & & \\ c \rightarrow b & & \end{array} \right\}$$

over $\mathcal{F}_1 = \{f, g, a, b, c\}$ and $\mathcal{R}_2 = \emptyset$ over $\mathcal{F}_2 = \{G\}$ (with G unary). It is straightforward to verify that $\mathcal{R}_1^{\mathcal{F}_1}$ is (strongly) innermost terminating (hence also weakly innermost terminating and weakly terminating). The crucial point is that an arbitrary infinite $\mathcal{R}_1^{\mathcal{F}_1}$- derivation must contain rewrite steps using the first rule. But whenever this rule is applicable, the contracted redex cannot be innermost, since some proper subterm must then be reducible by the remaining rules which constitute a terminating CTRS. Nevertheless, in the extended system $\mathcal{R}^{\mathcal{F}} = \mathcal{R}_1{}^{\mathcal{F}_1 \uplus \{G\}}$ we get the cyclic (hence infinite) innermost $\mathcal{R}^{\mathcal{F}}$-derivation

$$f(g(G(a), G(b))) \rightarrow_{\mathcal{R}^{\mathcal{F}}} f(g(G(a), G(b))) \rightarrow_{\mathcal{R}^{\mathcal{F}}} f(g(G(a), G(b))) \rightarrow_{\mathcal{R}^{\mathcal{F}}} \ldots$$

by applying the first rule (instantiating the extra variable z by $G(c)$). Note moreover that there is no other way of reducing $f(g(G(a), G(b)))$ (all its proper *subterms are in normal form w.r.t. $\mathcal{R}^{\mathcal{F}}$, and the second rule is clearly not applicable). Hence, $\mathcal{R}^{\mathcal{F}} = \mathcal{R}_1{}^{\mathcal{F}_1 \uplus \{G\}}$ is not innermost terminating (and also neither weakly innermost terminating nor weakly terminating).*

Example 2.2 (local confluence and joinability of critical pairs are not preserved under signature extensions)
Consider the join CTRS $\mathcal{R}^{\mathcal{F}}$ with

$$\mathcal{R} = \left\{ \begin{array}{rcl} f(x,y,z) \to g(x) & \Longleftarrow & x \downarrow y, y \downarrow z \\ f(x,y,z) \to g(z) & \Longleftarrow & x \downarrow y, y \downarrow z \\ b \to a & & \\ b \to c & & \\ c \to b & & \\ c \to d & & \\ g(a) \to g(d) & & \\ f(a,x,y) \to f(d,x,y) & & \\ f(x,a,y) \to f(x,d,y) & & \\ f(x,y,a) \to f(x,y,d) & & \end{array} \right\}$$

and $\mathcal{F} = \{a, b, c, d, f, g\}$. It is not very difficult to show that $\mathcal{R}^{\mathcal{F}}$ has joinable critical pairs and is even locally confluent (but not confluent).

Now we add a fresh unary function symbol G, i.e. we consider $\mathcal{R}^{\mathcal{F}'}$ with $\mathcal{F}' = \mathcal{F} \uplus \{G\}$. Then joinability of critical pairs and hence local confluence is lost. To wit, consider for instance the term $f(G(a), G(b), G(d))$ which reduces to two distinct normal forms by one $\mathcal{R}^{\mathcal{F}'}$-step, respectively :

$$g(G(a)) \leftarrow f(G(a), G(b), G(d)) \rightarrow g(G(d))$$

Clearly both $g(G(a))$ and $g(G(d))$ are irreducible. This divergence corresponds to an instance of the critical pair between the first two rules, namely

$$\langle g(x) = g(z) \rangle \Longleftarrow x \downarrow y, y \downarrow z\,.$$

Over the old signature $\mathcal{F}$ every substitution σ which is feasible for this critical pair satisfies $\sigma(g(x)) \downarrow \sigma(g(z))$ whereas this is not the case for the mixed substitution $\tau = \{x \mapsto G(a), y \mapsto G(b), z \mapsto G(d)\}$. Hence the critical pair above is not joinable any more over the extended signature $\mathcal{F}'$.

Note that in the above example $\mathcal{R}^{\mathcal{F}}$ is obviously non-terminating. This is not essential in the following sense. We may replace the 'non-terminating part' of $\mathcal{R}^{\mathcal{F}}$

$$\left\{ \begin{array}{l} b \to a \\ b \to c \\ c \to b \\ c \to d \end{array} \right\}$$

which has joinable critical pairs, hence is locally confluent (it is an unconditional TRS!), by a terminating CTRS with joinable critical pairs which is not confluent, hence necessarily not locally confluent. To this end, we can take for instance the system

$$\left\{ \begin{array}{rcl} h(x) \to k(b) & \Longleftarrow & k(x) \downarrow h(b) \\ k(a) \to h(a) & & \\ a \to b & & \end{array} \right\}$$

which has the desired properties (in particular, it is not locally confluent: $h(b) \leftarrow h(a) \to k(b)$ but $h(b)$ and $k(b)$ are irreducible). Then the remaining construction of $\mathcal{R}^{\mathcal{F}}$ is adapted accordingly.

Example 2.3 (joinability of critical pairs is not even preserved for terminating CTRSs under signature extensions)
Consider the CTRS $\mathcal{R}^{\mathcal{F}}$ with

$$\mathcal{R} = \left\{ \begin{array}{rcl} f(x,y,z) \to g(x) & \Longleftarrow & x \downarrow y, y \downarrow z \\ f(x,y,z) \to g(z) & \Longleftarrow & x \downarrow y, y \downarrow z \\ h(x) \to k(b) & \Longleftarrow & k(x) \downarrow h(b) \\ k(a) \to h(a) & & \\ a \to b & & \\ g(h(b)) \to g(k(b)) & & \\ h(h(b)) \to h(k(b)) & & \\ k(h(b)) \to k(k(b)) & & \\ f(h(b),x,y) \to f(k(b),x,y) & & \\ f(x,h(b),y) \to f(x,k(b),y) & & \\ f(x,y,h(b)) \to f(x,y,k(b)) & & \end{array} \right\}$$

and $\mathcal{F} = \{a,b,f,g,h,k\}$. It is easy to verify that $\mathcal{R}^{\mathcal{F}}$ is terminating. With some effort one can show that $\mathcal{R}^{\mathcal{F}}$ has joinable critical pairs. But $\mathcal{R}^{\mathcal{F}}$ is not (locally) confluent since we have $h(b) \leftarrow h(a) \to k(b)$ with both $h(b)$ and $k(b)$ irreducible.

Now we add a fresh unary function symbol G, i.e. we consider $\mathcal{R}^{\mathcal{F}'}$ with $\mathcal{F}' = \mathcal{F} \uplus \{G\}$. The new system $\mathcal{R}^{\mathcal{F}'}$ is still terminating. But joinability of critical pairs is lost. Consider the critical pair between the first two rules of $\mathcal{R}^{\mathcal{F}'}$,

$$\langle g(x) = g(z) \rangle \Longleftarrow x \downarrow y, y \downarrow z\,,$$

and the $\mathcal{F}'$-substitution $\tau = \{x \mapsto G(h(b)), y \mapsto G(h(a)), z \mapsto G(k(b))\}$. The corresponding instance of the critical peak is

$$g(G(h(b))) \leftarrow f(G(h(b)), G(h(a)), G(k(b))) \to g(G(k(b)))\,,$$

due to $G(h(b)) \downarrow G(h(a)) \downarrow G(k(b))$, but $g(G(h(b)))$ and $g(G(k(b)))$ are not joinable since they are both irreducible (in $\mathcal{R}^{\mathcal{F}'}$).

In the talk, these counterexamples and some related positive results will be presented and discussed in more detail.

References

[1] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Formal models and semantics, Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier - The MIT Press, 1990.

[2] Klaus Drosten. *Termersetzungssysteme*. Informatik-Fachberichte 210. Springer-Verlag, 1989. In German.

[3] Bernhard Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 5:131–158, 1994.

[4] Bernhard Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24:3–23, 1995. Special issue on term rewriting systems, ed. D. A. Plaisted.

[5] Bernhard Gramlich. *Termination and Confluence Properties of Structured Rewrite Systems*. PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, January 1996.

[6] Jan Willem Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 2–117. Clarendon Press, Oxford, 1992.

[7] Aart Middeldorp. *Modular Properties of Term Rewriting Systems*. PhD thesis, Free University, Amsterdam, 1990.

[8] Enno Ohlebusch. *Modular Properties of Composable Term Rewriting Systems*. PhD thesis, Universität Bielefeld, 1994. Report 94-01.

[9] Yoshihito Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25:141–143, 1987.

[10] Yoshihito Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1):128–143, 1987.

The First-Order Theory of One-Step Rewriting by a Linear Term Rewriting System is Undecidable (Extended Abstract)

Ralf Treinen[*]
Laboratoire de Recherche en Informatique (LRI), Bât. 490
Université de Paris-Sud, F-91405 Orsay cedex, France
email: `treinen@lri.fr`, web: `http://www.lri.fr/~treinen`

May 24, 1996

Abstract

The theory of one-step rewriting for a given rewrite system R and signature Σ is the first-order theory of the following structure: Its universe consists of all Σ-ground terms, and its only predicate is the relation "x rewrites to y in one step by R". The structure contains no function symbols and no equality. We show that there is no algorithm deciding the $\exists^*\forall^*$-fragment of this theory for an arbitrary linear and non-erasing term rewriting system.

1 Introduction

The problem of decidability of the first-order theory of one-step rewriting was posed in [CCD93]. It has been mentioned in the list of open problems in rewriting in 1993 [DJK93] and in 1995 [DJK95].

In [Tre96] we have proven the undecidability of the theory of one-step rewriting, leaving open the special case of linear rewriting systems. Here we prove undecidability even for the class of linear (no multiple variable occurrences in the left-hand, resp. right-hand side of a rule) and non-erasing (for every rule, the left- and right-hand side have the same set of variables) term rewriting systems. The proof presented here is in fact simpler than the one of [Tre96]. There, we used a more complicated definition of P-terms (see Definition 2) which allowed us to separate the non-shallow rules from the non-linear ones.

An alternative proof of undecidability of one-step rewriting has been given by Sergei Vorobyov [Vor95]. He defines *one particular* (non-linear) rewrite system R and shows the undecidability of the full first-order theory of one-step rewriting by R. He considers, however, the structure where all the function symbols of the signature are available in the language (this is not the case with

[*]Supported by the *Human Capital and Mobility Programme* of the European Union, under the contracts *SOL* (CHRX-CT92-0053) and *CONSOLE* (CHRX-CT94-0495).

our result), and he does not attempt to characterize a "simple" undecidable fragment of the theory. The rewrite system of [Vor95] uses in fact a "swapping of variables" to test certain equalities, similar to the system presented here, but it still contains one non-linear rule.

The reader is referred to [Tre96] for the motivation and history of this problem.

2 Preliminaries

We write a signature as a set of function symbols, where we specify (following the Prolog tradition) the arity of the function symbols after a "/"-sign. The set of terms build over a signature Σ and set X of variables is denoted as $T(\Sigma, X)$, we write $T(\Sigma) = T(\Sigma, \emptyset)$ for the set of Σ-*ground terms*.

We consider first-order predicate logic *without* equality. The $\exists^*\forall^*$-*fragment* of a theory T is the subset of T of all sentences having a prenex normal form of the form

$$\exists x_1, \ldots, x_n \, \forall y_1, \ldots, y_m \, Q$$

where Q contains no quantifier.

We denote concatenation of words by juxtaposition. An instance of the *Post Correspondence Problem (PCP)* is a finite set of pairs of non-empty binary words $\{(p_i, q_i) \mid 1 \leq i \leq n, p_i, q_i \in \{a, b\}^+\}$. A solution of P is a finite non-empty sequence $(i_1, \ldots, i_m) \in \{1, \ldots, n\}^+$ such that

$$p_{i_1} \cdots p_{i_m} = q_{i_1} \cdots q_{i_m}$$

It is undecidable whether an instance of the PCP has a solution [Pos46].

3 One-Step Rewriting

Definition 1 *Let Σ be a signature and R be a Σ-rewrite system. The structure $\mathcal{A}_{\Sigma,R}$ is defined as follows: The language of $\mathcal{A}_{\Sigma,R}$ contains no constants or function symbols, and its only predicate symbol is the binary predicate symbol $\to$. The universe of $\mathcal{A}_{\Sigma,R}$ is the set $T(\Sigma)$, and $t \to s$ holds in $\mathcal{A}_{\Sigma,R}$ iff t rewrites to s in one rewriting step of R.*

Theorem 1 *There is no algorithm which decides for any signature Σ and any non-erasing and linear Σ-rewrite system R the $\exists^*\forall^*$-fragment of the theory of $\mathcal{A}_{\Sigma,R}$.*

We show how to reduce the solvability of an instance of the Post-Correspondence Problem (PCP) to the validity of some $\exists^*\forall^*$-sentence in $\mathcal{A}_{\Sigma,R}$ for some signature Σ and non-erasing and linear rewrite system R. All constructions and proofs are parameterized by the given instance of the PCP. For the sake of convenience, we now fix this instance for the rest of the paper to be

$$P = \{(p_i, q_i) \mid 1 \leq i \leq n\}$$

Definition 2 *The signature Σ_P is*

$$\{\epsilon/0, a/1, b/1, g/3, k/5, eq/3\}$$

A Σ_P-ground term t is called a P-term if it does not contain an occurrence of k or eq.

Words from $\{a,b\}^*$ can easily be encoded in $T(\Sigma)$. First we define an application of a word from $\{a,b\}$ to an arbitrary term $t \in T(\Sigma)$ inductively by

$$\begin{aligned} \epsilon(t) &= t \\ wa(t) &= a(w(t)) \\ wb(t) &= b(w(t)) \end{aligned}$$

Note that, for the case of $a(t)$ and $b(t)$, this coincides with the definition of the operations in $\mathcal{A}_{\Sigma,R}$. A word $w \in \{a,b\}^*$ is now represented by the term $w(\epsilon)$. Note that

- the empty word is represented by the term ϵ,
- the encoding is injective, that is equality of words translates to equality of their respective representations,
- and $w(t)$ represents the word vw iff t represents the word v.

Definition 3 *The rewrite system R_P consists of the following rules:*

$$\begin{aligned} k(x_1,x_2,x_3,x_4,x_5) &\to k(x_1,x_2,x_3,x_4,x_5) && (1) \\ eq(x_1,x_2,x_3) &\to eq(x_1,x_2,x_3) && (2) \\ g(\epsilon,\epsilon,g(x,y,z)) &\to g(x,y,z) && (3) \\ g(x_1,y_1,g(x_2,y_2,z)) &\to k(x_1,y_1,x_2,y_2,z) && (4) \\ \{k(x_1,y_1,p_i(x_2),q_i(y_2),z) &\to g(x_2,y_2,g(p_i(x_1),q_i(y_1),z)) \mid 1 \le i \le n\} && (5) \\ \{g(x,y,h(\bar{z})) &\to eq(x,y,h(\bar{z})) \mid h \in \{\epsilon,a,b\}\} && (6) \\ eq(x,y,h(\bar{z})) &\to g(y,x,h(\bar{z})) && (7) \end{aligned}$$

We use the following formulae

$$\begin{aligned} \Phi(x) &:= \neg x \to x \\ \Psi(x) &:= \Phi(x) \wedge \exists x' \, (x \to x' \wedge \Phi(x')) \end{aligned}$$

Proposition 1 $\mathcal{A}_{\Sigma_P,R_P}, \alpha \models \Phi(x)$ *iff* $\alpha(x)$ *is a P-term.*
$\mathcal{A}_{\Sigma_P,R_P}, \alpha \models \Psi(x)$ *iff* $\alpha(x)$ *is a P-term containing a subterm of the form $g(\epsilon,\epsilon,g(x,y,z))$.*

Proof: The first claim holds since (1) and (2) are the only rules that can rewrite a term to itself. The second claim holds since (3) is the only rule that can rewrite a P-term to a P-term. □

Lemma 1 *P is solvable iff*

$$\mathcal{A}_{\Sigma_P,R_P} \models \exists x \left(\underbrace{\Psi(x) \land \forall y \left((x \to y \land \neg\Phi(y)) \Rightarrow y \to x\right)}_{(8)} \right)$$

Proof: Let $(i_1, \ldots, i_m)$ be a solution of P. Consider the term

$$t = g(\epsilon, \epsilon, g(r_1, s_1, \ldots, g(r_m, s_m, \epsilon) \ldots))$$

where

$$\begin{array}{rcl} r_k & = & p_{i_k}(p_{i_{k-1}}(\cdots(p_{i_1}(\epsilon))\cdots)) \\ s_k & = & q_{i_k}(q_{i_{k-1}}(\cdots(q_{i_1}(\epsilon))\cdots)) \end{array}$$

for $1 \leq k \leq m$. It is easy to see that the formula 8 is satisfied when taking t for the value of x. Note that (4) and (6) are the only rules that can rewrite a P-term to a non-P-term.

On the other hand, let the formula (8) be satisfied by the the term t. We can show by induction on s that every subterm s of t which starts with g is of the form

$$t = g(r_1, s_1, \ldots, g(r_m, s_m, u) \ldots))$$

where u is either ϵ or starts with a or b, and where there is a sequence $(i_1, \ldots, i_m) \in \{1, \ldots, n\}^*$ such that for all $1 < j \leq m$

$$\begin{array}{rcl} r_j & = & p_{i_j}(r_{j-1}) \\ s_j & = & q_{i_j}(s_{j-1})) \\ r_m & = & s_m \end{array}$$

Note that, since t is a P-term, any subterm s of t must contain only the symbols g, ϵ, a, b. Hence, if s is a subterm of t starting with g, then either rule (4) or (6) rewrite s at the root to some non-P-term s', which can only be rewritten back to s by the rules (5), resp. (7).

The claim follows since t contains, by Proposition 1, a subterm of the form $g(\epsilon, \epsilon, g(\cdot, \cdot, \cdot))$. $\square$

4 Conclusions

There are two important special classes of rewrite systems where decidability of the first-order theory of one-step rewriting is not yet known:

1. orthogonal rewrite systems.

2. (left)-shallow rewrite systems.

Furthermore, decidability of the $\exists^*$-fragment as well as of the positive theory of one-step rewriting are still open.

References

[CCD93] Anne-Cécile Caron, Jean-Luc Coquide, and Max Dauchet. Encompassment properties and automata with constraints. In Kirchner [Kir93], pages 328–342.

[DJK93] Nachum Dershowitz, Jean-Pierre Jouannaud, and Jan Willem Klop. More problems in rewriting. In Kirchner [Kir93], pages 468–487.

[DJK95] Nachum Dershowitz, Jean-Pierre Jouannaud, and Jan Willem Klop. Problems in rewriting III. In Jieh Hsiang, editor, *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, vol. 914, pages 457–471. Springer-Verlag, April 1995.

[Kir93] Claude Kirchner, editor. *Rewriting Techniques and Applications, 5th International Conference, RTA-93*, Lecture Notes in Computer Science, vol. 690, Montreal, Canada, 1993. Springer Verlag.

[Pos46] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the AMS*, 52:264–268, 1946.

[Tre96] Ralf Treinen. The first-order theory of one-step rewriting is undecidable. In Harald Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Rutgers University, NJ, USA, July 1996. Springer-Verlag. To appear.

[Vor95] Sergei Vorobyov. Elementary theory of one-step rewriting is undecidable (note). Draft, 1995.

Syntactic Unification Problems under Constrained Substitutions

Yuichi Kaji Kazuhiro Takada Tadao Kasami[1]

1 Introduction

We sometimes want to accomplish our purpose by using limited operations and limited materials only. To describe such cases in a natural way, a unification problem *under constrained substitutions* has been proposed[3]. Indeed, by using the unification problem under constrained substitutions, we can describe a verification problem of cryptographic protocols (in which users' operations are very restricted) as a natural extension of a unification problem (see Example 2.3).

In the unification problem under constrained substitutions, a finite set A of terms is given in addition to goal terms. The essential point of the problem is that only those terms that belong to A can replace for variables. For example, if $A = \{f(f(x))\}$, then solved forms of the problem must be of the form $\{x_i = f^{2n}(y_i)\}$. From the technical view point, there is strong relationship between the unification problem under constrained substitutions and order-sorted unification problems[2, 5, 6]. Indeed, some results for the latter problem can be easily translated into the former, and vice versa.

In this paper, the decidability and computational complexity of *syntactic* unification problems under constrained substitutions are discussed. It is shown that the problem is undecidable in general, and decidable if all terms in A are linear or ground. Furthermore, under the assumption that all terms in A are linear or ground, it is shown that the problem is solvable in deterministic polynomial time if goal terms are linear and there is no variable that occurs in both goal terms, and is $\mathcal{NP}$-hard, otherwise. These are theoretically interesting results, and give foundations of E-unification under constrained substitutions.

2 Preliminary

Throughout this paper, F denotes the set of function symbols and X denotes the set of variables. For a finite set A of terms, let $\mathcal{T}_A[X]$ be the minimal set of terms satisfying the following conditions.

- $X \subseteq \mathcal{T}_A[X]$.

[1]Graduate School of Information Science, Nara Institute of Science and Technology, Takayama 8916-5, Ikoma, Nara 630-01, Japan, E-mail: {`kaji,kazuh-t,kasami`}`@is.aist-nara.ac.jp`

- for a term $t \in A$ such that $\mathit{Var}(t) = \{x_1, \ldots, x_n\}$ and terms $t_1, \ldots, t_n \in \mathcal{T}_A[X]$, $t\sigma \in \mathcal{T}_A[X]$ where $\sigma = \{x_i \mapsto t_i \mid 1 \leq i \leq n\}$.

$\mathcal{T}_A$ denotes the set of ground terms in $\mathcal{T}_A[X]$. A substitution is said to be an *A-constrained substitution* if its co-domain is a subset of $\mathcal{T}_A[X]$. Terms s and t are *unifiable under A-constrained substitutions* if there is an A-constrained substitution σ such that $s\sigma = t\sigma$. For given a finite set A of terms and two goal terms g_1 and g_2, a *syntactic unification problem under constrained substitutions* (*SUPCS* for short) is a problem to decide whether g_1 and g_2 are unifiable under A-constrained substitutions.

Example 2.1: Let $g_1 = f(h(h(x_1)), x_2)$, $g_2 = f(x_3, x_3)$ and $A = \{h(h(x))\}$. In this case, the set $\mathcal{T}_A[X]$ is defined to be $\mathcal{T}_A[X] = \{h^{2n}(x) \mid n \geq 0, x \in X\}$. An A-constrained substitution $\sigma = \{x_2 \mapsto h(h(x_1)), x_3 \mapsto h(h(x_1))\}$ unifies g_1 and g_2. □

Example 2.2: Let $g_1 = f(h(x_1), x_2)$, $g_2 = f(x_3, x_3)$ and $A = \{h(h(x))\}$. Remark that the goal term g_1 is slightly different from the previous example. Though terms g_1 and g_2 are unifiable in the usual unification problem, they are not unifiable under A-constrained substitutions. □

Example 2.3: This is an example of a unification problem under constrained substitutions *modulo a rewriting rule*. Unification modulo rewriting rules is out of topic of this paper, though, this example figures out the motivation of constrained substitutions.

Consider the following cryptographic protocol of which purpose is to transmit messages secretly. In the protocol, there is a supervisor called a server who knows secret keys of all users. Assume that a user A wants to send a message m to B secretly, but the key of B is not known to A. To transmit the message m, the communication is carried out as follows.

1. A first chooses a random number r, and sends the server A, B and $E(K(A), r)$ where $K(x)$ and $E(x, y)$ denote the key of the user x and the ciphertext of y encrypted with x as a key, respectively.

2. The supervised server decrypts $E(K(A), r)$ with $K(A)$, and encrypts the result with $K(B)$ as a key. The resulting ciphertext $E(K(B), r)$ is sent back to A.

3. A sends B two ciphertexts $E(K(B), r)$ and $E(r, m)$.

4. B retrieves r by decrypting the first ciphertext, and retrieves m by using r as a key.

We want to verify whether an intruder, say C, can reveal m or not. The intruder C knows that if a message encrypted with a key is decrypted with the same key, then the original message yields, that is, $D(x, E(x, y)) \to y$. C also knows public information and information which was sent through insecure communication channel, that is, information represented by a set of terms $\mathit{Inf} = \{A, B, C, K(C), E(K(A), r), E(K(B), r), E(r, m)\}$ (the last three terms correspond to the information that C can find by wiretapping). Furthermore, C can execute some operations which are represented by $Opr = \{E(x,y), D(x,y), E(K(x), D(K(y), z))\}$ (the last term corresponds to what the server does). Remark that C cannot obtain other users' key and hence $K(x) \notin Opr$.

Let $A = Opr \cup Inf$, then terms in $\mathcal{T}_A$ correspond to the information which C can obtain. Observe that

$$D(D(K(C), E(K(C), D(K(A), E(K(A), r)))), E(r, m)) \in \mathcal{T}_A$$

and this term is rewritable to m. This means that C can reveal the secret message m by using limited operations and limited materials (information) only. In other words, x and m are unifiable with respect to the rewriting rule $D(x, E(x,y)) \to y$ under A-constrained substitutions if and only if the given cryptographic protocol is insecure. □

3 Order-Sorted Unification and the Decidability of SUPCS

From the viewpoint of technical aspect, there is strong relationship between SUPCS and *order-sorted unification problem with term declarations*[5] (*TD-unification problem* or Schemidt-Schauß style order-sorted unification problem). By associating each term in A of SUPCS with a term declaration in an order-sorted signature, SUPCS can be regarded as a special case of TD-unification problems with at most two sorts (the arrows(*) in Figure 1 denote this relation). It is known that TD-unification problems are undecidable in general (see Theorem 6.1 of [5]). At a glance, this result does not contribute to SUPCS since TD-unification is more general framework than SUPCS. However, careful observation of the proof of Theorem 6.1 of [5] tells us that the proof is applicable to SUPCS. Thus we have the following theorem.

Theorem 3.1: SUPCS is undecidable in general.
Proof. See the proof of Theorem 6.1 of [5]. □

If all terms in A of SUPCS are linear, then SUPCS is regarded as an *order-sorted unification problem with linear term declarations*[5] (*LTD-unification problem* for short). To the authors' knowledge, the decidability of LTD-unification

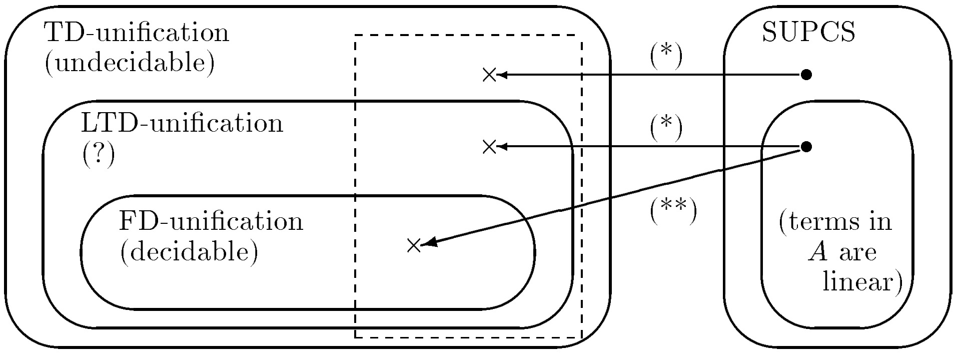


Figure 1: Order-sorted unification problems and SUPCS.

problems is not known yet. As a subclass of LTD-unification problems, *order-sorted unification problems with function declarations* (*FD-unification problem*) are considered in [5]. FD-unification problem coincides with Smolka style order-sorted unification problem[6], and known to be decidable. In the following, we illustrate that SUPCS is polynomial-time reducible to an FD-unification problem (the arrow(**) in Figure 1 corresponds to this reduction).

Example 3.1: Let $A = \{g(g(a)), f(g(x), y)\}$. We define an order-sorted signature $(S, F, \geq_S)$ as follows. First, the set of sorts are defined to be $S = \{\underline{g(g(a))}, \underline{g(a)}, \underline{a}, \underline{f(g(\Omega), \Omega)}, \underline{g(\Omega)}, \underline{\Omega}, \text{OTHER}\}$. Remark that a sort is introduced for each subterm of a term in A. Function symbols are defined to be $F = \{f, g, a\}$ where

$$\begin{array}{lrcl}
g: & \underline{\Omega} \to \underline{g(\Omega)} & f: & \underline{g(\Omega)} \times \underline{\Omega} \to \underline{f(g(\Omega), \Omega)} \\
& \underline{a} \to \underline{g(a)} & & \text{OTHER} \times \text{OTHER} \to \text{OTHER} \\
& \underline{g(a)} \to \underline{g(g(a))} & a: & \to \underline{a} \\
& \text{OTHER} \to \text{OTHER} & & \to \text{OTHER},
\end{array}$$

and sort ordering is defined to be $\underline{g(g(a))} <_S \underline{\Omega}$, $\underline{f(g(\Omega), \Omega)} <_S \underline{\Omega}$ (remark that both of $\underline{g(g(a))}$ and $\underline{f(g(\Omega), \Omega)}$ correspond to terms in A) and $\underline{s} <_S \text{OTHER}$ for every $\underline{s} \in S$. Observe that if a term t is of sort $\underline{\Omega}$, then t is either of sort $\underline{g(g(a))}$ or $\underline{f(g(\Omega), \Omega)}$. In the former case, t must be $g(g(a))$, and in the latter case, $t = f(g(x), y)\sigma$ where σ is an A-constrained substitution. Hence the set of terms of sort $\underline{\Omega}$ equals to the set $\mathcal{T}_A[X]$.

Let g_1 and g_2 be given goal terms of SUPCS. It is easily understood that g_1 and g_2 are unifiable under A-constrained substitutions if and only if g_1 and g_2 are unifiable with respect to this order-sorted signature where each variable in g_1 and g_2 is considered to be of sort $\underline{\Omega}$. □

It is not difficult to verify that this is a (polynomial-time) reduction from SUPCS to FD-unification problems (Smolka style order-sorted unification problems). Thus the following theorem holds.

Theorem 3.2: If all terms in A are linear, then SUPCS is decidable. □

4 The Computational Complexity of SUPCS

4.1 Classification of Instances of SUPCS

In the following, the computational complexity of SUPCS under some restricted situations is discussed. Consider the following three kinds of conditions on instances of SUPCS.

- a condition on terms in A;

  (1) terms in A are ground.

  (2) terms in A are ground or linear.

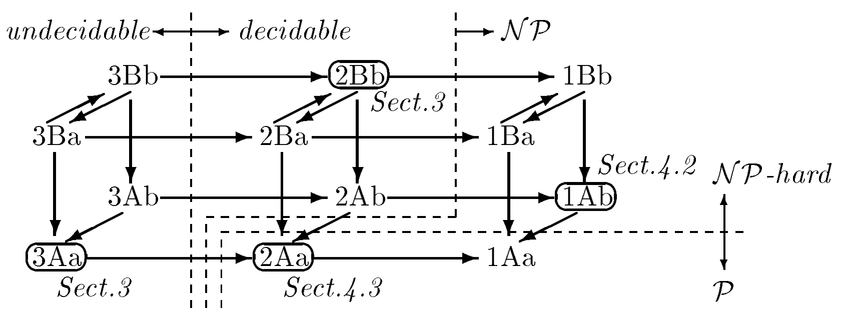


Figure 2: Relation of 12 classes of instances of SUPCS.

(3) there is no restriction on terms in A.

- a condition on linearity of g_1 and g_2;

  (A) both of g_1 and g_2 are linear.

  (B) there is no restriction on linearity of goal terms.

- a condition on variables occurring in g_1 and g_2;

  (a) $Var(g_1) \cap Var(g_2) = \emptyset$.

  (b) there is no restriction on variables occurring in goal terms.

According to these conditions, we can define 12 classes of instances of SUPCS. Figure 2 illustrates relations of these classes of instances of SUPCS with respect to reducibility of the problem. In the figure, "1Aa" stands for the class of instances of SUPCS that satisfy the conditions (1), (A) and (a) above. An arc $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ in the figure means that SUPCS for the class $\mathcal{C}_1$ trivially includes SUPCS for the class $\mathcal{C}_2$, or that SUPCS for the class $\mathcal{C}_2$ is polynomial-time reducible to SUPCS for the class $\mathcal{C}_1$.

Lemma 4.1: SUPCS for the class 3Bb (resp. 2Bb, 1Bb) is polynomial-time reducible to SUPCS for the class 3Ba (resp. 2Ba, 1Ba).

Sketch of Proof. For goal terms g_1 and g_2 such that $Var(g_1) \cap Var(g_2) \neq \emptyset$, let g_1' and g_2' be terms that are obtained from g_1 and g_2 by renaming variables so that $(Var(g_1) \cup Var(g_2)) \cap (Var(g_1') \cup Var(g_2')) = \emptyset$. Introduce a new symbol *dummy* with arity three, and define $g_1'' = dummy(g_1, g_1, g_2)$ and $g_2'' = dummy(g_1', g_2', g_2')$. It is easily verified that g_1 and g_2 are unifiable under constrained substitutions if and only if g_1'' and g_2'' are unifiable under constrained substitutions. $\square$

In the previous section 3, we used Theorem 6.1 of [5] to show that SUPCS is undecidable in general. Indeed, the goal terms used in the proof of Theorem 6.1 of [5] satisfy the conditions (A) and (a) above. Therefore, we can conclude that SUPCS for the class 3Aa is undecidable, and so is SUPCS for the classes 3Ab, 3Ba and 3Bb. On the other hand, the decidability result introduced in the previous section do not depend on goal terms. Thus SUPCS is decidable for the class 2Bb, and for the classes which are not more difficult than 2Bb.

4.2 Complex Goal Terms Make the Problem Intractable

It is shown that 3SAT problem is polynomial-time reducible to SUPCS for the class 1Ab. By tracing the arrows in the Figure 2 from 1Ab in a reverse way, we can conclude that SUPCS for the classes {1,2}Ab (i.e. classes 1Ab and 2Ab) and {1,2}B{a,b} are $\mathcal{NP}$-hard. Furthermore, SUPCS for the classes 1Ab and 1B{a,b} are $\mathcal{NP}$-complete since SUPCS obviously belongs to $\mathcal{NP}$ if A contains ground terms only.

Since the reduction from 3SAT to SUPCS is easy, we present a simple example of polynomial-time reduction from 3SAT to SUPCS for the class 1Ab, instead of general reduction algorithm and a proof of its correctness. See [7] for details.

Example 4.1: Let $E = (v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2 \vee \neg v_2)$ be a given Boolean expression. The set A of terms and goal terms g_1 and g_2 are constructed as follows.

$$\begin{array}{rcl}
g_1 & = & \mathrm{ROOT}(\mathrm{OR}(x_{11}, x_{12}, N(x_{13})), \mathrm{OR}(N(x_{21}), x_{22}, N(x_{23})), z_1, z_2, z_3) \\
g_2 & = & \mathrm{ROOT}(y_1, y_2, V_1(x_{11}, x_{21}), V_2(x_{12}, x_{22}, x_{23}), V_3(x_{13})) \\
A & = & \{T, N(T)\} \\
 & & \cup\ \{V_1(T,T), \quad V_1(N(T), N(T))\} \\
 & & \cup\ \{V_2(T,T,T), \quad V_2(N(T), N(T), N(T))\} \\
 & & \cup\ \{V_3(T), \quad V_3(N(T))\} \\
 & & \cup\ \{\mathrm{OR}(t_1, t_2, t_3) \mid t_1, t_2, t_3 \in \{T, N(T), N(N(T))\} \setminus \\
 & & \quad \{\mathrm{OR}(N(T), N(T), N(T))\}
\end{array}$$

It is easily understood that E is satisfiable if and only if g_1 and g_2 are unifiable under A-constrained substitutions. □

As we have seen in Section 3, SUPCS can be polynomial-time reducible to FD-unification problems (Smolka style order-sorted unification problem) when all terms in A are linear or ground. Therefore we have the following corollary.

Corollary 4.2: FD-unification problem (Smolka style order-sorted unification problem) is $\mathcal{NP}$-hard in general. □

4.3 Simple Goal Terms Make the Problem Tractable

SUPCS for the class 2Aa is solvable in deterministic polynomial time (i.e. belongs to $\mathcal{P}$). The overview of the procedure is as follows: First, find the most general unifier $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ of g_1 and g_2, ignoring the set A. Then construct tree automata[4] M_i that accepts $\{t_i\sigma \mid \sigma$ is a ground A-constrained substitution$\}$ for $1 \le i \le n$ and a tree automaton M_0 that accepts $\mathcal{T}_A$. If the intersection of terms accepted by M_0 and M_i is not empty for all $1 \le i \le n$, then g_1 and g_2 are unifiable under A-constrained substitutions. It is easily verified that the construction of tree automata is possible in deterministic polynomial time and thus the size of each tree automaton is bounded by a polynomial to the size of an input. On the other hand, it is known that both of calculating the intersection of regular tree languages and checking the emptiness of a regular tree language are possible in deterministic polynomial-time to the size of corresponding tree automata[4]. Therefore, the above procedure ends in polynomial time.

If goal terms violate the conditions (A) or (a), then we cannot use this procedure since it can happen that t_i in the mgu is not linear, and we cannot construct the tree automaton M_i.

5 Conclusion

Decidability and computational complexity of SUPCS are discussed. We note that the decidability result for the class 2Bb and the deterministic polynomial-time procedure for the class 2Aa greatly owe to favorable properties of tree automata (for technical discussions, see [2] and [7]). By replacing tree automata with "enhanced" tree automata such as in [1], we may relax conditions on linearity of terms.

References

[1] Bogaert, B. and Tison, S.: "Equality and Disequality Constraints on Direct Subterms in Tree Automata", STACS'92, LNCS **577**, pp.161–171 (1992).

[2] Comon, H.: "Equational Formulaes in Order-Sorted Algebras", ICALP'90, LNCS **443**, pp.674–688 (1990).

[3] Kaji, Y., Fujiwara, T. and Kasami, T: "Solving a Unification Problem under Constrained Substitutions Using Tree Automata", FST & TCS'94, LNCS **880**, pp.276–287 (1994).

[4] Gécseq, F. and Steinby, M.: *Tree Automata*, Akadémiai Kiadó, Budapest (1984).

[5] Schmidt-Schauß, M.: *Computational Aspects of an Order-Sorted Logic with Term Declarations*, LNCS **395** (1987).

[6] Smolka, G., Nutt, W., Goguen, J.A. and Meseguer, J.: "Order-Sorted Equational Computation", In *Resolution of Equations in Algebraic Structures*, Vol. 2, *Rewriting Techniques*, pp.297–367, Academic Press (1989).

[7] Takada, K., Kaji, Y. and Kasami, T: "Syntactic Unification Problem under Constrained Substitutions", Technical Report of Nara Institute of Science and Technology, No. 96012 (1996).

Image sequence evaluation with timed transition diagrams

Christoph Brzoska

Nearly all of the symbolic formalism proposed for image sequence evaluation purposes are based explicitly or implicitly on some kind of grammar formalisms. Such formalisms allow to describe sequences of words, which can be identified with sequences of primitive conceptual descriptions of images and thereby to reduce recognition of image sequences to recognition of words generated by non-terminal symbols of the corresponding grammar. As a consequence, algorithms for accepting the languages (respectively, their extensions) associated tothe corresponding grammar formalisms are generally utilized to recognize and evaluate image sequences on the conceptual level.

In this talk, we will adopt a more general view of image recognition and see it as evaluation of events annotated with their occurrence time which a priori are not restricted by additional assumptions, for example, that the occurrence time of consecutive events increase monotonicly. We propose so called *timed transition diagrams* for description of such sequences of events, define its semantics, and sketch recognition and evaluation algorithms for the sequences specified. This formalism allows a natural hierarchical composition and is thereby well suited to be taken as basis for more advanced representation formalisms. The formalism itself is an extension of that of constraint automata proposed recently by K. Schulz and D. Gabbay.

This is a join work with K. Schäfer.

Term Graph Narrowing

Annegret Habel
Universität Hildesheim[*]

Detlef Plump
Universität Bremen[†]

1 Introduction

Narrowing was devised in the field of theorem proving as an equation solving method for the case that an equational theory is represented by a convergent term rewriting system. Fay [Fay79] was the first who showed the completeness of narrowing. In order to reduce the search space of the narrowing procedure, Hullot [Hul80] considered a strategy called basic narrowing and showed that it is still complete. Later, narrowing became popular as the computational paradigm for the combination of functional and logic programming. Since then there has been much research activity on improving the efficiency of narrowing and on relaxing the requirements for completeness (see the recent survey of Hanus [Han94]).

In order to implement narrowing efficiently, it is advisable to represent terms by graph-like data structures. This is because the simple tree representation of terms enforces copying of subterms in rewrite steps and hence leads to multiplication of evaluation work. In this paper we introduce *term graph narrowing* as an approach for solving equations by transformations on term graphs. Our main result is that term graph narrowing is complete for all term rewriting systems over which term graph rewriting is normalizing and confluent. This includes, in particular, all convergent term rewriting systems. Completeness means that if an equation is represented by a term graph, then for every solution of this equation, term graph narrowing can find a more general solution (that is, every solution is equivalent to an instance of a computed solution).

Term graph narrowing combines term graph rewriting with first-order term unification (see [SPE93] for a recent collection of papers on term graph rewriting). We use the term graph rewriting model studied in [HP91, Plu93a, Plu93b]. It allows, besides applications of rewrite rules, collapsing steps on term graphs to increase the degree of sharing. This model is complete with respect to equational deduction in the same sense as term rewriting is. Our completeness proof for term graph narrowing exploits existing results on term graph rewriting concerning the relationship to term rewriting with respect to termination, confluence and related properties.

This paper is an extended abstract of [HP96].

[*]Institut für Informatik, Marienburger Platz 22, 31141 Hildesheim, Germany.
E-mail: `habel@informatik.uni-hildesheim.de`

[†]Fachbereich Mathematik und Informatik, Postfach 33 04 40, 28334 Bremen, Germany.
E-mail: `det@informatik.uni-bremen.de`

2 Term graphs and substitutions

Let Σ be a set of *function symbols*. Each function symbol f comes with a natural number $\mathit{arity}(f) \geq 0$. Function symbols of arity 0 are called *constants*. We further assume that there is an infinite set Var of *variables* such that $\mathit{Var} \cap \Sigma = \emptyset$. For each variable x, we set $\mathit{arity}(x) = 0$.

A *hypergraph* over $\Sigma \cup \mathit{Var}$ is a system $G = \langle V_G, E_G, \mathit{lab}_G, \mathit{att}_G \rangle$ consisting of two finite sets V_G and E_G of *nodes* and *hyperedges*, a labelling function $\mathit{lab}_G: E_G \to \Sigma \cup \mathit{Var}$, and an attachment function $\mathit{att}_G: E_G \to V_G^*$ which assigns a string of nodes to a hyperedge e such that the length of $\mathit{att}_G(e)$ is $1 + \mathit{arity}(\mathit{lab}_G(e))$. In the following we call hypergraphs and hyperedges simply graphs and edges. The set of variables occurring in G is denoted by $\mathit{Var}(G)$, that is, $\mathit{Var}(G) = \mathit{lab}_G(E_G) \cap \mathit{Var}$.

Given a graph G and an edge e with $\mathit{att}_G(e) = v\, v_1 \ldots v_n$, node v is the *result node* of e while $v_1, \ldots, v_n$ are the *argument nodes*. Given two nodes v and v' in G, we write $v >_G^1 v'$ if there is an edge e with result node v such that v' is an argument node of e. The transitive (reflexive-transitive) closure of $>_G^1$ is denoted by $>_G$ ($\geq_G$). G is *acyclic* if $>_G$ is irreflexive. We write $G[v]$ for the graph consisting of all nodes v' with $v \geq_G v'$ and all edges having these nodes as result nodes.

Definition 2.1 (term graph) *A graph G is a* term graph *if*

(1) there is a node root_G such that $\mathit{root}_G \geq_G v$ for each node v,

(2) G is acyclic, and

(3) each node is the result node of a unique edge.

Figure 1 shows three term graphs with function symbols `f`, `g`, `a`, and variables `x`, `y`. The symbol `f` is binary, `g` is unary, and `a` is a constant. Edges are depicted as boxes with inscribed labels, and bullets represent nodes. A line connects each edge with its result node, while arrows point to the argument nodes. The order in the argument string is given by the left-to-right order of the arrows leaving the box.

Definition 2.2 (term representation) *A node v in a term graph G represents the term $\mathit{term}_G(v) = \mathit{lab}_G(e)(\mathit{term}_G(v_1), \ldots, \mathit{term}_G(v_n))$, where e is the unique edge with result node v, and where $\mathit{att}_G(e) = v\, v_1 \ldots v_n$. We write $\mathit{lab}_G(e)$ instead of $\mathit{lab}_G(e)()$ if $v_1 \ldots v_n$ is the empty string.*

Note that $\mathit{term}_G(v)$ is well-defined by properties (2) and (3) of Definition 2.1. In the following we abbreviate $\mathit{term}_G(\mathit{root}_G)$ by $\mathit{term}(G)$.

A *graph morphism* $f: G \to H$ between two graphs G and H consists of two functions $f_V: V_G \to V_H$ and $f_E: E_G \to E_H$ that preserve labels and attachment to nodes, that is, $\mathit{lab}_H \circ f_E = \mathit{lab}_G$ and $\mathit{att}_H \circ f_E = f_V^* \circ \mathit{att}_G$ (where $f_V^*: V_G^* \to V_H^*$ maps a string $v_1 \ldots v_n$ to $f_V(v_1) \ldots f_V(v_n)$). We omit the subscripts V and E if no confusion is possible. The morphism f is *injective* (*surjective, bijective*) if f_V and f_E are so. If f is bijective, then it is an *isomorphism*. In this case G and H are *isomorphic*, which is denoted by $G \cong H$.

Definition 2.3 (collapsing) *Given two term graphs G and H, G* collapses *to H if there is a graph morphism $c: G \to H$ mapping $root_G$ to $root_H$. This is denoted by $G \succeq_c H$ or simply by $G \succeq H$. We write $G \succ_c H$ or $G \succ H$ if c is non-injective. The latter kind of collapsing is said to be* proper. *A term graph G is* fully collapsed *if there is no H with $G \succ H$.*

It is easy to see that the collapse morphisms are the surjective morphisms between term graphs and that $G \succeq H$ implies $term(G) = term(H)$. An example of collapsing is given in Figure 1.

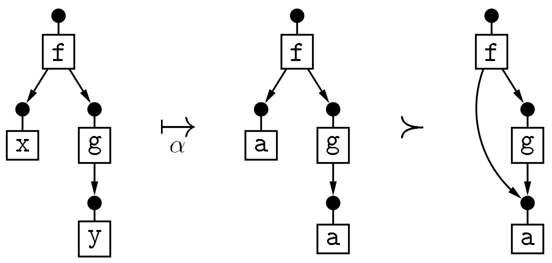


Figure 1: A substitution application and a collapsing

The term graph substitutions defined next correspond to first-order term substitutions. They are a special case of the general graph substitutions introduced in [PH96] which operate on variable edges with an arbitrary number of attachment nodes.

A *substitution pair x/G* consists of a variable x and a term graph G. Given a term graph H and an edge e in H labelled with x, the application of x/G to e proceeds in two steps: (1) Remove e from H, yielding the graph $H - \{e\}$, and (2) construct the disjoint union $(H - \{e\}) + G$ and fuse the result node of e with $root_G$. It is easy to see that the resulting graph is a term graph.

Definition 2.4 (term graph substitution) *A* term graph substitution *is a finite set $\alpha = \{x_1/G_1, \ldots, x_n/G_n\}$ of substitution pairs such that $x_1, \ldots, x_n$ are pairwise distinct and $x_i \neq term(G_i)$ for $i = 1, \ldots, n$. The* domain *of α is the set $Dom(\alpha) = \{x_1, \ldots, x_n\}$. The application of α to a term graph H yields the term graph $H\alpha$ which is obtained by applying all substitution pairs in α simultaneously to all edges with label in $Dom(\alpha)$.*

Example 2.5 *Let $\mathtt{x}, \mathtt{y}$ be variables and A be a term graph with $term(A) = \mathtt{a}$. An application of the substitution $\alpha = \{\mathtt{x}/A, \mathtt{y}/A\}$ is shown in Figure 1.*

Given a term graph G, we write $\underline{G}$ for the graph that results from removing all edges labelled with variables. If α is a term graph substitution, we assume for technical convenience that the unique graph morphism $in: \underline{G} \to G\alpha$ with $in(root_G) = root_{G\alpha}$ satifies $in(a) = a$ for all nodes and edges a.

Definition 2.6 (induced term substitution) *For every term graph substitution α, the* induced *term substitution α^{term} is defined by*

$$\alpha^{term} = \{x/term(G) \mid x/G \in \alpha\}.$$

3 Term graph rewriting

In this section we review the term graph rewriting model investigated in [HP91, Plu93a, Plu93b]. In particular, we state results concerning the soundness and completeness of term graph rewriting and the relation to term rewriting with respect to normalization and confluence.

We assume that the reader is familiar with basic concepts of term rewriting systems and abstract reduction systems (see, for example, [DJ90, Klo92]). In the following $\mathcal{R}$ denotes a term rewriting system and $\to_{\mathcal{R}}$ the rewrite relation associated with $\mathcal{R}$.

Let v be a node in a term graph G. Define $indegree_G(v) = \sum_{e \in E_G} \#_v(e)$, where for each edge e with $att_G(e) = v_0 v_1 \ldots v_n$, $\#_v(e)$ is the number of occurrences of v in $v_1 \ldots v_n$.

Definition 3.1 *For every term t, let $\Diamond t$ be a term graph representing t such that only variable nodes are shared, that is, (1) $indegree_G(v) \leq 1$ for each node v with $term_{\Diamond t}(v) \notin Var$, and (2) $v_1 = v_2$ for all nodes v_1,v_2 with $term_{\Diamond t}(v_1) = term_{\Diamond t}(v_2) \in Var$.*

Definition 3.2 (redex and preredex) *Let G be a term graph, v be a node in G, and $l \to r$ be a rule in $\mathcal{R}$. The pair $\langle v,\, l \to r\rangle$ is a* redex *if there is a graph morphism $red\colon \underline{\Diamond l} \to G$, called the* redex morphism*, such that $red(root_{\Diamond l}) = v$. The pair $\langle v,\, l \to r\rangle$ is a* preredex *if there is a term substitution σ such that $term_G(v) = l\sigma$.*

One can show that every redex is a preredex. The converse also holds if the rule $l \to r$ is left-linear, since then $\Diamond l$ is a tree[1]. However, if l contains repeated variables, then there need not exist a graph morphism sending $root_{\Diamond l}$ to v. In this case a suitable collapsing of G turns the preredex $\langle v,\, l \to r\rangle$ into a redex.

Definition 3.3 (term graph rewriting) *Let G, H be term graphs and $\langle v,\, l \to r\rangle$ be a redex in G with redex morphism $red\colon \underline{\Diamond l} \to G$. Then there is a* proper rewrite step *$G \Rightarrow_{v,\, l \to r} H$ if H is isomorphic to the term graph G_3 constructed as follows:*

(1) $G_1 = G - \{e\}$ is the graph obtained from G by removing the unique edge e having result node v.

(2) G_2 is the graph obtained from the disjoint union $G_1 + \underline{\Diamond r}$ by

- *identifying v with $root_{\Diamond r}$,*
- *identifying $red(v_1)$ with v_2, for each pair $\langle v_1, v_2\rangle \in V_{\Diamond l} \times V_{\Diamond r}$ with $term_{\Diamond l}(v_1) = term_{\Diamond r}(v_2) \in Var$.*

(3) $G_3 = G_2[root_G]$ is the term graph obtained from G_2 by removing all nodes and edges not reachable from $root_G$ ("garbage collection").

[1]A term graph is a *tree* if all nodes but the root have indegree one.

We define the term graph rewrite relation $\Rightarrow_{\mathcal{R}}$ by adding proper collapse steps: $G \Rightarrow_{\mathcal{R}} H$ if $G \succ H$ or $G \Rightarrow_{v,\,l \to r} H$ for some redex $\langle v,\, l \to r\rangle$.

A *term graph rewrite derivation* is either an isomorphism $G \to H$, which is a derivation of length 0, or a non-empty sequence

$$G = G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_n = H.$$

We denote such a derivation by $G \Rightarrow^*_{\mathcal{R}} H$.

Example 3.4 *A term graph rewrite step with rule* $\mathtt{f(x,g(x))} \to \mathtt{h(x,x,a)}$ *is given in Figure 2. Note that the left term graph is obtained from the middle term graph of Figure 1 by collapsing. However, the rule is not applicable to the middle term graph of Figure 1 because there is no graph morphism from* $\underline{\Diamond\mathtt{f(x,g(x))}}$ *into that graph.*

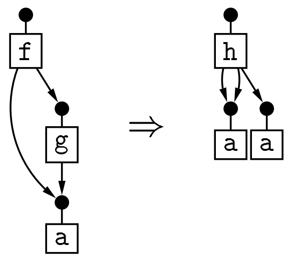


Figure 2: A term graph rewrite step

The term graph rewrite relation $\Rightarrow_{\mathcal{R}}$ is sound with respect to term rewriting in the sense that every proper step $G \Rightarrow_{v,\,l\to r} H$ corresponds to a parallel application of $l \to r$ to several occurrences of the subterm $\mathit{term}_G(v)$ in $\mathit{term}(G)$. This parallelism is the reason for the possible speed-up of term graph rewriting with respect to term rewriting. Note that if $G \Rightarrow_{\mathcal{R}} H$ is a collapse step, then $\mathit{term}(G) = \mathit{term}(H)$ and hence $\mathit{term}(G) \to^*_{\mathcal{R}} \mathit{term}(H)$.

Theorem 3.5 (soundness of rewriting [HP91]) *For all term graphs G and H, $G \underset{\mathcal{R}}{\Rightarrow} H$ implies $\mathit{term}(G) \overset{*}{\underset{\mathcal{R}}{\to}} \mathit{term}(H)$.*

The converse of the implication (with $\Rightarrow_{\mathcal{R}}$ replaced by $\Rightarrow^*_{\mathcal{R}}$) does not hold since certain term rewrite derivations do not correspond to term graph rewrite derivations.

Although not all term derivations possess corresponding graph derivations, the conversion of term graphs is complete with respect to conversion of terms.

Theorem 3.6 (completeness of rewriting [Plu93a]) *For all term graphs G and H, $\mathit{term}(G) \overset{*}{\underset{\mathcal{R}}{\leftrightarrow}} \mathit{term}(H)$ if and only if $G \overset{*}{\underset{\mathcal{R}}{\Leftrightarrow}} H$.*

The next two theorems explain the relationship between term graph rewriting and term rewriting with respect to normalization, confluence, and convergence. These results are used in proving the completeness of term graph narrowing.

Theorem 3.7 (normalization [HP91])

1. *A term graph G is a normal form with respect to $\Rightarrow_{\mathcal{R}}$ if and only if G is fully collapsed and $term(G)$ is a normal form with respect to $\rightarrow_{\mathcal{R}}$.*

2. *If $\Rightarrow_{\mathcal{R}}$ is normalizing, then so is $\rightarrow_{\mathcal{R}}$.*

The converse of the second statement does not hold (see [Plu93b] for a counterexample).

Theorem 3.8 (confluence and convergence [Plu93a])

1. *If $\Rightarrow_{\mathcal{R}}$ is confluent, then so is $\rightarrow_{\mathcal{R}}$.*

2. *If $\rightarrow_{\mathcal{R}}$ is convergent, then so is $\Rightarrow_{\mathcal{R}}$.*

For both statements, the converse does not hold (see [Plu93a]).

4 Term graph narrowing

Our goal is to solve term equations by transformations on term graphs. To this end we define term graph narrowing and establish a completeness result which corresponds to Hullot's result for term narrowing [Hul80, MH94].

An *equation* $s = t$ is a pair of terms s and t. We are interested in solutions to such equations modulo the equational theory induced by a term rewriting system $\mathcal{R}$. That is, a *solution* of $s = t$ is a term substitution σ such that $s\sigma \leftrightarrow^*_{\mathcal{R}} t\sigma$. If such a solution exists, we say that s and t are *$\mathcal{R}$-unifiable.*

Definition 4.1 *(term graph narrowing) Let G and H be term graphs, v be a non-variable node in G, $l \rightarrow r$ be a rule*[2] *in $\mathcal{R}$, and α be a term graph substitution. Then there is a* narrowing step *$G \rightsquigarrow_{v,\,l\rightarrow r,\,\alpha} H$ if α^{term} is a most general unifier of l and $term_G(v)$, and*

$$G\alpha \underset{c}{\succeq} G' \underset{c(v),\,l\rightarrow r}{\Longrightarrow} H$$

for some collapsing $G\alpha \succeq_c G'$. We denote such a step also by $G \rightsquigarrow_\alpha H$.

The collapsing *after* application of α is necessary to make narrowing complete. For, if $l \rightarrow r$ is not left-linear, then there need not exist a step $G\alpha \Rightarrow_{v,\,l\rightarrow r} H$ even if l is unifiable with $term_G(v)$ and G is fully collapsed (see Example 4.2).

A *term graph narrowing derivation* $G \rightsquigarrow^*_\alpha H$ is either an isomorphism $G \rightarrow H$ together with the empty substitution or a non-empty sequence

$$G = G_0 \rightsquigarrow_{\alpha_1} G_1 \rightsquigarrow_{\alpha_2} \ldots \rightsquigarrow_{\alpha_n} G_n = H$$

such that $\alpha = \alpha_1\alpha_2\ldots\alpha_n$.

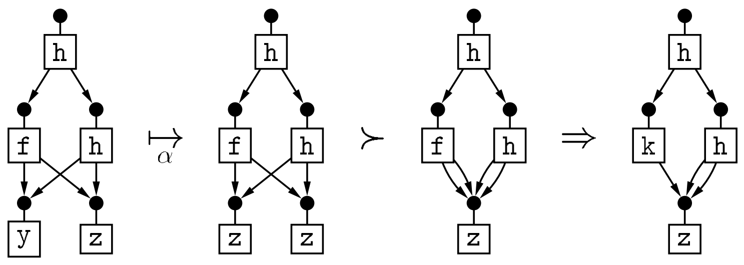


Figure 3: The components of a term graph narrowing step

Example 4.2 *Figure 3 shows a term graph narrowing step in its three component steps. The applied term rewrite rule is* $\mathtt{f(x,x)} \to \mathtt{k(x)}$ *and the computed term graph substitution is* $\alpha = \{\mathtt{x}/Z, \mathtt{y}/Z\}$, *where* $term(Z) = \mathtt{z}$. *Note that* α^{term} *is a most general unifier of* $\mathtt{f(x,x)}$ *and* $\mathtt{f(y,z)}$. *Since* $\mathtt{f(x,x)} \to \mathtt{k(x)}$ *is non-left-linear, there is no graph morphism from* $\overline{\Diamond \mathtt{f(x,x)}}$ *to the term graph resulting from the application of* α. *That is, the rule cannot be applied to this graph. We first have to identify the two* z*-labelled edges by a collapsing.*

From now on we assume that $\mathcal{R}$ contains the rule $\mathtt{x} =^? \mathtt{x} \to \mathtt{true}$, where the binary function symbol $=^?$ and the constant $\mathtt{true}$ do not occur in any other rule. A *goal* is a term of the form $s =^? t$ such that s and t do not contain $=^?$ and $\mathtt{true}$. We denote by $\Delta\mathtt{true}$ a term graph representing $\mathtt{true}$.

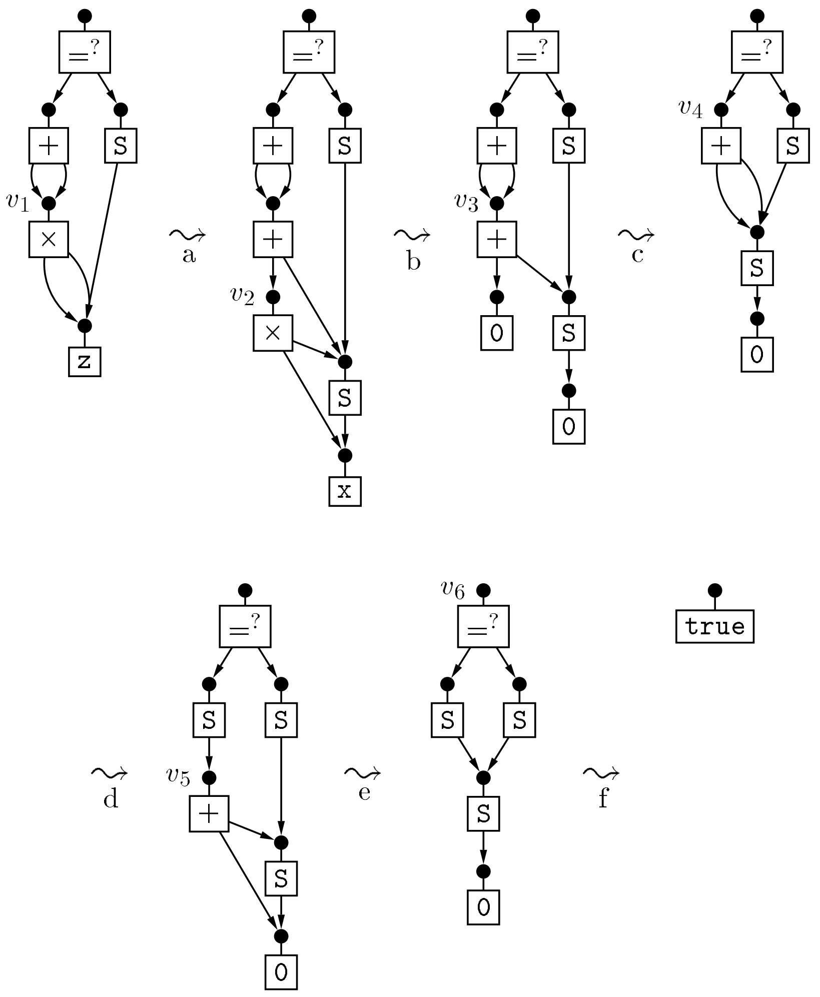
Example 4.3 *Let* $\mathcal{R}$ *consist of the following rules:*

$$\begin{array}{lll@{\qquad\qquad}lll}
\mathtt{0} + \mathtt{x} & \to & \mathtt{x} & \mathtt{0} \times \mathtt{x} & \to & \mathtt{0} \\
\mathtt{S(x)} + \mathtt{y} & \to & \mathtt{S(x + y)} & \mathtt{S(x)} \times \mathtt{y} & \to & \mathtt{(x \times y) + y} \\
 & & & \mathtt{x} =^? \mathtt{x} & \to & \mathtt{true}
\end{array}$$

Suppose that we want to solve the goal $(\mathtt{z} \times \mathtt{z}) + (\mathtt{z} \times \mathtt{z}) =^? \mathtt{S(z)}$. *Figure 4 shows a term graph narrowing derivation starting from a fully collapsed representation of this goal. For each narrowing step, the applied rewrite rule and the involved term substitution are given. Note that steps c,d and e are proper rewrite steps and that step f consists of a collapse step and a proper rewrite step. The derivation computes the term substitution* $\{\mathtt{x}/\mathtt{0}, \mathtt{x}'/\mathtt{S(0)}, \mathtt{y}/\mathtt{S(0)}, \mathtt{z}/\mathtt{S(0)}\}$ *in six steps. Restricting this substitution to the variables of the initial term graph yields the solution* $\{\mathtt{z}/\mathtt{S(0)}\}$. *Solving the same goal by term narrowing requires nine steps, demonstrating that term graph narrowing speeds up the computation.*

Theorem 4.4 (soundness of narrowing) *Let* G *be a term graph such that* $term(G)$ *is a goal* $s =^? t$. *If* $G \rightsquigarrow_\alpha^* \Delta\mathtt{true}$, *then* α^{term} *is an* $\mathcal{R}$*-unifier of* s *and* t.

[2]We assume that this rule has no common variables with G. If this is not the case, then the variables in $l \to r$ are renamed into variables from $Var - Var(G)$.



step	node	rewrite rule	substitution
a	v_1	$\mathtt{S(x)} \times \mathtt{y} \to (\mathtt{x} \times \mathtt{y}) + \mathtt{y}$	$\{\mathtt{y}/\mathtt{S(x)}, \mathtt{z}/\mathtt{S(x)}\}$
b	v_2	$\mathtt{0} \times \mathtt{x}' \to \mathtt{0}$	$\{\mathtt{x}/\mathtt{0}, \mathtt{x}'/\mathtt{S(0)}\}$
c	v_3	$\mathtt{0} + \mathtt{x} \to \mathtt{x}$	$\{\mathtt{x}/\mathtt{S(0)}\}$
d	v_4	$\mathtt{S(x)} + \mathtt{y} \to \mathtt{S}(\mathtt{x} + \mathtt{y})$	$\{\mathtt{x}/\mathtt{0}, \mathtt{y}/\mathtt{S(0)}\}$
e	v_5	$\mathtt{0} + \mathtt{x} \to \mathtt{x}$	$\{\mathtt{x}/\mathtt{S(0)}\}$
f	v_6	$\mathtt{x} =^? \mathtt{x} \to \mathtt{true}$	$\{\mathtt{x}/\mathtt{S(S(0))}\}$

Figure 4: A term graph narrowing derivation with its rewrite rules and substitutions

In order to prove the completeness of term graph narrowing, we use the following lifting lemma. It allows to transform term graph rewrite derivations into term graph narrowing derivations.

Lemma 4.5 (Lifting Lemma) *Let $G \Rightarrow_{\mathcal{R}}^{*} H$ be a rewrite derivation and G' be a term graph such that $G'\alpha = G$ for some normalized substitution α. Moreover, let V be a finite subset of Var such that $Var(G') \cup Dom(\alpha) \subseteq V$. Then there is a narrowing derivation $G' \leadsto_{\beta}^{*} H'$ and a normalized substitution γ such that $H'\gamma \succeq H$ and $(\beta\gamma)|_V = \alpha$.*

The proof of this lemma consists of two steps: at first the given rewrite derivation is transformed into a "minimally collapsing" rewrite derivation with a subsequent collapsing (see Theorem 4.8), and then this derivation is directly lifted to a narrowing derivation. In a minimally collapsing derivation, collapse steps are only used to turn preredexes of non-left-linear rewrite rules into redexes.

Definition 4.6 (minimal collapsing) *A collapsing $G \succeq M$ is* minimal *with respect to a redex $\langle v, l \to r\rangle$ in M if for each term graph M' with $G \succeq M' \succ M$ and each preimage v' of v in M', the preredex $\langle v', l \to r\rangle$ is not a redex.*

In particular, if G and M are isomorphic, then $G \succeq M$ is minimal since no M' with $G \succeq M' \succ M$ exists. A proper collapsing $G \succ M$ is minimal only if $l \to r$ is non-left-linear and cannot be applied at any preimage of v in G.

Definition 4.7 (minimally collapsing rewrite derivation) *A rewrite derivation $P \Rightarrow_{\mathcal{R}}^{*} Q$ is* minimally collapsing *if each collapse step $G \succ M$ in the derivation is followed by a proper rewrite step $M \Rightarrow_{v, l\to r} N$ such that $G \succ M$ is minimal with respect to $\langle v, l \to r\rangle$.*

By the following result, derivability with respect to the rewrite relation $\Rightarrow_{\mathcal{R}}$ is not affected if one restricts to minimally collapsing derivations with a subsequent collapsing.

Theorem 4.8 (transformation of derivations) *For every derivation $G \Rightarrow_{\mathcal{R}}^{*} H$ there is a minimally collapsing derivation $G \Rightarrow_{\mathcal{R}}^{*} H'$ such that $H' \succeq H$.*

Given two term substitutions σ and τ, and a subset V of *Var*, we write $\sigma =_{\mathcal{R}} \tau\ [V]$ if $x\sigma \leftrightarrow_{\mathcal{R}}^{*} x\tau$ for each $x \in V$. We write $\sigma \leq_{\mathcal{R}} \tau\ [V]$ if there is a substitution ρ such that $\sigma\rho =_{\mathcal{R}} \tau\ [V]$. The *restriction* $\sigma|_V$ of a term substitution σ to a subset V of *Var* is the substitution $\{x/t \in \sigma \mid x \in V\}$. The restriction of a term graph substitution is defined analogously. A term graph substitution $\alpha = \{x_1/G_1, \ldots, x_n/G_n\}$ is *normalized* if $G_1, \ldots, G_n$ are normal forms with respect to $\Rightarrow_{\mathcal{R}}$.

Theorem 4.9 (completeness of narrowing) *Let $\Rightarrow_{\mathcal{R}}$ be convergent and G be a term graph such that term(G) is a goal $s =^{?} t$. Then for every $\mathcal{R}$-unifier σ of s and t, there is a narrowing derivation $G \leadsto_{\beta}^{*} \Delta$*`true` *such that $\beta^{term} \leq_{\mathcal{R}} \sigma\ [Var(G)]$.*

By Theorem 3.8.2, $\Rightarrow_{\mathcal{R}}$ is convergent whenever $\rightarrow_{\mathcal{R}}$ is. Hence we have the following corollary.

Corollary 4.10 *Term graph narrowing is complete for every convergent term rewriting system.*

Inspecting the proof of Theorem 4.9 in [HP96] shows that termination of $\Rightarrow_{\mathcal{R}}$ can be relaxed to normalization. Hence we can strengthen the completeness result as follows.

Theorem 4.11 *Term graph narrowing is complete whenever term graph rewriting is normalizing and confluent.*

5 Conclusion

We have introduced term graph narrowing as a mechanism for solving equations by transformations on term graphs. The advantage of term graph narrowing over conventional narrowing is that common subterms can be shared. Sharing saves not only space but also time since repeated computations can be avoided.

We have shown that term graph narrowing is a complete equation solving method for all term rewriting systems over which term graph rewriting is normalizing and confluent. This includes all convergent term rewriting systems. To achieve completeness, narrowing steps have to allow a collapsing between the substitution application and the rewrite step.

Our completeness proof is based on two results. On the one hand, we have shown that minimally collapsing rewrite derivations can be lifted to narrowing derivations, where minimally collapsing derivations contain only collapse steps that are necessary to enable applications of non-left-linear rewrite rules. On the other hand, we have established a normal form result for term graph rewrite derivations, showing that every derivation can be transformed into a minimally collapsing derivation together with a subsequent collapsing.

Our results suggest to consider *minimally collapsing term graph narrowing* as a restricted form of term graph narrowing in which narrowing steps contain only collapse steps that are minimal with respect to the rewrite steps. From our proofs it is easy to see that minimally collapsing narrowing is in the same sense complete as general term graph narrowing.

References

[Apt90] K.R. Apt. Logic programming. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science,* volume B, chapter 10. Elsevier (1990)

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science,* volume B, chapter 6. Elsevier (1990)

[Fay79] M. Fay. First-order unification in an equational theory. In *Proc. 4th Workshop on Automated Deduction*, pages 161–167, Austin, Texas. Academic Press (1979)

[Hab92] A. Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag (1992)

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *The Journal of Logic Programming*, 19 & 20:583–628 (1994)

[HP91] B. Hoffmann and D. Plump. Implementing term rewriting by jungle evaluation. *RAIRO Theoretical Informatics and Applications*, 25(5):445–472 (1991)

[HP96] A. Habel and D. Plump. Term Graph Narrowing. *Mathematical Structures in Computer Science*. To appear.

[Hul80] J.-M. Hullot. Canonical forms and unification. In *Proc. 5th International Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag (1980)

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science,* volume 2, pages 1–116. Oxford University Press (1992)

[MH94] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253 (1994)

[Plu93a] D. Plump. Collapsed tree rewriting: Completeness, confluence, and modularity. In *Proc. Conditional Term Rewriting Systems*, volume 656 of *Lecture Notes in Computer Science*, pages 97–112. Springer-Verlag (1993)

[Plu93b] D. Plump. Evaluation of functional expressions by hypergraph rewriting. Dissertation, Universität Bremen, Fachbereich Mathematik und Informatik (1993)

[PH96] D. Plump and A. Habel. Graph unification and matching. In *Proc. Workshop on Graph Grammars and Their Application to Computer Science '94*, Lecture Notes in Computer Science. Springer-Verlag. To appear.

[SPE93] R. Sleep, R. Plasmeijer, and M. van Eekelen, editors. *Term Graph Rewriting: Theory and Practice.* John Wiley (1993)

A rule based first order unification for higher order types

Emmanuel Engel

Our goal is to define a clean rule based first order unification for higher order types. We do not wants to take bounds and free variable from different sets and then, the main difficulty is to manage properly the bound variables. To solve this problem we use a rule based algorithm. We prove termination, correctness and completeness.

ISHTAR: A Functional Logic Language with Polymorphic Order-Sorted Types *

J.M. Almendros-Jiménez and A. Gil-Luezas†
Dpto. Informática y Automática. Facultad de CC. Matemáticas.
Universidad Complutense. Madrid. 28040.
E-mail:{jesusmal,anagil}@eucmos.sim.ucm.es.

1 Introduction

Type systems have been traditionally considered in functional languages and incorporated as an extension to logic programming. The usefulness of type systems has been widely accepted to detect programming errors and to obtain more readable programs and run-time optimizations.

Polymorphic order-sorted type systems include both parametric and inclusion polymorphism providing more powerful expressivity. Parametric polymorphism parametrizes types by means of type variables which represent any type whereas inclusion polymorphism allows subtype relations between types.

Parametric polymorphism was introduced in functional languages within the language ML and incorporated as an extension to logic programming. These type systems were proved to be in general static type systems, that is, type information is not required at run-time. Inclusion polymorphism was studied for languages base d on order-sorted equational logic and incorporated to the language OBJ-3. By contrast, type systems with inclusion polymorphism are dynamic type systems, that is type information must be checked at run-time. The combination of both kind of polymorphism has been studied for logic programming (cfr. [Smo89], [Han91], [HiTo92], [Bei95]) and for functional programming (cfr. [FuMi90], [Smi94]). These approaches investigate the problem of type inference (cfr. [Smo89], [FuMi90], [Smi94], [Bei95]) or operational semantics based on typed unification for logic programming (cfr. [Smo89], [Han91], [HiTo92]). TEL is a language which combines both kinds of polymorphism in a logic language.

The integration of logic and functional programming has been studied in the last years (see [Han94a] for a survey). Operational semantics based on lazy narrowing has been presented in[GHLR96]. This combination is adequated to include lazy evaluation allowing partial non-strict functions and infinite values. Parametric type systems for functional-logic programming have been studied in [Han90].

*An extended version can be found in [AlGi96].

†This research has been partially supported by the Spanish CICYT (project TIC 95-0433-C03-01 "CPD").

This work investigates the integration of a polymorphic order-sorted type system into functional logic programming including lazy functions. Operational semantics for logic languages with polymorphic order-sorted types only requires type checking for data terms during the unification process. However, in lazy narrowing, type checking of expressions not being data terms could require evaluation. For this reason, a lazy type checking must be introduced and combined with lazy narrowing. We will define a lazy notion of type checking based on the type declarations, in such a way that the expressions involving function symbols will not be evaluated if the type declaration is enough to deduce the type.

The type system we present is based on that presented in [HiTo92]. Hence it allows subtype relations between type constructors with the same arity, defining a quasi-lattice and satisfying the monotonicity property. Our language ISHTAR follows the line of BABEL [MoRo92] and BABLOG [AGL94]. The programs consist of a specification of types, type declarations for data constructors and functions and a set of constructor-based well-typed conditional rewriting rules. The rule conditions include data and type conditions and work as constraints for the rule applicability. The user can make a case distinction based on subtypes of the type declaration or can restrict the applicability of the rule for some cases. In any case, type conditions must assure the well-typedness of the expressions involved in the rule. This well-typedness property is crucial for a sound operational semantics. We present an operational semantics based on transformation rules of data and type constraint systems, and combining lazy narrowing and type solving. We extend the operational semantics of lazy narrowing presented in [GHLR96] by considering type constraints.

2 Some Examples

Our typed programs consist of a specification of types, a set of type declarations for data constructors and functions and a set of program rules for every function as the following example shows.

TYPES

$opnat, nat \leq int$
$negint, zero \leq opnat$
$posint, zero \leq nat$

$technician, artist \leq person$
$doctor, comp\_scientist,$
$architect \leq technician$
$sculptor, painter, musician \leq artist$

$elist(\alpha), nelist(\alpha) \leq list(\alpha)$
$earbin(\alpha), nearbin(\alpha) \leq arbin(\alpha)$
$bool$

DATA CONSTRUCTORS

$0 : zero$
$suc : nat \rightarrow posint$
$pred : opnat \rightarrow negint$

$nil : elist(\alpha)$
$[\_|\_] : \alpha \times list(\alpha) \rightarrow nelist(\alpha)$

$empty : earbin(\alpha)$
$tree : arbin(\alpha) \times \alpha \times arbin(\alpha) \rightarrow nearbin(\alpha)$

$john : doctor$
$frank, david : comp\_scientist$
$thomas, margaret : architect$
$richard, marie : sculptor$
$robert, nathalie : painter$
$michael : musician$

$true, false : bool$

FUNCTIONS

$head : nelist(\alpha) \rightarrow \alpha$
$head([X|X_s]) := X \Leftarrow X : \alpha, X_s : list(\alpha)$

$tail : nelist(\alpha) \rightarrow list(\alpha)$
$tail([X|X_s]) := X_s \Leftarrow X : \alpha, X_s : list(\alpha)$

$second : nelist(\alpha) \rightarrow \alpha$
$second(X) := head(tail(X))$
$\quad \Leftarrow X : nelist(\alpha), tail(X) : nelist(\alpha)$

$append : list(\alpha) \times list(\alpha) \rightarrow list(\alpha)$
$append(nil, L) := L \Leftarrow L : list(\alpha)$
$append([X|L1], L2) := [X|append(L1, L2)]$
$\quad \Leftarrow X : \alpha, L1 : list(\alpha), L2 : list(\alpha)$

$preord : arbin(\alpha) \rightarrow list(\alpha)$
$preord(empty) := nil$
$preord(tree(LT, X, RT)) :=$
$[X|append(preord(LT), preord(RT))]$
$\quad \Leftarrow X : \alpha, LT : arbin(\alpha), RT : arbin(\alpha)$

$n\_th : nat \times list(\alpha) \rightarrow \alpha$
$n\_th(0, [X|L]) := X \Leftarrow X : \alpha, L : \alpha$
$n\_th(s(N), [X|L]) := n\_th(N, L)$
$\quad \Leftarrow N : nat, X : \alpha, L : list(\alpha)$

$father : person \rightarrow person$
$father(david) := michael$
$father(thomas) := richard$
$father(richard) := david...$

$mother : person \rightarrow person$
$mother(richard) := marie$
$mother(marie) := nathalie...$

$fam\_tree : person \rightarrow nearbin(person)$
$fam\_tree(X) := tree(fam\_tree(father(X)),$
$\quad X, fam\_tree(mother(X))) \Leftarrow X : person$

$any\_anc\_artist? : person \rightarrow bool$
$any\_anc\_artist?(X) := true$
$\quad \Leftarrow X : person, sel\_artist(preord$
$\quad (fam\_tree(X))) : nelist(person)$
$any\_anc\_artist?(X) := false$
$\quad \Leftarrow X : person, sel\_artist(preord$
$\quad (fam\_tree(X))) : elist(person)$

$sel\_artist : list(person) \rightarrow list(artist)$
$sel\_artist(nil) := nil$
$sel\_artist([X|L]) := [X|sel\_artist(L)]$
$\quad \Leftarrow X : artist, L : list(person)$
$sel\_artist([X|L]) := sel\_artist(L)$
$\quad \Leftarrow X : technician, L : list(person)$

The program rules are conditional rewriting rules including data and type conditions whose intended meaning is to constraint the applicability of a rule. In this example, we show the expressivity of our language. We can classify people into artists: painters, sculptors and musicians, and technicians: doctors, computer scientists and architects.We can define polymorphic constructors for lists and trees. We can use them to manage data bases, for example, the family tree of a person (describing the profession of every ancestor). Furthermore, we can define polymorphic functions as head, tail, second, append, preord or n_th. The language also allows to formulate several queries in a simple way.

Goals

Goals of programs are as conditions rules, for example, for the program on the example we can write:

?-$P{=}{=}father(X), M{=}{=}mother(X)\Box n\_th(N, preord(fam\_tree(X))){:}musician,$
$N{:}posint, X{:}person, X{=}richard, P{=}david, M{=}marie,$
$N{=}suc(suc(0))\Box X{:}sculptor, P{:}comp\_scientist, M{:}sculptor, N{:}posint$

The lazy notion of type checking is shown in the following example.

Lazy Type Checking

The type constraint $sel\_artist(preord\ (fam\_tree(X))) : list(artist)$ does not require evaluation due to $sel\_artist : list(person) \rightarrow list(artist)$. By contrast, $sel\_artist(preord\ (fam\_tree(X))) : nelist(artist)$ requires evaluation, however, the lazyness of the language does not force to compute the complete list of artists. It is only required to find one.

The operational semantics that we present is based on transformation rules of constraints combining lazy narrowing and type solving. It works with strict equalities ($l == r$) that represent that l and r must be evaluated into a finite totally defined value; non strict equalities ($e \doteq t$) that represent the lazy unification of e and t (where e is any expression and t is a pattern of a function), type conditions ($e : \sigma$) that will be solved by narrowing e into an expression of type σ, subtype conditions ($\tau \sqsubseteq \tau'$) and solved typed ($\gamma = \sigma$). Goals and rules only contain strict equalities and type conditions. The rest will appear during the transformation process. As result of this process we obtain a solved system including an environment (type assumptions for data variables), a substitution for data variables, a substitution for type variables and subtype conditions of the form $\alpha \sqsubseteq \beta$. During the process; for every data variable X we use a type variable α_X in order to compute the type. The following example shows the process.

Constraint Transformations

Given the goal $Y == second([0, suc(0)])\Box Y : \alpha_Y$ the operational calculus proceeds as follows:

$$Y == second([0, suc(0)])\Box Y : \alpha_Y \hookrightarrow^*_{OS}$$

$$Y == head(tail([0, suc(0)]))\Box Y : \alpha_Y, [0, suc(0)] : nelist(\alpha), tail([0, suc(0)]) : nelist(\alpha) \hookrightarrow^*_{OS}$$

$$Y == head(tail([0, suc(0)]))\Box Y : \alpha_Y, [0, suc(0)] : nelist(\alpha), [suc(0)] : nelist(\alpha) \hookrightarrow^*_{OS}$$

$$Y == head(tail([0, suc(0)]))\Box Y : \alpha_Y, \alpha = int \hookrightarrow^*_{OS}$$

$$Y == suc(0)\Box Y : \alpha_Y, \alpha = int, suc(0) : \alpha_Y \hookrightarrow^*_{OS}$$

$$Y == suc(0)\Box Y : posint, \alpha = int$$

The operational semantics is lazy in the sense of the evaluation of data expressions is performed only if is necessary to obtain values for data variables or to check types. For it, we prove results of soundness and completeness w.r.t. a lazy semantic calculus.

We are also interested in an efficient implementation of the operational semantics that takes advantages from the typedness of the language.

References

[AlGi96] J.M.ALMENDROS-JIMÉNEZ, A. GIL-LUEZAS. *Lazy Narrowing with Polymorphic Order-Sorted Types (Extended Version)*, Technical Report DIA 30/96. Universidad Complutense. 1996.

[AGL94] P. ARENAS-SÁNCHEZ, A.GIL-LUEZAS, F.LÓPEZ-FRAGUAS. *Combining Lazy Narrowing with Disequality Constraints*, Procs. PLILP'94, Springer LNCS 844, pp. 385-399, 1994.

[Bei95] CH. BEIERLE. *Type Inferencing for Polymorphic Order-Sorted Logic Programs*, Procs. of the 12th International Conference on Logic Programming, The MIT Press, pp. 765-779, 1995.

[FuMi90] Y. FUH, P. MISHRA. *Type Inference with Subtypes*, Theoretical Computer Science, 73, pp. 155-175, 1990.

[GHLR96] J.C. GONZÁLEZ-MORENO, T. HORTALÁ-GONZÁLEZ, F. LÓPEZ FRAGUAS, M. RODRÍGUEZ-ARTALEJO. *A Rewriting Logic for Declarative Programming*, Procs. ESOP'96, Springer LNCS 1058, pp. 156-172, 1996.

[Han90] M. HANUS. *A Functional and Logic Language with Polymorphic Types*, Procs. Int. Symp. on Design and Implementation of Symbolic Computation Systems, Springer LNCS 429, pp. 215-224, 1990.

[Han91] M. HANUS. *Parametric Order-Sorted Types in Logic Programming*, Procs. TAPSOFT'91, Springer LNCS 494, pp. 181-200, 1991.

[Han94a] M. HANUS. *The Integration of Functions into Logic Programming: A Survey,* Journal of Logic Programming (19,20), Special issue "Ten Years of Logic Programming", pp. 583-628, 1994.

[HiTo92] P.M. HILL, R.W. TOPOR. *A Semantics for Typed Logic Programming*, Chapter 1, Types in Logic Programming, Logic Programming Series, Frank Pfenning Editor, The MIT Press, pp. 1-58, 1992.

[MoRo92] J.J. MORENO-NAVARRO, M. RODRÍGUEZ-ARTALEJO. *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Programming, 12, pp. 191-223, 1992.

[Smi94] G.S. SMITH. *Principal Type Schemes for Functional Programs with Overloading and Subtyping*, Science of Computer Programming, 23, pp. 197-226, 1994.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*, PhD thesis, Universitat Kaiserslautern, Germany, 1989.

Equational Unification and Type Inference: A Case Study

Andrew J. Kennedy
LIX, École Polytechnique
91128 Palaiseau cedex, France
kennedy@lix.polytechnique.fr

The application of free unification to type inference for programminglanguages is well-known [6, 1]. In this talk a type system is presented whose inference algorithm is based on *equational* unification for Abelian groups.

Programs written in languages like ML must be well-typed before a compiler will accept them. In the same way, physical equations in science must be dimensionally consistent. The ML type system can be extended to support the notion of physical dimension. It is then possible to be sure that dimension errors (such as adding a length to a time) will not occur at run-time.

This aim is achieved by adding a parameter to numeric types which represents units of measure such as kilograms or metres per second. Unit expressions are formed from base units (such as the SI units kilograms, metres and seconds) combined using product (for example, to obtain metres squared) and inverse (to express velocities in metres per second, for instance). Also a symbol representing 'no units' is required for dimensionless quantities such as refractive index and angle. We therefore use the following syntax for units of measure:

$$\mu \ ::= \ u \ \mid \ b \ \mid \ \mathbf{1} \ \mid \ \mu_1\mu_2 \ \mid \ \mu^{-1}$$

Here b ranges over some set of base units $\mathcal{B}$, dimensionless quantities have the units $\mathbf{1}$, product is represented by juxtaposition and inverse by its usual notation. For large programs it is essential that functions can be written which are *polymorphic* in their units of measure – even something as simple as a squaring function requires this. Hence the syntax of unitsallows unit *variables*, ranged over by u.

Types in the language have the syntax

$$\tau \ ::= \ t \ \mid \ \tau_1 \to \tau_2 \ \mid \ \tau_1 \times \tau_2 \ \mid \ \mathsf{num}\ \mu$$

where t ranges over type variables (to allow ordinary type polymorphism), function types and cartesian product types are represented by the usual $\tau_1 \to \tau_2$ and $\tau_1 \times \tau_2$, and numeric types have the form $\mathsf{num}\ \mu$ for a units of measure expression μ. Then the standard arithmetic operations are given the following polymorphic types:

$$+, - \quad : \quad \mathsf{num}\ u \times \mathsf{num}\ u \to \mathsf{num}\ u$$

$$\begin{array}{rcl} * & : & \mathsf{num}\ u_1 \times \mathsf{num}\ u_2 \rightarrow \mathsf{num}\ u_1u_2 \\ / & : & \mathsf{num}\ u_1 \times \mathsf{num}\ u_2 \rightarrow \mathsf{num}\ u_1u_2^{-1} \end{array}$$

Notice how addition and subtraction insist that their arguments have the same units of measure, whereas multiplication and division operations produce results whose units of measure are respectively the product and quotient of the units of their arguments.

So far, the type system resembles many an extension to ML, except that we allow quantification over two different syntactic classes. What distinguishes it is that units of measure satisfy certain *equations*, namely those of an Abelian group. Hence we incorporate into the type system an equational theory $=_E$ generated by the following set E of equations:

$$\begin{array}{rcll} u_1u_2 & = & u_2u_1 & \textit{commutativity} \\ (u_1u_2)u_3 & = & u_1(u_2u_3) & \textit{associativity} \\ \mathbf{1}u & = & u & \textit{identity} \\ uu^{-1} & = & \mathbf{1} & \textit{inverses} \end{array}$$

The typing rules are then extended with the following rule:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2}\ \tau_1 =_E \tau_2$$

Here Γ is a type *environment* which maps program identifiers to polymorphic types, and e is an expression in the programming language.

The ML type inference algorithm makes use of free unification whenever it processes a function application $e_1\ e_2$. The type τ_1 deduced for e_1 is unified with the type $\tau_2 \rightarrow t$, where τ_2 is the type deduced for e_2 and t is a freshly-generated type variable. The fact that free unification is *unitary* (for any two unifiable types there is a single most general unifier) leads to the *principal types* property of ML (for any typable expression there is a single most general type). A quick scan of any survey on unification [2, 8] will reveal that few algebraic theories are unitary unifying. Fortunately, Abelian group unification with free constants *is* such a theory. Hence the use of free unification in the ML type inference algorithm can be replaced by AG-unification to obtainprincipal types for units of measure [3, 4].

To illustrate the power of the system, here is a function written in ML which differentiates another function numerically.

```
fun diff h f = fn x => (f (x+h) - f (x-h)) / (2.0 * h)
```

It is assigned the polymorphic type shown below.

$$\mathsf{num}\ u_1 \rightarrow (\mathsf{num}\ u_1 \rightarrow \mathsf{num}\ u_2) \rightarrow (\mathsf{num}\ u_1 \rightarrow \mathsf{num}\ u_2u_1^{-1})$$

An interesting extension to the system is the provision of different systems of units and the automatic insertion of conversions between them. These are formalised as additional equations in E which represent *equivalences* between base units in different systems (for example, one kilogram is equivalent to 2.2

pounds). The type inference algorithm and unification procedure must then be modified so that suitable coercions are inserted into the program.

A number of other type systems incorporate equational theories and possess inference algorithms based on equational unification: examples include record types [7] and isomorphisms between data representations [9]. The (unitary) theory of Boolean rings holds great potential too [5].

References

[1] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[2] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. Technical Report 561, Université de Paris-Sud, 1990.

[3] A. J. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, 1994.

[4] A. J. Kennedy. *Programming Languages and Dimensions.* PhD thesis, Computer Laboratory, University of Cambridge, 1995. Available as Technical Report No. 391.

[5] A. J. Kennedy. Type inference and equational theories. Submitted for publication, 1996.

[6] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[7] D. Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, INRIA Rocquencourt, May 1991.

[8] J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7(3/4):207–274, 1989.

[9] S. R. Thatte. Coercive type isomorphism. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 29–49. Springer-Verlag, 1991.

Higher-Order Matching and Tree automata An abstract of preliminary work

Hubert Comon
LRI, Bât 490
Univ. Paris-Sud
91405 Orsay cedex
France
Hubert.Comon@lri.fr

May 24, 1996

This small note reports on a very preliminary work on higher-order matching. There might be typos and unprecise points. We will start by a description of the problem, followed by a sketch of our solution.

Types

The language $\mathcal{L}$ of *simple types* is the smallest set containing a type constant o and such that for any $\sigma, \tau \in \mathcal{L}$, $\sigma \rightarrow \tau \in \mathcal{L}$. As usual, a type $\sigma \rightarrow (\sigma' \rightarrow \sigma'')$ is also written $\sigma, \sigma' \rightarrow \sigma''$; from now on, we assume this conversion from the former to the latter. (Then any type distinct from o can be written $\sigma_1, \ldots, \sigma_n \rightarrow o$)

The *order* of a type $\sigma \in \mathcal{L}$ is defined by: $O(o) = 1$ and $O(\sigma_1, \ldots, \sigma_n \rightarrow o) = 1 + \max(O(\sigma_1), \ldots, O(\sigma_n))$

Terms

We consider the simply typed lambda-calculus: the set of terms $\mathcal{T}$ is the smallest set containing variables (of a type $\tau \in \mathcal{L}$) and a set of typed constants (there maybe an arbitrary set of constants for each type) and closed under abstraction and application: if t is a term of type τ and x is a variable of type σ, then $\lambda x.t$ is a term of type $\sigma \rightarrow \tau$. If s is a term of type $\sigma_1, \ldots, \sigma_n \rightarrow \tau$ and $t_1, \ldots, t_n$ are terms of types $\sigma_1, \ldots, \sigma_n$ respectively, then $s(t_1, \ldots, t_n)$ is a term of type τ.

The *order* of a term t is the maximum order of the types of all its subterms.

β-reduction is terminating on $\mathcal{T}$. We write $s\downarrow$ the unique normal form of s w.r.t. β-reduction. $s =_\beta t$ when $s\downarrow = t\downarrow$

Higher-order matching problems

A higher-order matching problem is an equation $s = t$ between two terms $s, t \in \mathcal{T}$ such that t does not contain any free variable. s, t may be assumed β-irreducible without loss of generality.

The *order* of a (higher-order) matching problem is the maximum order of s and t.

A *solution* of a matching problem $s = t$ is a term assignment σ to the free variables of s such that $s\sigma =_\beta t$. We may also restrict ourselves to irreducible assignments σ without loss of generality.

First order matching problems have either no solution or a unique solution. Second-order matching problems have always finitely many solutions up to α-conversion (an algorithm can be derived from Huet's thesis). Third-order matching problems may have infinitely many solutions, but they are still decidable (As shown by G. Dowek in 93). For example the problem

$$x(\lambda y.y) = a$$

where a is a constant of order 1, has the following solutions:

$$x = \lambda z.z^n(a)$$

where n is any non-negative integer.

Fourth order matching is also decidable, as shown by V. Padovani in 1994. However, the problem of deciding whether an order n matching problem has a solution or not is open for $n > 4$.

Dual Interpolation problems

Dual interpolation problems are Boolean combinations of the following particular higher-order matching problems:

$$x(s_1, \ldots, s_n) = t$$

where $s_1, \ldots, s_n, t$ are ground terms (they do not contain free variables).

V. Padovani has shown in 1994 that the general nth order matching problems reduce to nth order dual interpolation problems.

In what follows, we concentrate on a single matching problem of the above form and consider the question of recognizability (by a finite tree automaton) of the set of u such that $\{x \mapsto u\}$ is a solution. If such a set of solutions is effectively recognizable, then we solve the higher-order matching problem since a Boolean combination of recognizable tree languages is again an (effectively computable) recognizable tree language.

Recognizability of the set of solutions

The following result does not solve the general problem but sheds new light on the decidability at order 4:

proposition 1 *The set of solutions of a problem $x(s_1, \ldots, s_n) = t$ where x, the only free variable has only one occurrence, is recognizable for matching problems of order $n \leq 4$.*

Sketch of the proof:

We build an automaton $\mathcal{A}$ as follows:

the set of states Q is the smallest set closed under subterm and containing:

- t
- The solutions of $y(r_1, \ldots, r_k) = u$ (up to α-conversion) where $r_1, \ldots, r_k, u$ are first-order subterms of t (free variables of these terms being considered as constants) or the special constant $\Box_o$.

  All these solutions are assumed to contain distinct bound variables.
- the type of x

For convenience, we will write q_t (or q_τ) instead of t (or τ) when they are considered as states.

Final states : There is only one final state q_t

The set of rules (which maybe infinite because the alphabet is infinite) consists of

- The rules $c(q_{t_1}, \ldots, q_{t_n}) \to q_{c(t_1,\ldots,t_n)}$ for each $c(t_1, \ldots, t_n) \in Q$ and c is either a constant or a variable or a binder.
- The rules $x_i(q_{t_1}, \ldots, q_{t_n}) \to q_u$ if $s_i(t_1, \ldots, t_m) \downarrow = u \in Q$, $x_1, \ldots, x_n$ are variables which occur nowhere else. (If t_i is a type τ, then t_i is replaced with the special constant $\Box_\tau$ in the reduction).
- Rules such that in state q_τ where τ is a type, we accept all terms of type τ (up to α-conversion)

We show that $\mathcal{A}$ accepts the set of terms u with free variables $x_1, \ldots, x_n$ such that $\lambda x_1 \ldots \lambda x_n.u$ is a solution of $x(s_1, \ldots, s_n) = t$.

It is not difficult to see that each term which is accepted by the automaton is also a solution.

Conversely, we show by induction on the pair (number of reduction steps, size of u) that, for all $q_u \in Q$ such that $u \in \mathcal{T}$,

$$v\sigma \downarrow = u \quad \Rightarrow \quad v \text{ is accepted in } q_u$$

where

$$\begin{cases} \sigma = \{x_1 \mapsto s_1; \ldots; x_n \mapsto s_n\} \\ Var(v) \subseteq \{x_1, \ldots, x_n\} \end{cases}$$

There are two cases: if $v = c(v_1, \ldots, v_m)$ where $c \neq x_i$, then we use the induction hypothesis, the closure of Q by subterm and the definition of the set of transition rules. If $v = x_i(v_1, \ldots, v_m)$, we consider an index j such that replacing the jth argument of $s_i(v_1\sigma, \ldots, v_m\sigma)$ with a dummy constant does not yield u as a normal form. We show that v_j is accepted in a state q_{u_j} such that $v_j\sigma \downarrow = u_j$. For, we consider a particular reduction sequence from which we derive $(v_j\sigma) \downarrow (r_1, \ldots, r_k) = u'$ where u' is a subterm of u.

Now, we can use the order hypothesis: we can see that, in case x is of order at most 4, $r_1, \ldots, r_k$ are of order 1. Then for each i, either r_i is a subterm of t, or is irrelevant in the reduction sequence. Hence $(v_j\sigma) \downarrow$ is accepted either in $q_{u'}$ or in some q_{u_j} where u_j is a solution of $y(r_1, \ldots, r_k) = u'$.

Finally, we use the induction hypothesis: v_j is accepted in q_{u_j}. We can repeat the argument for all indices j on which the reduction depends and, by construction of the automaton, $x_i(v_1, \ldots, v_m)$ is accepted in q_u.

Extensions

We are currently extending this technique to arbitrary higher-order matching problems. We believe again that the set of solutions of dual interpolation problems is still recognizable. This would also imply that the set of solutions of general higher-order matching problems.

Higher-Order Equational Unification via Explicit Substitutions

Claude Kirchner, Christophe Ringeissen

INRIA-Lorraine & CRIN-CNRS
Technopule de Nancy-Brabois — Campus Scientifique
615, rue du Jardin Botanique
BP 101, 54602 Villers-lès-Nancy Cedex France
e-mail: {Claude.Kirchner, Christophe.Ringeissen}@loria.fr

Higher-order E-unification is equational unification with respect to the equivalence relation $=_{\beta\eta\cup E}$ generated by $\beta\eta$-conversion and an arbitrary first-order equational theory E. We present how to perform higher-order E-unification thanks to the use of explicit substitutions and the related first-order rewrite system $\lambda\sigma$. The theory $\lambda\sigma E$ of interest is defined as the combined equational theory $=_{\lambda\sigma(E)\cup E}$ where $\lambda\sigma(E)$ is a $\lambda\sigma$-calculus integrating the first-order equational theory E and its function symbols. It is a non-disjoint combination of first-order equational theories and so we cannot reuse the well-known techniques developed for combining unification algorithm when signatures of the related theories are built over disjoint sets of function symbols.

We design a complete $\lambda\sigma E$-unification procedure. The chosen approach is to slightly modify the simple algorithm due to G. Dowek, T. Hardin and C. Kirchner for $\lambda\sigma$. This leads to few additional transformation rules for dealing with E and with the interaction between E and $\lambda\sigma$. For sake of simplicity, we assume that E is regular (left and right hand sides of axioms have the same variables) and collapse-free (there is no variable as left or right hand sides of axioms). But this could be generalized to arbitrary theories E at the cost of more complicated rules and more sophisticated E-unification algorithms.

The unification procedure may be viewed as a set of transformation rules together with a given strategy. The application of rules mainly depends on the top-symbols of s and t in an equation $s =^?_{\lambda\sigma E} t$:

1. If these top-symbols are constructors in $\lambda\sigma$ (roughly speaking, like λ and de Bruijn indices), then we apply the decomposition rules developed for $\lambda\sigma$ and which are still correct in this context.

2. If these top-symbols are function symbols in E, then we use the well-known notion of variable-abstraction to purify the equation. The pure equation will be solved thanks to the E-unification algorithm.

3. If these top-symbols are respectively a constructor in $\lambda\sigma$ and a function

symbol in E, there is a "theory clash" and the process fails since E is assumed to be collapse-free.

4. Otherwise, members of equations are normalized thanks to the weakly normalizing rewrite system $\lambda\sigma(E)$ in order to reach one of the different forms considered in the rules. There exist also explosion rules which are aimed to perform a step towards a solution.

The interest of this unification procedure lies in the result that a higher-order E-unification problem can be translated into a first-order $\lambda\sigma E$-unification problem and solutions of the latter remain in the image of the translation. Hence, we show how to reduce higher-order E-unification into first-order unification.

Unification via Explicit Substitutions: The Case of Higher-Order Patterns

Gilles Dowek Thérèse Hardin Claude Kirchner
Frank Pfenning

Following the general method and related completeness results on using explicit substitutions to perform higher-order unification proposed in [Dowek, Hardin, Kirchner; LICS 1995], we investigate the case of higher-order patterns as introduced by Miller. We show that our general algorithm specializes in a very convenient way to patterns. We also sketch an efficient implementation of the abstract algorithm and its generalization to constraint simplification, which has yielded good experimental results at the core of a higher-order constraint logic programming language.

Undecidability of the word problem in the union of theories sharing constructors

Extended Abstract

E. Domenjoud
Crin/Cnrs & Inria-Lorraine
BP 239 F-54506 Vandœuvre-lès-Nancy Cedex
France
Eric.Domenjoud@loria.fr

In [1], E. Domenjoud, F. Klay and Ch. Ringeissen established modularity results for unification, matching, and the word problem in the union of equational theories sharing constructors (with a suitable definition of the constructors). For unification, counter-example were given which show that no general combination algorithm exists if any condition of the modularity theorem is removed. We present here a similar counter-example for the word problem. It shows that weakening the conditions of the modularity theorem may lead to an undecidable word problem in the union. The counter-example is very general in the sense that the theories we consider are simple and linear.

This modularity theorem is as follows:

Theorem 1 (Domenjoud, Klay and Ringeissen 1994 [1]) *Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two equational theories sharing only constructors. If for each $i = 1, 2$:*

1. *the word problem is decidable modulo $\mathcal{E}_i$;*
2. *for each shared constructor h and any term t, $\exists t_1, \ldots, t_n,\ t =_{\mathcal{E}_i} h(t_1, \ldots, t_n)$ is decidable.*

then the word problem modulo $\mathcal{E}_1 \cup \mathcal{E}_2$ is decidable and for each shared constructor h and any term t, $\exists t_1, \ldots, t_n,\ t =_{\mathcal{E}_1 \cup \mathcal{E}_2} h(t_1, \ldots, t_n)$ is decidable.

The counter example we present shows that we may not simply remove the second condition of this theorem. This means that the word problem alone is not modular for theories sharing constructors.

The idea to prove this result is to build an equational theory $\mathcal{E}$ encoding a procedure which searches for the smallest solution of some semi-decidable problem P. A term of the form $search(P)$ will be $\mathcal{E}$-equal to $sat(P, x_0)$ if and only if x_0 is the smallest solution of P in some ordering. Searching for the smallest solution of P will ensure that sat is a constructor. Provided that

P and its solutions may be encoded using only constructors, considering the theory $\mathcal{E}'$ obtained by renaming *search* to *search'*, $search(P)$ and $search'(P)$ will be equal modulo $\mathcal{E} \cup \mathcal{E}'$ if and only if P has a solution. The trick is that it will be possible to decide $search(P) =_{\mathcal{E}} sat(P, x_0)$ since both P and x_0 are given. But to decide $search(P) =_{\mathcal{E}\cup\mathcal{E}'} search'(P)$, we have to decide whether x_0 exists, which is impossible.

The semi-decidable problem we consider is the Post Correspondence Problem (PCP) [2].

Definition 1.1 (Post Correspondence Problem (PCP)) *Let $\mathcal{A}$ and $\mathcal{C}$ be fixed finite alphabets. The Post Correspondence Problem consists in deciding for each pair of morphisms φ and φ' from $\mathcal{A}^*$ to $\mathcal{C}^*$ whether there exists a non-empty $\alpha \in \mathcal{A}^+$ such that $\varphi(\alpha) = \varphi'(\alpha)$.*

Theorem 2 (Post 1946 [2]) *The Post Correspondence Problem is undecidable in general.*

PCP is obviously decidable if $|\mathcal{C}| = 1$ and for $|\mathcal{C}| \geq 2$ the following results are known.

Theorem 3

- *if $|\mathcal{A}| \leq 2$ then PCP is decidable.*
- *if $|\mathcal{A}| \geq 9$ then PCP is undecidable.*

For $3 \leq |\mathcal{A}| \leq 8$ the problem is still open.

In the sequel, $\mathcal{A} = \{a_1, \ldots, a_n\}$ and $\mathcal{C} = \{c_1, \ldots, c_m\}$ are fixed disjoint sets of unary function symbols. We take $\mathcal{A}$ and $\mathcal{C}$ large enough to get the undecidability of PCP. From what precedes, it suffices to take $n \geq 9$ and $m \geq 2$. φ, as an argument of a function symbol, always stands for a sequence of n terms $\varphi_1, \ldots, \varphi_n$, and $\varphi[t]_i$ stands for a sequence of n terms the i^{th} of which is t.

We consider the theory $\mathcal{E}$ defined by the set of axioms given in the table below and the (non-terminating) rewriting system $\mathcal{R}$ obtained by orienting all axioms of $\mathcal{E}$ from left to right.

0		$Post(\varphi, \varphi') = apply(\varphi, \varphi', \perp, \perp, a_1\perp, \perp, \perp, \perp)$
1	$\forall i, j$	$apply(\varphi[c_j w]_i, \varphi', z, z', a_i x, y, u, u') = apply(\varphi[w]_i, \varphi', c_j z, z', a_i x, y, c_j u, u')$
2	$\forall i$	$apply(\varphi[\perp]_i, \varphi'[c_k w']_i, z, z', a_i x, y, u, u') = apply(\varphi[\perp]_i, \varphi'[w']_i, z, c_k z', a_i x, y, u, c_k u')$
3	$\forall i$	$apply(\varphi[\perp]_i, \varphi'[\perp]_i, z, z', a_i x, y, u, u') = restore(\varphi[\perp]_i, \varphi'[\perp]_i, z, z', a_i x, y, u, u')$
4	$\forall i, j$	$restore(\varphi[w]_i, \varphi', c_j z, z', a_i x, y, u, u') = restore(\varphi[c_j w]_i, \varphi', z, z', a_i x, y, u, u')$
5	$\forall i, j$	$restore(\varphi, \varphi'[w']_i, \perp, c_k z', a_i x, y, u, u') = restore(\varphi, \varphi'[c_k w']_i, \perp, z', a_i x, y, u, u')$
6	$\forall i, j$	$restore(\varphi, \varphi', \perp, \perp, a_i x, y, u, u') = apply(\varphi, \varphi', \perp, \perp, x, a_i y, u, u')$
7	$\forall j$	$apply(\varphi, \varphi', \perp, \perp, \perp, y, c_j u, c_j u') = apply(\varphi, \varphi', \perp, \perp, \perp, y, u, u')$
8		$apply(\varphi, \varphi', \perp, \perp, \perp, y, \perp, \perp) = sat(\varphi, \varphi', y)$
9	$\forall j \neq k$	$apply(\varphi, \varphi', \perp, \perp, \perp, y, c_j u, c_k u') = fail(\varphi, \varphi', y, u, u')$
10	$\forall j$	$apply(\varphi, \varphi', \perp, \perp, \perp, y, c_j u, \perp) = fail(\varphi, \varphi', y, u, \perp)$
11	$\forall k$	$apply(\varphi, \varphi', \perp, \perp, \perp, y, \perp, c_k u') = fail(\varphi, \varphi', y, \perp, u')$
12	$\forall j$	$fail(\varphi, \varphi', y, c_j u, u') = fail(\varphi, \varphi', y, u, u')$
13	$\forall k$	$fail(\varphi, \varphi', y, \perp, c_k u') = fail(\varphi, \varphi', y, \perp, u')$
14		$fail(\varphi, \varphi', y, \perp, \perp) = next(\varphi, \varphi', \perp, y)$
15		$next(\varphi, \varphi', x, a_n y) = next(\varphi, \varphi', a_1 x, y)$
16	$\forall i \neq n$	$next(\varphi, \varphi', x, a_i y) = shift(\varphi, \varphi', a_{i+1} x, y)$
17		$next(\varphi, \varphi', x, \perp) = shift(\varphi, \varphi', a_1 x, \perp)$
18	$\forall i$	$shift(\varphi, \varphi', x, a_i y) = shift(\varphi, \varphi', a_i x, y)$
19		$shift(\varphi, \varphi', x, \perp) = apply(\varphi, \varphi', \perp, \perp, x, \perp, \perp, \perp)$

The idea behind this set of axioms is the following. Consider a morphism φ from $\mathcal{A}^*$ to $\mathcal{C}^*$. We can encode φ as the sequence $\omega_1\perp, \ldots, \omega_n\perp$ where $\omega_i = \varphi(a_i)$. Now consider the term $Post(\varphi, \varphi')$ where φ and φ' are two morphisms encoded in this way. The rewriting system $\mathcal{R}$, when applied to this term, will apply φ and φ' to each non-empty word $\alpha \in \mathcal{A}^+$ in some increasing order, until a solution of $\varphi(\alpha) = \varphi'(\alpha)$ is found, provided one exists. Otherwise, it will never stop. Axiom 0 converts $Post(\varphi, \varphi')$ into $apply(\varphi, \varphi', \perp, \perp, a_1\perp, \perp, \perp, \perp)$ which starts the search for a solution of PCP and for each term $apply(\varphi, \varphi', \perp, \perp, \alpha\perp, \perp, \perp, \perp)$ with $\alpha \in \mathcal{A}^*$, $\mathcal{R}$ acts as follows.

- Axioms 1 through 6 apply φ and φ' to α. Axioms 1 and 2 perform the application to the first a_i in α, destructing the morphisms which are restored by axioms 4 and 5. Axiom 3 prepares for this restoration and axiom 6 prepares for the application to the next a_i in α. Eventually, we get $apply(\varphi, \varphi', \perp, \perp, \perp, \overline{\alpha}\perp, \overline{\varphi(\alpha)}\perp, \overline{\varphi'(\alpha)}\perp)$ where $\overline{\alpha}$, $\overline{\varphi(\alpha)}$ and $\overline{\varphi'(\alpha)}$ denote the words obtained by reversing α, $\varphi(\alpha)$ and $\varphi'(\alpha)$. Note that the application of the morphisms is destructive to avoid a copy which would make the theory non-linear.

- Axioms 7 destructs $\varphi(\alpha)$ and $\varphi'(\alpha)$ one symbol at once until either everything has been removed or a difference is found. If no difference is found then α is a solution of PCP. Axiom 8 stops the process and returns $sat(\varphi, \varphi', \overline{\alpha}\perp)$ where $\overline{\alpha}$ is the word obtained by reversing α.

- Axioms 9, 10 and 11 detect all possible differences between $\varphi(\alpha)$ and $\varphi'(\alpha)$.
- Axioms 12 and 13 destruct what remains from $\varphi(\alpha)$ and $\varphi'(\alpha)$ after a difference has been detected. This destruction is done one symbol at once to keep the theory regular. Eventually, we get $fail(\varphi, \varphi', \overline{\alpha}\bot, \bot, \bot)$
- Once $\varphi(\alpha)$ and $\varphi'(\alpha)$ have been completely destructed, axiom 14 prepares for the computation of the next α which is performed by axioms 15 through 18.
- At last, axiom 19 prepares for the application of φ and φ' to this next α and the whole process starts again with axiom 1.

This process may be depicted as follows

$$\begin{tikzcd}
Post(\varphi, \varphi') \arrow[r] & apply(\varphi, \varphi', \bot, \bot, a_1\bot, \bot, \bot, \bot)
\end{tikzcd}$$

$$\begin{tikzcd}
apply(\varphi, \varphi', \bot, \bot, \alpha\bot, \bot, \bot, \bot) \arrow[d] \\
apply(\varphi, \varphi', \bot, \bot, \bot, \overline{\alpha}\bot, \overline{\varphi(\alpha)}\bot, \overline{\varphi'(\alpha)}\bot) \arrow[u, bend left=90, "\alpha\leftarrow next(\alpha)", "\varphi(\alpha)=\varphi'(\alpha)"'] \arrow[d, "\varphi(\alpha)\neq\varphi'(\alpha)"'] \\
sat(\varphi, \varphi', \overline{\alpha}\bot)
\end{tikzcd}$$

We establish the following results for $\mathcal{E}$.

Proposition 3.1 *$\mathcal{E}$ is simple*

Theorem 4 *The word problem is decidable modulo $\mathcal{E}$.*

Idea of the proof. The proof works in three steps. We first consider the regular set of terms

$$\mathcal{T}^0 = apply((\mathcal{C}^*\bot)^n, (\mathcal{C}^*\bot)^n, \bot, \bot, \mathcal{A}^*\bot, \bot, \bot, \bot) \ \cup \ sat((\mathcal{C}^*\bot)^n, (\mathcal{C}^*\bot)^n, \mathcal{A}^*\bot)$$

and show that $\mathcal{E}$-equality is decidable in $\mathcal{T}^0$.

In a second time, we show that any outermost $\mathcal{R}$-derivation eventually yields a term of the form $C[t_1, \ldots, t_p]$ where each t_i belongs to $\mathcal{T}^0$ and $C[]$ is an $\mathcal{R}$-irreducible context of which no other subterm belongs to $\mathcal{T}^0$.

Finally we show that if $t = C[t_1, \ldots, t_p]$ and $t' = C'[t'_1, \ldots, t'_q]$ have this form then $t =_{\mathcal{E}} t'$ if and only if $C[] \equiv C'[]$ and for each i, $t_i =_{\mathcal{E}} t'_i$. From what precedes, these last equalities are decidable. □

Proposition 4.1 *Each symbol in $\mathcal{A} \cup \mathcal{C} \cup \{sat\}$ is a constructor of $\mathcal{E}$.*

Idea of the proof. We consider a RPO with a total precedence which makes any symbol in $\mathcal{A} \cup \mathcal{C} \cup \{sat\}$ smaller then any symbol in $\{Post, apply, restore, fail, next, shift\}$. We show that the completion of $\mathcal{E}$ with this ordering will never fail to orient an equation. Finally, we show that the symbols in $\mathcal{A} \cup \mathcal{C} \cup \{sat\}$ may not occur at the top of a left-hand side of a rule in the (infinite) completed system. □

We define now $\mathcal{E}'$ as the theory obtained by adding a prime to *Post*, *apply*, *restore*, *fail*, *next* and *shift* in $\mathcal{E}$. From the proposition 4.1, $\mathcal{E}$ and $\mathcal{E}'$ share only constructors. Finally, we get the theorem:

Theorem 5 *The word problem is undecidable modulo* $\mathcal{E} \cup \mathcal{E}'$

Idea of the proof. We show that if φ and φ' are the encoding of two morphisms from $\mathcal{A}^*$ to $\mathcal{C}^*$ then $Post(\varphi, \varphi') =_{\mathcal{E} \cup \mathcal{E}'} Post'(\varphi, \varphi')$ if and only if there exists a non-empty $\alpha \in \mathcal{A}^+$ such that $\varphi(\alpha) = \varphi'(\alpha)$. Hence the theorem. □

References

[1] E. Domenjoud, F. Klay, and Ch. Ringeissen. Combination techniques for non-disjoint equational theories. In Alan Bundy, editor, *Proceedings 12th International Conference on Automated Deduction, Nancy (France)*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 267–281. Springer-Verlag, June/July 1994.

[2] Emil L. Post. A Variant of Recursively Unsolvable Problem. *Bull. Am. Math. Soc.*, 52:264–268, 1946.

Combination of Compatible Reduction Orderings that are Total on Ground Terms
– Extended Abstract –

Franz Baader
LuFg Theoretical Computer Science, RWTH Aachen
Ahornstraße 55, 52074 Aachen, Germany
e-mail: baader@informatik.rwth-aachen.de

1 Introduction

Reduction orderings that are *total on ground terms* play an important rôle in many areas of automated deduction. For example, unfailing completion [4]—a variant of Knuth-Bendix completion that avoids failure due to incomparable critical pairs—presupposes such an ordering. In addition, using a reduction ordering that is total on ground terms, one can show that any finite set of ground equations has a decidable word problem [13, 20]. It is very easy to obtain such orderings. Indeed, many of the standard methods for constructing reduction orderings yield orderings that are total on ground terms: both Knuth-Bendix orderings [12] and lexicographic path orderings [10] are total on ground terms if they are based on a *total precedence ordering* on the set of function symbols.

Things become more complex if one is interested in reduction orderings that are *compatible with a given equational theory* E. Such orderings, which are, for example, used in rewriting modulo equational theories [8, 9, 2], can be seen as orderings on E-equivalence classes. E-compatible reduction orderings that are total on (E-equivalence classes of) ground terms can be employed for similar purposes as the usual reduction orderings that are total on ground terms. For example, let AC denote a theory that axiomatizes associativity and commutativity of several binary function symbols, where the signature may contain additional free function symbols. An AC-compatible reduction ordering that is total on ground terms can be used to show that for any finite set G of ground equations, the word problem is decidable for $AC \cup G$ [14, 15]. The first AC-compatible reduction ordering total on ground terms was described in [15]. It is based on a relatively complex polynomial interpretation in which the coefficients of the polynomials are again integer polynomials. Surprisingly, it turned out to be rather hard to construct AC-compatible reduction orderings by appropriately modifying standard orderings such as recursive path orderings [7]. The main idea underlying most proposals in this direction (e.g., [5, 3, 11, 6]) is to apply certain transformations such as flattening to the terms before comparing them with one of the standard path orderings. A major drawback of these approaches is that they impose rather strong restrictions on the precedence orderings on function symbols that may be used. One consequence of these restrictions is that the obtained AC-compatible orderings are not total on ground terms if more than one AC-symbol is present. This problem has finally been overcome in [18, 19], where an AC-compatible reduction ordering total on ground terms is defined that is based on a recursive path ordering (with status). In [17] it was shown that this approach can even be used to construct reduction orderings total on ground terms that are compatible with theories that axiomatize several associative, commutative, associative-commutative, and free symbols.

The present paper proposes a different way of attacking the problem of how to construct E-compatible orderings that are total on ground terms. It was motivated by the observation that it is very easy to define an AC-compatible reduction ordering total on ground terms if there is only one AC-symbol in the signature. Instead of directly defining an AC-compatible ordering total on ground terms for the case of more than one AC-symbol, we try to obtain such an ordering by combining the orderings that exist for the case of one AC-symbol.[1] To be more precise, assume that AC_1 axiomatizes associativity-commutativity of the symbol $+ \in \Sigma_1$ and that AC_2 axiomatizes associativity-commutativity of the symbol $* \in \Sigma_2$, where Σ_1 and Σ_2 are disjoint signatures that may contain additional free function symbols. For $i = 1, 2$, let $\succ_i$ be an AC_i-compatible reduction ordering that is total on the AC_i-equivalence classes of ground terms, i.e., $\succ_i$ can be seen as a total ordering on $\mathcal{T}(\Sigma_i, \emptyset)/_{=_{AC_i}}$. In order to define a reduction ordering that is total on $\mathcal{T}(\Sigma_1 \cup \Sigma_2, \emptyset)/_{=_{AC_1 \cup AC_2}}$ from the given orderings $\succ_1$ and $\succ_2$, we utilize the fact that this combined algebra can be represented as the amalgamated product of the single algebras $\mathcal{T}(\Sigma_i, \emptyset)/_{=_{AC_i}}$. This product was introduced in [1] in the context of combining unification algorithms. The construction of the amalgamated product represents the universe of $\mathcal{T}(\Sigma_1 \cup \Sigma_2, \emptyset)/_{=_{AC_1 \cup AC_2}}$ as a (possibly infinite) tower of layers. In principle, the combined ordering compares elements of the combined algebra first with respect to the layers they are in: elements in higher layers are larger than elements in lower ones. If two elements are in the same layer, then one of the original orderings ($\succ_1$ or $\succ_2$) is used to compare them.

This combination approach is, of course, not restricted to AC-theories. It can be used to combine arbitrary compatible reduction orderings that are total on ground terms, provided that the single theories are over disjoint signatures and satisfy some additional properties that will be introduced below. For example, theories that axiomatize associativity, commutativity, or associativity-commutativity of a binary function symbol satisfy these properties.

[1]This should not be confused with Rubio's approach for combining orderings on disjoint signatures [17]. To obtain his combined ordering, which extends given orderings on terms over the single signatures to an ordering on terms over the union of the signatures, he presupposes the existence of a compatible reduction ordering total on ground terms for the combined signature. In the present paper, the main goal is to show that such an ordering exists.

2 Compatible reduction orderings

Let Σ be a signature, and let $T(\Sigma, X)$ denote the terms over Σ with variables in X. A *reduction ordering* on $T(\Sigma, X)$ is a strict partial ordering $\succ$ that is Noetherian, stable under Σ-operations (i.e., $s \succ t$ implies $f(\ldots, s, \ldots) \succ f(\ldots, t, \ldots)$ for all $f \in \Sigma$), and stable under substitutions (i.e., $s \succ t$ implies $\sigma(s) \succ \sigma(t)$ for all Σ-substitutions σ). In the following, we will restrict our attention to reduction orderings on ground terms, which means that stability under substitutions can be dispensed with. However, the ground terms that will be considered may contain additional *free constants* from a set of constants C with $C \cap \Sigma = \emptyset$. By a slight abuse of notation, the set of these ground terms will be written as $T(\Sigma, C)$. The only difference between variables and free constants is the fact that constants cannot be replaced by substitutions, and thus it is possible to order them with a reduction ordering.

Let E be a set of identities over Σ, and let $=_E$ denote the equational theory induced by E. A reduction ordering $\succ$ is *E-compatible* iff $s \succ t$, $s =_E s'$, and $t =_E t'$ imply $s' \succ t'$. Thus, an E-compatible reduction ordering induces a well-defined ordering on the set of $=_E$-equivalence classes. For a set of free constants C, the E-free algebra with generators C, i.e., $\mathcal{T}(\Sigma, C)/_{=_E}$, will be denoted by $\langle C \rangle_{\Sigma,E}$. The set of free constants occurring in a term t is denoted by $C(t)$. We call a reduction ordering *total on $\langle C \rangle_{\Sigma,E}$* (or simply "total on ground terms," if the set of ground terms is clear from the context) iff it induces a total ordering on $\langle C \rangle_{\Sigma,E}$, i.e., iff for all $s, t \in T(\Sigma, C)$ we have $s \succ t$, or $s =_E t$, or $s \prec t$.

If E is a consistent equational theory (i.e., admits models of cardinality greater than 1), then we have $c \neq_E c'$ for every pair of distinct free constants $c, c' \in C$. Thus, an E-compatible reduction ordering total on $\langle C \rangle_{\Sigma,E}$ yields a total Noetherian ordering on C. We say that an E-compatible reduction ordering *extends* a total Noetherian ordering $>$ on C iff its restriction to C coincides with $>$. In the following, we consider *only consistent equational theories* (without mentioning it explicitly as a condition).

We close this second by stating some properties of equational theories and reduction orderings compatible with equational theories that will be important for the proof of our combination result:

Lemma 2.1 *1. If there exists a non-empty E-compatible reduction ordering, then E is a regular equational theory. In particular, we have for all terms $s, t \in T(\Sigma, C)$ that $s =_E t$ implies $C(s) = C(t)$.*

2. If there exists a non-empty E-compatible reduction ordering, then for any free constant $c \in C$ and term $t \in T(\Sigma, C)$ we can have $c =_E t$ only if c occurs exactly once in t.

3. If $\succ$ is an E-compatible reduction ordering total on $\langle C \rangle_{\Sigma,E}$, then $c \in C(t)$ for a free constant $c \in C$ and a term $t \neq_E c$ implies $t \succ c$.

4. *Let $\succ$ be an E-compatible reduction ordering total on $\langle C\rangle_{\Sigma,E}$, and assume that* $0 \in \Sigma$ *is a signature constant and* $c \in C$ *is a free constant. If there exists a term* s containing 0 *such that* $s =_E c$*, then* 0 *is the smallest element of* $\langle C\rangle_{\Sigma,E}$ *with respect to* $\succ$.

3 Combination of orderings

In principle, we want to solve the following combination problem: Let Σ_1, Σ_2 be disjoint signatures and E_1, E_2 be equational theories over the respective signature. Assume that, for $i = 1, 2$ and any set C of free constants, there exists an E_i-compatible reduction ordering $\succ_i$ that is total on $\langle C\rangle_{\Sigma_i,E_i}$. Can the orderings $\succ_1, \succ_2$ be used to construct an $(E_1 \cup E_2)$-compatible reduction ordering that is total on $\langle C\rangle_{\Sigma_1\cup\Sigma_2,E_1\cup E_2}$?

The next example demonstrates that this is not always possible.

Example 3.1 Let $\Sigma_1 := \{+, 0\}$, $\Sigma_2 := \{*, 1\}$, $E_1 := \{x + 0 = x\}$, and $E_2 := \{x * 1 = x\}$. It is easy to see that there exist E_i-compatible reduction orderings $\succ_i$ that are total on $\langle C\rangle_{\Sigma_i,E_i}$. In fact, since any term in $T(\Sigma_1, C)$ is $=_{E_1}$-equivalent to a term in $T(\{+\}, C)$, and since $=_{E_1}$ is the syntactic equality on $T(\{+\}, C)$, one can simply take a lexicographic path ordering that is induced by a well-ordering of C. The same argument applies to E_2.

However, assume that $\succ$ is an $(E_1 \cup E_2)$-compatible reduction ordering total on $\langle C\rangle_{\Sigma_1\cup\Sigma_2,E_1\cup E_2}$. Obviously, we have $c + 0 =_{E_1\cup E_2} c$ and $c * 1 =_{E_1\cup E_2} c$. By Property 4 of Lemma 2.1, both 0 and 1 must be the smallest element in $\langle C\rangle_{\Sigma_1\cup\Sigma_2,E_1\cup E_2}$, which is a contradiction since $0 \neq_{E_1\cup E_2} 1$.

In our general combination result, this kind of problem is avoided by restricting the attention to theories whose signatures do not contain function symbols, i.e., the only constants that may occur are free constants.[2]

There is a second restriction that must hold for our method to apply. The orderings $\succ_1, \succ_2$ must satisfy the following constant dominance condition:

Definition 3.2 Let $\succ$ be an E-compatible reduction ordering total on $\langle C\rangle_{\Sigma,E}$. Then $\succ$ satisfies the *constant dominance condition (CDC)* iff for all $t \in T(\Sigma, C)$ and $c \in C$ such that $c \succ c'$ for all $c' \in C(t)$, we have $c \succ t$.

Intuitively, this means that large constants dominate terms containing only small constants. An arbitrary E-compatible reduction ordering total on ground terms need not satisfy this property. For certain equational theories, however, the existence of an arbitrary E-reduction ordering total on ground terms implies the existence of such an ordering that also satisfies the CDC. Let C be a countably infinite set of free constants. For a term $t \in T(\Sigma, C)$ and a free

[2]Actually, it would be sufficient to apply this restriction to one of the two theories to be combined.

constant $c \in C$, let $|t|_c$ denote the number of occurrences of c in t. We say that the equational theory E is *strongly regular* iff $s =_E t$ implies $|s|_c = |t|_c$ for all terms $s, t \in T(\Sigma, C)$ and free constants c.

Lemma 3.3 *Let E be strongly regular. If there exists an E-compatible reduction ordering total on $\langle C\rangle_{\Sigma,E}$, then there also exists such an ordering that additionally satisfies the CDC.*

For example, theories axiomatizing commutativity, associativity, or associativity-commutativity of a binary function symbol are obviously strongly regular.

Our method for combining compatible reduction orderings depends on the representation of $\langle C\rangle_{\Sigma_1\cup\Sigma_2,E_1\cup E_2}$ as the free amalgamated product of $\langle C\rangle_{\Sigma_1,E_1}$ and $\langle C\rangle_{\Sigma_2,E_2}$, as introduced in [1].[3]

The free amalgamated product

The free amalgamated product of $\langle C\rangle_{\Sigma_1,E_1}$ and $\langle C\rangle_{\Sigma_2,E_2}$ is defined using two ascending towers of the following form: We consider disjoint sets of free constants $C_\infty = \bigcup_{i=0}^{\infty} C_i$ and $D_\infty = \bigcup_{i=0}^{\infty} D_i$ such that $C_0 = C$. In addition, for $n \geq 0$, let A_n be the carrier set of $\langle\bigcup_{i=0}^{n} C_i\rangle_{\Sigma_1,E_1}$, and let B_{n+1} be the carrier set of $\langle\bigcup_{i=0}^{n} D_i\rangle_{\Sigma_2,E_2}$. The partitioning of C_∞ and D_∞ into the sets C_i and D_i is such that sets on corresponding floors of the double tower shown in Figure 1 have the same cardinality.

Thus, there are bijections $h_0 : A_0 \to D_0$, $g_1 : B_1 \setminus D_0 \to C_1$, and for all $n \geq 1$, bijections $h_n : A_n \setminus (A_{n-1} \cup C_n) \to D_n$ and $g_{n+1} : B_{n+1} \setminus (B_n \cup D_n) \to C_{n+1}$.

Let A_∞ be the carrier set of $\langle C_\infty\rangle_{\Sigma_1,E_1}$, i.e., the union of all set in the left tower, and let B_∞ be the carrier set of $\langle D_\infty\rangle_{\Sigma_2,E_2}$, i.e., the union of all set in the right tower. The above bijections can be used in the obvious way to define bijections

$$h_\infty := \bigcup_{i=0}^{\infty} h_i \cup g_{i+1}^{-1} : A_\infty \to B_\infty \quad \text{and} \quad g_\infty := \bigcup_{i=0}^{\infty} h_i^{-1} \cup g_{i+1} : B_\infty \to A_\infty.$$

By definition, A_∞ is equipped with a Σ_1-structure, and the bijections h_∞ and g_∞ can be used to carry the Σ_2-structure on B_∞ to A_∞ (see [1] for details). As shown in [1], the $(\Sigma_1 \cup \Sigma_2)$-algebra $\mathcal{A}_\infty$ with carrier set A_∞ that is obtained this way is isomorphic to $\langle C\rangle_{\Sigma_1\cup\Sigma_2,E_1\cup E_2}$.

An ordering on the free amalgamated product

As mentioned above, we assume that the signatures Σ_1 and Σ_2 do not contain constant symbols, i.e., the only constants are free constants. In addition, assume

[3]It should be noted, however, that we use a slightly modified construction, which is not as symmetric as the original one, but more easy to adapt to our purposes.

$$\begin{array}{cc}
\dots & \dots \\
 & \\
A_{n+1} \setminus (A_n \cup C_{n+1}) & D_{n+1} \\
C_{n+1} & B_{n+1} \setminus (B_n \cup D_n) \\
A_n \setminus (A_{n-1} \cup C_n) & D_n \\
C_n & B_n \setminus (B_{n-1} \cup D_{n-1}) \\
 & \\
\dots & \dots \\
 & \\
C_2 & B_2 \setminus (B_1 \cup D_1) \\
A_1 \setminus (A_0 \cup C_1) & D_1 \\
C_1 & B_1 \setminus D_0 \\
A_0 & D_0
\end{array}$$

Figure 1: The double tower of the amalgamation construction.

that, for $i = 1, 2$, there is a mechanism for constructing E_i-compatible reduction orderings that satisfies the following properties:

1. For any finite or countably infinite set of free constants C and any total Noetherian ordering $>$ on C, the mechanism yields an E_i-compatible reduction ordering $\succ^{(i)}_{C,>}$ that extends $>$, is total on $\langle C \rangle_{\Sigma_i, E_i}$, and satisfies the CDC.

2. The mechanism is monotone in the following sense: Let $C_1 \subseteq C_2$, let $>_1$ be a total Noetherian ordering on C_1, and let $>_2$ be a total Noetherian ordering on C_2 such that $>_1 \subseteq >_2$. Then $\succ^{(i)}_{C_1,>_1} \subseteq \succ^{(i)}_{C_2,>_2}$.

3. The mechanism is invariant under monotone renaming of free constants. To be more precise, let $>_1$ be a total Noetherian ordering on C_1, $>_2$ be a total Noetherian ordering on C_2, and let $\pi : C_1 \to C_2$ be an order isomorphism. Then $s \succ^{(i)}_{C_1,>_1} t$ implies $\pi(s) \succ^{(i)}_{C_2,>_2} \pi(t)$, where the terms $\pi(s), \pi(t)$ are obtained from s, t by replacing the free constants in these terms by their π-images.

Theorem 3.4 *Assume that Σ_1 and Σ_2 are disjoint signatures that do not contain constant symbols, and that, for $i = 1, 2$, there exist mechanisms for constructing E_i-compatible reduction orderings total on ground terms satisfying the three conditions from above.*

1. *Then there exists an $(E_1 \cup E_2)$-compatible reduction ordering that is total on $\langle C \rangle_{\Sigma_1 \cup \Sigma_2, E_1 \cup E_2}$.*

2. *If the word problem for E_i and the orderings $\succ^{(i)}_{C,>}$ are decidable for $i = 1, 2$, then the combined ordering is also decidable.*

Instead of giving a formal proof of the first part of the theorem (which would violate the page limit), we give an intuitive description of how this ordering looks like. Its definition depends on the representation of $\langle C\rangle_{\Sigma_1\cup\Sigma_2,E_1\cup E_2}$ as the free amalgamated product $\mathcal{A}_\infty$ of $\langle C\rangle_{\Sigma_1,E_1}$ and $\langle C\rangle_{\Sigma_2,E_2}$. Going from bottom to top, one simultaneously defines an ordering on A_∞ and B_∞ by induction. Elements that belong to different levels of one of the towers are compared according to their height in the tower. Elements in a level $A_n \setminus (A_{n-1} \cup C_n)$ are compared with respect to the E_1-compatible ordering on A_n obtained by the mechanism (assuming that the precedence ordering on $\bigcup_{i=0}^{n} C_i$ is already defined). Elements in a level C_n are ordered using the bijection $g_n : B_n \setminus (B_{n-1} \cup D_{n-1}) \to D_n$ (assuming that the ordering on $B_n \setminus (B_{n-1} \cup D_{n-1})$ is already defined). The right tower is treated analogously.

In this construction, the induction base is given by an arbitrary total Noetherian ordering on C. The combined ordering obtained this way depends on the set C and on the ordering on C used for starting the inductive construction. Thus, we again obtain a construction mechanism that transforms a given total Noetherian ordering on a set of free constants C into an $(E_1 \cup E_2)$-compatible reduction ordering that is total on $\langle C\rangle_{\Sigma_1\cup\Sigma_2,E_1\cup E_2}$. The combined ordering does not satisfy the CDC. However, if E_1 and E_2 are strongly regular, then so is $E_1 \cup E_2$. Thus, Lemma 3.3 can be used to modify the combined ordering into one satisfying the CDC. It can be shown that the mechanism satisfies the other properties required in Theorem 3.4. Consequently, the construction can be applied iteratedly, provided that the involved theories are strongly regular.

The decision procedure for the combined ordering depends on a method that is similar to the approach used to show that the word problem for $E_1 \cup E_2$ is decidable, provided that the word problems for the single theories E_1, E_2 are decidable (see, e.g., [16]).

4 Conclusion

The aim of this work was to develop a general approach for combining compatible orderings that are total on ground terms. The main motivation was that it is often relatively easy to design such orderings for "small" signatures and theories, whereas it is rather involved to give a direct definition of an appropriate ordering in the case of signatures that contain several symbols axiomatized by equational theories over disjoint subsets of the signature. As an example, we have mentioned the case of signatures containing free symbols and more than one AC-symbol.

The main restrictions that must hold for this combination approach to apply are

1. The signatures of the single theories must not contain constant symbols, i.e., the only available constants are free constants.

2. Both theories must admit compatible orderings total on ground terms that satisfy the constant dominance condition (CDC).

These restrictions seem to be not overly severe. In fact, we have shown by an example that a violation of the first condition may lead to cases where a compatible ordering total on ground terms does not exist for the combined theory. In addition, for strongly regular theories (such as associativity, commutativity, or associativity-commutativity of a binary function symbol), the existence of a compatible orderings total on ground terms implies the existence such an ordering that also satisfies the CDC.

A major drawback of the presented combination approach is that until now it does not yield a non-trivial ordering for terms with variables. Indeed, we have defined an ordering on $\langle C\rangle_{\Sigma_1\cup\Sigma_2,E_1\cup E_2}$, where the elements of C are treated as free *constants*. For an ordering on terms with variables, one must also have stability under substitution. For some application (e.g., the decision problem for ground equations modulo AC), having an ordering on ground terms is sufficient. For other applications where one works with terms containing variables (such as unfailing completion), this is not quite satisfactory. For example, for unfailing completion, using an ordering where all terms with variables are incomparable would mean that none of the identities can be oriented into a rule, and thus all of them must be used in both directions to compute critical pairs. Thus, an important open problem is to extend the combined ordering in a non-trivial way to an ordering on terms with variables. It might be that this makes additional restrictions on the theories necessary (such as requiring them to be collapse-free).

References

[1] F. Baader and K.U. Schulz. Combination of constraint solving techniques: An algebraic point of view. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Springer LNCS*, pages 352–366, Berlin, 1995.

[2] L. Bachmair. *Canonical Equational Proofs.* Birkhäuser, Boston, Basel, Berlin, 1991.

[3] L. Bachmair. Associative-commutative reduction orderings. *Information Processing Letters*, 43:21–27, 1992.

[4] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, chapter 1, pages 1–30. Academic Press, New York, 1989.

[5] L. Bachmair and D.A. Plaisted. Associative path orderings. In J. P. Jouannaud, editor, *Proceedings of the International Conference on Rewriting Techniques and Applications*, volume 202 of *Springer LNCS*, pages 241–254, Berlin-Heidelberg-New York, 1986.

[6] C. Delor and L. Puel. Extension of the associative path ordering to a chain of associative commutative symbols. In C. Kirchner, editor, *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications (Montreal, Canada)*, volume 690 of *Springer LNCS*, pages 389–404, Berlin, 1993.

[7] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[8] G. Peterson and M.E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:223–264, 1981.

[9] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15:1155–1196, 1984.

[10] S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Univ. of Illinois at Urbana-Champaign. Unpublished manuscript, 1980.

[11] D. Kapur, G. Sivakumar, and H. Zhang. A new method for proving termination of AC-rewrite systems. In *Proceedings of the Tenth International Conference of Foundations of Software Technology and Theoretical Computer Science*, volume 472 of *Springer LNCS*, pages 133–148, Berlin, 1990.

[12] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–267. Pergamon Press, Oxford, 1970.

[13] D. S. Lankford. Canonical inference. Memo ATP-32, Automatic Theorem Proving Project, University of Texas, Austin, TX, December 1975.

[14] C. Marche. On ground AC-completion. In R. Book, editor, *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications (Como, Italy)*, volume 488 of *Springer LNCS*, pages 411–422, Berlin, 1991.

[15] P. Narendran and M. Rusinowitch. Any ground associative-commutative theory has a finite canonical system. In R. Book, editor, *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications (Como, Italy)*, volume 488 of *Springer LNCS*, pages 423–433, Berlin, 1991.

[16] T. Nipkow. Combining matching algorithms: The regular case. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications (Chapel Hill, NC)*, volume 355 of *Springer LNCS*, pages 343–358, Berlin, 1989.

[17] A. Rubio. *Automated Deduction with Constrained Clauses.* PhD Thesis, Universita Politècnica de Catalunya, Barcelona, Spain, 1994.

[18] A. Rubio and R. Nieuwenhuis. A precedence-based total AC-compatible ordering. In C. Kirchner, editor, *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications (Montreal, Canada)*, volume 690 of *Springer LNCS*, pages 374–388, Berlin, 1993.

[19] A. Rubio and R. Nieuwenhuis. A total AC-compatible ordering based on RPO. *Theoretical Computer Science*, 142, 1995.

[20] W. Snyder. Efficient ground completion: An $O(nlogn)$ algorithm for generating reduced sets of ground rewrite rules equivalent to a set of ground equations E. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications (Chapel Hill, NC)*, volume 355 of *Springer LNCS*, pages 419–433, Berlin, 1989.

Combining Unification and Built-In Constraints (Abstract)

Farid Ajili Claude Kirchner
INRIA Lorraine & CRIN
615 rue du jardin botanique BP 101
54602 Villers-lès-Nancy Cedex, France
email: {Farid.Ajili,Claude.Kirchner}@loria.fr

In less than a decade, Deduction with Constraints (DwC) has opened a new universe in computer science. DwC can be viewed from two perspectives: one related to the automated deduction framework [17, 19], the other to the development and usage of programming languages based on logic [7, 14].

Within the automated deduction framework, constraints on the generic data structure "terms" have become a popular tool because they allow to express and encode strategies and to modularise deduction processes [19]. We call *symbolic*, constraints over terms [11]. There are plenty of symbolic constraint systems, some examples are unification (see [16] for a survey), disunification [10], ordering [5], membership [9, 13] and feature constraints [1]. The most well-known example is equational unification. Equational unification is nothing but solving equations between terms when the function symbols of the terms satisfy a certain equational theory.

Within programming languages based on logic, the purpose is to develop a class of programming languages, which incorporates the computational properties of a logical theory with the efficiency of constraint solving. In this setting, Constraint Logic Programming (CLP) [7, 8, 15], instance of the Constraint Programming paradigm, is an elegant generalisation of logic programming, aimed at replacing unification by the concept of constraint solving over a computation domain. Thus, CLP is a class of languages, which merges the computational properties of Horn-clause logic and efficient constraint solving over a given domain. J. Jaffar & J.-L. Lassez proposed in [14] a theoretical semantic, $CLP(\mathcal{D})$, explaining the meaning of a CLP program, which is parametrised by the computation domain $\mathcal{D}$ in which variables can have values. Thus, a taxonomy of CLP languages is to classify them according to their domains: one can have $CLP(\mathbb{R})$, $CLP(\mathbb{Q})$, $CLP(FD)$ and so forth. We call *built-in*, constraints over such mathematical domains [2].

The need for more complex combined constraints involving several primitive constraint languages is of prime interest in many application areas. A simple example of such a situation exists in using a combined theory of naturals and strings to express the simple property that a natural is divisible by 9 iff the sum of its digits is divisible by 9. This could be expressed in the theory of naturals alone but in a less natural way.

Combination techniques have been thoroughly investigated in the last decade for symbolic constraints [3, 6, 18, 20]. In the case of equational unification, the problem is stated as follows: given two unification algorithms in two (consistent) equational theories E_1 and E_2, how to find a unification algorithm for $E_1 \cup E_2$. M. Schmidt-Schauß solved the general problem for disjoint function symbols sets [20]. Some extensions of this result were considered: in [18] and [12], sharing of constants and constructors are respectively allowed.

In [4], F. Baader & K. Schulz have showed how to combine constraint solvers for two arbitrary "Simply Combinable" structures over disjoint signatures into a solver for their combined structure. In addition, many CLP dialects allow that different kinds of built-in constraints coexist and must be solved in appropriate domains. For example, the structure underlying Prolog III [8] allows "mixed" constraints on lists of rational trees, where some nodes can be lists or booleans and so forth.

This paper relies on the fact that a combination of symbolic and built-in constraint languages often makes it possible to express and tackle problems that none of these languages can overcome alone in a natural way. For example, J. Avenhaus & K. Becker [2] have provided an approach of how to enrich an equational specification with a built-in algebra and asserted that such an approach makes the programming language more powerful. So, we are considering a problem, which has many important practical applications, but which is in full generality, undecidable. This is why in this paper we are presenting a general framework that provides some tools to solve this problem in specific cases.

We consider the problem of combining unification constraints interpreted in a term quotient Δ-structure $\mathcal{A} = \mathcal{T}(\mathcal{F}, \mathcal{X})/_{=_E}$, on one side, with built-in constraints interpreted in a term generated Σ-structure $\mathcal{B}$ on the other side. Here, we assume that we are in presence of a set of functions $\tilde{\Pi}$ from $\mathcal{A}$ and $\mathcal{B}$. One can notice that, because of variables of $\mathcal{X}$, a non-trivial function $\tilde{\pi}$ in $\tilde{\Pi}$ has to be defined only over the variable-free (*i.e*, ground) sub-structure of $\mathcal{A}$. The main difficulty comes from the fact that if a variable of $\mathcal{X}$ appears both in a unification constraint and in a built-in constraint then it has not the same behaviour. To illustrate the point, assume that $\mathcal{A} = \mathcal{T}(\mathcal{F}, \mathcal{X})/_{=_E}$ is the quotient term algebra of terms, where $\mathcal{F} = \{a, f\}$ and $E = \{f(x_1, x_2) = f(x_2, x_1)\}$, let $\mathcal{B}$ be the usual structure of naturals. Suppose that $\tilde{\Pi}$ is containing a function computing the size (number of nodes) of a *ground* term and let π be a function symbol used to refer $\tilde{\pi}$ in the syntax. In the combined formula:

$$\phi := f(x, a) =_E f(a, x) \wedge \pi(x) + y_1 \leq 2y_2 + s(0)$$

x is intended to represent any element of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ in the unification sub-formula and only ground terms in the built-in sub-formula because one needs a natural value for $\pi(x)$ to decide whether the inequality holds in $\mathcal{B}$. Thus the valuations of x are not the same in the two situations.

This point motivates the *modular* approach that we adopt in this paper. The basic idea is to break a formula ϕ in the combined theory into three formulae:

a Δ-formula ϕ_{Δ}, a Σ-formula ϕ_{Σ} and a "heterogeneous" formula $\phi_{\mathcal{H}}$ containing at least a symbol π, which refers to some shared function $\tilde{\pi}$ in $\tilde{\Pi}$. On one side, the semantic interest of such an approach allows to define an *own interpretation* for $\phi_{\mathcal{H}}$, while those of "pure" formulae ϕ_{Δ} and ϕ_{Σ} are preserved. On the other side, it allows to "use the right tool for the job" and gives a mean to filter information throughout the three levels in a cooperative way.

We propose a canonical form, called *quasi-solved form*, for the mixed constraints in the case where only homomorphisms are allowed in $\tilde{\Pi}$. Such a *quasi-solved form* is incremental and provides a relatively weak satisfiability test.

References

[1] H. Ait-Kaci, A. Podelski, and G. Smolka. A feature constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1-2):263–283, 1994.

[2] J. Avenhaus and K. Becker. Operational specifications with built-in's. 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 187–198. Springer-Verlag, February 1994.

[3] Franz Baader and Klaus Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *Proceedings 11th International Conference on Automated Deduction, Saratoga Springs (N.Y., USA)*, pages 50–65, 1992.

[4] Franz Baader and Klaus U. Schulz. On the combination of symbolic constraints, solution domains, and constraint solvers. In Ugo Montanari and Francesca Rossi, editors, *Proceedings 1st International Conference on Principles and Practice of Constraint Programming, Cassis (France)*, volume 976 of *Lecture Notes in Computer Science*, pages 380–397. Springer-Verlag, September 1995.

[5] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proceedings 1st IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pages 346–357, June 1986.

[6] A. Boudet. Unification in a combination of equational theories: An efficient algorithm. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1990.

[7] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, 1990.

[8] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[9] H. Comon. Equational formulas in order-sorted algebras. In Paterson, editor, *Proceedings ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 674–688. Springer-Verlag, 1990.

[10] 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.

[11] Hubert Comon, Mehmet Dincbas, Jean-Pierre Jouannaud, and Claude Kirchner. A methodological view of constraint solving, 1995. In preparation.

[12] E. Domenjoud, F. Klay, and Ch. Ringeissen. Combination techniques for non-disjoint equational theories. In Alan Bundy, editor, *Proceedings 12th International Conference on Automated Deduction, Nancy (France)*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 267–281. Springer-Verlag, June/July 1994.

[13] C. Hintermeier, C. Kirchner, and H. Kirchner. Dynamically-typed computations for order-sorted equational presentations –extended abstract–. In S. Abiteboul and E. Shamir, editors, *Proc. 21st International Colloquium on Automata, Languages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 450–461. Springer-Verlag, 1994.

[14] J. Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium on Principles Of Programming Languages, Munich (Germany)*, pages 111–119, 1987.

[15] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[16] 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.

[17] 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.

[18] H. Kirchner and C. Ringeissen. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.

[19] Hélène Kirchner. On the use of constraints in automated deduction. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 128–146. Springer-Verlag, 1995.

[20] M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. In Claude Kirchner, editor, *Unification*, pages 217–266. Academic Press inc., London, 1990.

Optimizations for Combining Unification Algorithms

Extended Abstract

Jörn Richts

Theoretische Informatik, RWTH Aachen
D-52056 Aachen, Germany
richts@informatik.rwth-aachen.de

Unification in equational theories is a widely studied area. There has been much progress in algorithms for specific equational theories as well as in the combination of disjoint theories. The first general combination method was presented by M. Schmidt-Schauß [SS89] and a more efficient version of this algorithm was described by A. Boudet [Bou90]. Like them, most authors deal with algorithms for computing complete sets of unifiers but decision procedures gain more and more importance. An algorithm for combining decision procedures of disjoint equational theories was presented by Baader and Schulz in [BS92]. This algorithm leads to many interesting theoretical results but due to the large search space its direct implementation is useless for practical purposes.

When investigating possibilities of optimizations for this algorithm, one is confronted with several problems. In contrast to the case where complete sets of unifiers are computed, in the case of decision procedures there is little information the optimizations can be based on. In order to get over this drawback, we present an extended combination algorithm which tries to choose a decision deterministically by calling pre-tests specific for the theories used in the unification problem.

While running the algorithm for combining decision procedures several non-deterministic choices have to be made (see figure 1). Some variables have to be identified, each variable has to be indexed with one of the theories occuring in the unification problem, and the variables have to be ordered with a linear ordering. The theory indices determine which variables have to be treated as constants in Γ_E or Γ_F, respectively, i.e. in which part of the unification problem they must not be instantiated. The ordering induces linear constant restrictions (LCR) restricting the set of unifiers. For each constant these restrictions specify a set of variables which must not be mapped to a term containing this constant. The unification problem Γ_0 is solvable if and only if there is a pair (Γ'_E, Γ'_F) where Γ'_E and Γ'_F are solvable with their LCR.

Depending on the theories involved, some of these choices can be made deterministically; e.g. for a collapse-free theory E_i the equation $x \doteq t$ with $t \notin \mathcal{V}$ imposes that x has to be instantiated by E_i, i.e. $\mathrm{ind}(x) = E_i$; for a

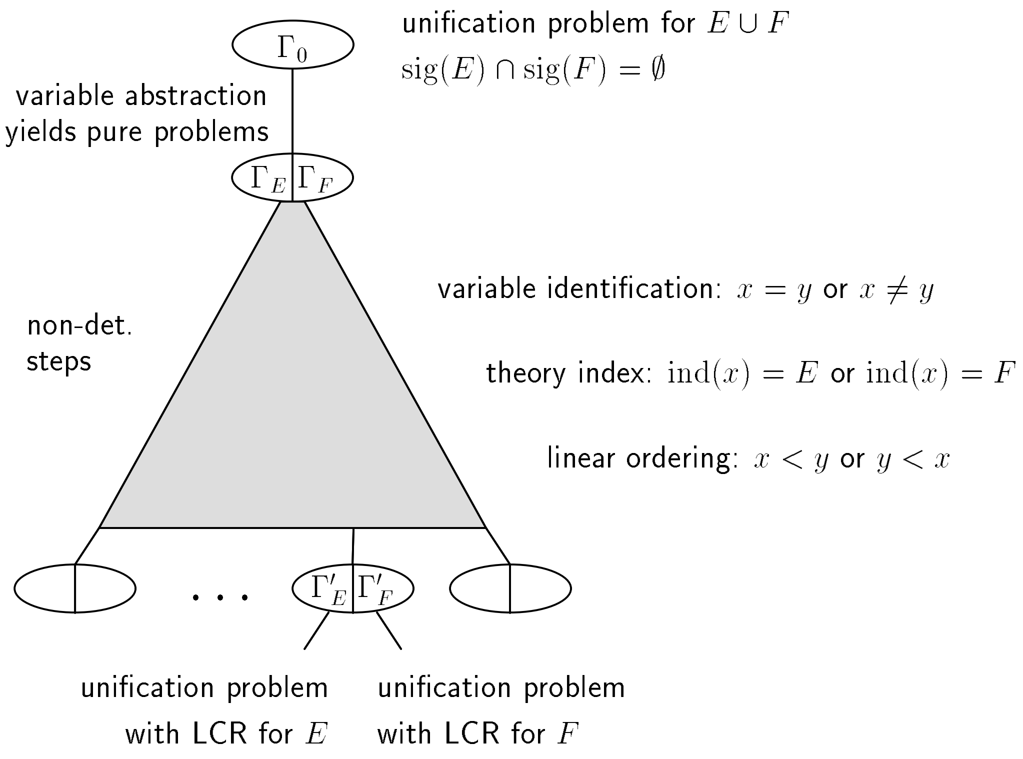

Γ_0
unification problem for $E \cup F$
$\mathrm{sig}(E) \cap \mathrm{sig}(F) = \emptyset$
variable abstraction yields pure problems
Γ_E Γ_F
variable identification: $x = y$ or $x \neq y$
non-det. steps
theory index: $\mathrm{ind}(x) = E$ or $\mathrm{ind}(x) = F$
linear ordering: $x < y$ or $y < x$
. . .
Γ'_E Γ'_F
unification problem with LCR for E
unification problem with LCR for F

Figure 1: The basic combination algorithm

regular theory E_j the equation $y \doteq t|_x$ with $\mathrm{ind}(x) \neq E_j$ imposes that y has to be mapped to a term containing the constant x, i.e. $x < y$. For specific theories more rules of this kind can be set up. As the example for regular theories shows, some choices (that have been made earlier in one theory) can imply new choices in another theory. This interplay between different theories proposes to use an algorithm where the choices made so far are propagated back and forth between the component algorithms of the theories involved. Starting with some initial information (like that in the example for collapse-free theories) each theory alternately computes new information. If this process comes to an end because no new information can be computed, a nondeterministic choice has to be made. After this choice the propagation process can be started again with the new information.

For most examples this combination algorithm has a search space which is significantly smaller than that of the original algorithm. But as a drawback new component algorithms for the theories occuring in the problem are needed which are capable of computing the desired information. Moreover, these algorithms should consider the special way in which they are called. Standard unification algorithms are "one shot" algorithms, they are started only once with all information they need given and compute final results. Component algorithms for our combination method must be able to cope with partial information and deliver something which is not necessarily the final result yet meaningful. More importantly, when receiving new information the algorithms should not restart computation from scratch but rather continue on the base of their prior internal states. Otherwise, the search space would be partially shifted from the combination algorithm to the component algorithms.

For the free theory, A, AC, and ACI such algorithms can be built by extending some well known methods. For example, the AC-algorithm is based on the minimal solutions of the homogeneous diophantine equations corresponding to the AC-unification problem. Information relevant for the combination method can be derived from these solutions.

The combination method and component algorithms for the free theory and AC have been implemented in COMMON-LISP using the KEIM toolkit [HKK+94]; component algorithms for A and ACI are currently implemented and tested. In the following we show some results of our optimizations. Table 1 gives an overview of the running time for some collections of AC-unification problems which come from the REVEAL theorem prover. Each collection contains all unification problems that have to be solved during the proof search or completion of the respective example. The first three examples are simple completions or proofs and the other three examples are from the REVEAL distribution. All examples except the first one contain two AC-symbols and several free symbols.

Name	Num.	A	B	C
Abelian group	29	3.6	5.0	54
Boolean ring	51	3.2	4.0	9.3
Boolean algebra	122	12	24.2	
`exboolston`	87	12	990	
`exgrobner`	1002	138	1800	
`exuqsl2`	404	112	>12h	

Table 1: Running time in seconds

The second column shows the number of unification problems in each example. Column A contains the running time of the algorithm with all optimizations, including the special algorithms for AC and the free theory. Column B shows the running time of the algorithm if the special AC-algorithm is substituted by a procedure which uses only the fact that AC is regular and collapse-free; in column C the running time of the algorithm without any optimizations can be seen. A missing value means that the calculation has been aborted after two hours. This table shows that our optimizations enable the combination method to be usable in practice.

References

[Bou90] Alexandre Boudet. Unification in a combination of equational theories: an efficient algorithm. In Mark E. Stickel, editor, *10th Conference on Automated Deduction*, Proceedings, pages 292–307, Kaiserslautern, Germany, 1990. Springer LNAI 449.

[BS92] Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In Deepak Kapur, editor, *Automated Deduction — CADE-11*, Proceedings, pages 50–65, Saratoga Springs, NY, USA, 1992. Springer LNAI 607.

[HKK+94] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. KEIM: A Toolkit for Automated Deduction. In Alan Bundy, editor, *Automated Deduction — CADE-12*, Proceedings, pages 807–810, Nancy, France, 1994. Springer LNAI 814.

[SS89] Manfred Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *Journal of Symbolic Computation*, 8(1,2):51–99, 1989.

Linear Completion for Equational Logic Programs

Gilles Richard and Frédéric Saubion
LIFO, Dépt. d'Informatique, Université d'Orléans, 45067 Orléans Cedex 02 (France),
e-mail: {richard, saubion}@lifo.univ-orleans.fr

Abstract

Introducing equality into standard Horn clauses leads to a programming paradigm known as Equational Logic Programming. On another hand, Linear Completion is a powerful mechanism for evaluation of logic programs. We propose here a scheme to extend this technique to the equational framework. Thus we provide a goal-oriented solving procedure, keeping the well-known advantages of Linear Completion : a reduced search space with a *loop avoiding* effect and the possibility to finitely *synthesize* an infinite set of answers.

1 Introduction

The combination of logic and functional programming arouse much interest since the beginning of the last decade and different techniques have been proposed to merge these two features (see [7] for a survey). Equations allow to represent functional programs while Horn clauses are suited for logic programs. The mix of these two formalisms leads to the notion of equational clause ; this combined language is known as Equational Logic Programming. Two main approaches can be distinguished to execute such languages. The first one ([9], [8]) consists in considering that equational logic programs are logic programs with a subjacent equality theory E distinct from the standard syntactic Clark equality. It is implicitly assumed that the set of clauses decomposes into disjoint sets of equations (defining E) and predicate headed clauses : thus, the solutions of equations are computed by a specific E-unification algorithm, which is then called by the Prolog procedure instead of a usual syntactic unification. Unfortunately, E-unification is a very hard problem ([15]) but there are restrictions ensuring a usable algorithm.

The second approach, such as [6], [12] and [2], considers clauses as conditional rewrite rules and applies evaluation mechanisms issued from term rewriting techniques. To deal with formulas involving only equality predicate, general predicates $p(\overline{x})$ are turned into equations $p(\overline{x}) = true$ where $true$ is a new constant symbol (in that way, a pure logic program is viewed as an equational one). Then, execution relies on performing superposition between heads of programs clauses or between the head of a program clause and a goal. These techniques mix bottom-up and top-down strategies and the efficiency of a concrete implementation is strongly related to the strategy for the choice the inference rule to apply.

In fact, there is an other way to adapt rewrite technique for evaluating pure logic programs. The idea is to transform a logic program into a set of rewrite rules and to execute them using a restricted version of KB completion called Linear Completion (*LC*). Linear Completion acts also as a completion mechanism but this completion is focused on a particular goal. For a given goal, *LC* computes a set of terminal rewrite rules. This set of answer rules can be considered as the specialization of the initial program for this particular goal : it defines the full set of solutions for a given query, even if this set is infinite.

The contribution of this paper is to propose an extension of *LC* including the equational framework. Equational Linear Completion (*ELC*) keeps the operational advantages of *LC* and provides a mechanism to execute equational logic programs focused on goal solving. The usual systems combine resolution (top-down) and completion (bottom-up) of the initial program. The completion part is required in order to compute new clauses that are equational consequences of the program and thus to insure completeness of the mechanism. The problem is that unnecessary clauses can be computed which are not involved in the resolution of the goal. For instance, if we consider the following program containing only two facts $a \simeq b$ and $a \simeq c$, given the goal $\leftarrow b \simeq c$, which is an equational consequence of the program, the initial set of clauses has to be completed by superposing $a \simeq b$ onto $a \simeq c$ in order to get the new fact $b \simeq c$. But if we add the fact $p(a)$ and if we want to prove $p(b)$, the mechanism will also complete the program, will generate the two atoms $p(b)$ and $p(c)$ while it suffices to prove $p(a) \wedge a \simeq b$ (which is logically equivalent). In fact, no completion is required in that case. Completion of the initial program can drastically degrade operational efficiency. *ELC* will be able to solve such a goal without requiring these bottom-up inferences. As in *LC*, for a given goal, *ELC* computes a set of terminal rewrite rules. This set of answer rules can be considered as the specialization of the initial program for this particular goal : it defines the full set of solutions for a given query, even if this set is infinite.

The following example, despite its simplicity, shows the different problems we want to tackle. Considering the following program :

1. $p(f(x)) \leftarrow p(x)$.
2. $p(g(x)) \leftarrow g(x) \simeq x$.
3. $g(a) \simeq a$.
4. $h(g(x)) \simeq a \leftarrow h(x) \simeq a$.

we want to compute the answers to the goal $\leftarrow p(x)$. The correct answers are a, the $f^n(a)$ and the $g^n(a)$, but they are obtained differently. The first clause defines recursively p and will cause an infinite computation. Since we do not affirm that $f(x) \simeq x$, the only way to give all the answers with a classical mechanism is to enumerate all the $f^n(a)$. The second clause define also an infinite set of answer but since we have $g(a) \simeq a$, it suffices to provide a (as usual we are only interested by the smallest answer w.r.t. the equational theory). At last, the classical mechanism performs completion over 3. and 4. and generate $h(a) \simeq a \leftarrow h(a) \simeq a$ which also causes termination problems. This is the behaviour of the usual methods designed for equational logic programs, based on completion techniques [6, 12, 2]. *ELC* leads to the following derivation tree :

$$\begin{tikzcd}
 & p(x) \to Ans(x) \arrow[dl, dash] \arrow[ddr, dash] & \\
Ans(g(x)) \to true\{g(x) \simeq x\} \arrow[d, dash] & & \\
Ans(g(a)) \to true \arrow[d, dash] & & Ans(f(x)), p(x) \to p(x) \arrow[d, dash] \\
Ans(a) \to true & & Ans(f(x)), Ans(x) \to Ans(x)
\end{tikzcd}$$

The philosophy of ELC is to generate from a given goal, a set of rules containing only the predicate Ans which defines the whole set of answers. Here we get two terminal rules: $Ans(a) \to true$ which defines a ground answer a and a formula $Ans(x), Ans(f(x)) \to Ans(x)$ whose meaning is "if x is an answer then $f(x)$ is still an answer". No bottom-up inference is needed in this derivation. ELC allows to validate the definition of predicate p by a finite calculus that defines as well the smallest ground answer and the representation of all the answers that are not reducible to this ground answer. ELC invokes completion only for the equational part of the program and only if nothing else can be done to compute the solutions. If we add the clause $p(x) \leftarrow p(k(x))$ to the previous program, this leads to a looping goal derivation $\leftarrow p(x), \leftarrow p(k(x)), \leftarrow p(k(k(x)))$ ... with the usual mechanisms. Using ELC this calculus is stopped thanks to a simplification by ancestor goal.

In section 2, some technical preliminaries are given and the transformation of equational logic programs into equational rewrite programs is defined. In section 3, we describe our mechanism by a transition system and we show some applications, pointing out its properties. In section 4, we show that the logical meanings of equational logic programs and equational rewrite programs are the same. We also prove that our mechanism is sound and complete w.r.t. this declarative meaning. In section 5, we explore some future works and we conclude.

2 Preliminaries

We assume the reader familiar with basic rewrite [5] and logic programming notions [11].

2.1 Equational Logic Programs

Given a set of variables $\mathcal{V}$ and a set of function symbols $\mathcal{F}$, We denote by $\mathcal{T}(\mathcal{V}, \mathcal{F})$ the set of first order terms build over the two previous sets. $\overline{x}$ (resp. $\overline{s}$ or $\overline{t}$) will denote a list of variables (resp. terms). A substitution σ is a finite domain mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{V}, \mathcal{F})$. Substitutions are extended by homomorphism over $\mathcal{T}(\mathcal{V}, \mathcal{F})$. We denote by $t_{|\omega}$ the subterm of t at occurrence ω (occurrences are classically defined by induction) and by $t[\omega \leftarrow t']$, the term equal to t except for the subterm at occurrence ω which has been replaced by t'. Given a set $\mathcal{P}$ of predicate symbols, *atoms* are predicates applied to terms. A special symbol $\simeq$ belonging to $\mathcal{P}$ will be used in infix notation : an atom $s \simeq t$ where s and t

are in $\mathcal{T}(\mathcal{V},\mathcal{F})$ will be called an *equation*. $\overline{s} \simeq \overline{t}$ is an abbreviation for the set of equations $s_1 \simeq t_1, \cdots s_n \simeq t_n$ if $\overline{s} = (s_1, \cdots, s_n)$ and $\overline{t} = (t_1, \cdots, t_n)$. We consider *definite equational logic clauses*: $A \leftarrow E_1, ..., E_n, B_1, ..., B_m$ (or $A \leftarrow E,\ B$ to abbreviate) where A is an atom, $E_1, ..., E_n$ are equations and $B_1, ..., B_m$ are non-equality atoms. A is called the *head* of the clause and $E_1, ..., E_n, B_1, ..., B_m$ the *body*. As usual, the intended meaning of such a clause is the universally quantified logical formula $B_1 \wedge \cdots \wedge B_n \wedge E_1 \wedge \cdots \wedge E_m \Rightarrow A$. A clause with an empty body is called a *fact*. An *equational logic program* is a finite set of clauses. An *equational logic goal* is a formula $\leftarrow E, B$ where E is a multiset of equations and B a multiset of atoms.

2.2 Equational Rewrite Programs

We want to translate equational logic programs into set of rewrite rules. Thus a simplification ordering $\prec$ is assumed on terms, allowing to orient the equalities $l \simeq r$. In the following, we suppose that each equation $l \simeq r$ is such that $l \not\prec r$. Since we handle non-equality atoms, we need an extension of this ordering to atoms. Thus we assume a precedence over predicate symbols such that $\simeq$ will be the biggest predicate symbol and *true* the least one. Doing that, we get a full ordering over the atoms and we shall consider its multiset extension.

Our transformation of an equational clause is based on the fact that $B \wedge E \Rightarrow A$ has the same informative content than $A \wedge B \wedge E \Leftrightarrow B \wedge E$ (i.e. the formula $(A \Leftarrow B \wedge E) \Leftrightarrow (A \wedge B \wedge E \Leftrightarrow B \wedge E)$ is a theorem in first order logic). Such an equivalence is denoted $A, B \rightarrow B\{E\}$ to separate the purely equational atoms from the logical ones and is considered, from now on, as a rewrite rule. If E is empty, we merely write $A, B \rightarrow B$.

An *equational rewrite program* is just a set of such rewrite rules and we give a transformation function which generates such a program from a given equational logic program.

Definition **1** *The transformation function ψ is a function whose input is an equational logic program P and output is an equational rewrite program $\psi(P)$. The rules of $\psi(P)$ are obtained in the following way :*

1. *A fact A is transformed into the rewrite rule $A \rightarrow true$. Such a rule is a* fact rule*.*

2. *A clause $A \leftarrow E, B$ is transformed into $A, B \rightarrow B\{E\}$. Such a rule is an* if rule*.*

where A denotes an atom or an equation, B a conjunction of atoms, E a conjunction of equations.

We can remark that two levels of rewrite equations are interwoven here: the equational theory over $\mathcal{T}(\mathcal{X},\mathcal{F})$ defined by $\simeq$ and the theory defined by the rewrite rules of the program over the atoms.

3 Operational Mechanism

The operational semantics defined in [2] and [12] are based on completion mechanisms. Proving a goal $\leftarrow E, B$ w.r.t. an equational program P consists in generating the empty clause from $P \cup \{\leftarrow E, B\}$ (i.e. proving the inconsistency of this set of clauses). Linear completion is a restricted completion mechanism focused on goal solving. We adapt here this technique to the equational framework. During the execution of our mechanism, substitutions will appear. Final substitutions constitute the answers to the initial query. Considering a substitution as a set of syntactic equalities allows us to deal with them as constraints (symbolic constraints at this step). Thus we extract the current substitution from the goal and we record it outside the goal as a constraint. This way will render easier the extension of our inference system to a non symbolic constraint context. Now, a rewrite rule has the following form :

$$L \rightarrow R\{E\}[\![\theta]\!]$$

Such a rule stands for $\theta(L) \rightarrow \theta(R)\{\theta(E)\}$. In the context of rewrite programs, we need to translate a logic goal into a rewrite rule. We add a special predicate symbol Ans not defined in the program and such that : $\forall p \in \mathcal{P}, true \prec Ans \prec p$. A *query rule* is a rule of the form $B \rightarrow Ans(\overline{x})\{E\}$ where $\overline{x}$ are the free variables occurring in B and E (Ans predicate has an arity equal to the number of free variables occurring in the initial goal).

3.1 The Inference System

As mentioned in the introduction, completion is needed to insure completeness of our mechanism. But we restrict these bottom-up computation steps to the equation headed clauses. If the theory defined by $\simeq$ is not confluent (for instance, $b \simeq c$ is an equational consequence of $a \simeq b$ and $a \simeq c$ but can not be treated by narrowing), we have to complete the program. Since this completion process can lead to infinite computations, the completed program is generated apart from the top-down goal solving process and anyway, these completion rules will be applied only when nothing else can be done. We define a special set of rules $\psi(P)^*$ which contains equational rewrite rules that are deduced from the initial program rules by the following standard inference rules :

Program Completion Rules PCR

Deduce

$$\frac{\psi(P)^*}{\psi(P)^* \cup \{t_1[\omega \leftarrow s_2] \simeq s_1, B_1, B_2 \rightarrow B_1, B_2\{E_1, E_2\}[\![\theta_1 \wedge \theta_2 \wedge \sigma]\!]\}}$$

if $t_1 \simeq s_1, B_1 \rightarrow B_1\{E_1\}[\![\theta_1]\!], t_2 \simeq s_2, B_2 \rightarrow B_2\{E_2\}[\![\theta_2]\!] \in \psi(P)^*$ and $\sigma\theta_1 t_1|_\omega = \sigma\theta_2 t_2$

Tautology deletion

$$\frac{\psi(P)^* \cup \{s \simeq t, B \rightarrow B\{E\}[\![\theta]\!]\}}{\psi(P)^*} \text{ if } \theta s = \theta t$$

Clearly, we start with $\psi(P)^* = \psi(P)$.
Now, we define the goal oriented part of our mechanism (top-down evaluation). Equational Linear Completion (ELC) is described as a pair (I, Σ) where I is a set of inference rules and Σ is a strategy defining a precedence over these inference rules. An inference rule $i \in I$ is presented as:

$$\frac{(g\,;M)}{(g'\,;M')}$$

where g and g' are goal rules and M and M' are set of goal rules. Since $M' = M \cup \{g\}$ for some inference rules, M represents the memory of the derivation in the sense that the mechanism stores in this set some of the previous goal rules which could be used to simplify the current goal.

Definition **2** *A* derivation *is a chain:*

$$(g_0\,;M_0) \rightsquigarrow_I \ldots \rightsquigarrow_I (g_n\,;M_n)$$

where each inference $\rightsquigarrow_I$ *is performed using a rule* $i \in I$ *according to the strategy* Σ. *A derivation stops if no more inference rule can be applied to the current computation state. We also use the notation* $(g\,;M)\vdash_I(g_n\,;M_n)$ *when there is a derivation from* $(g\,;M)$ *to* $(g_n\,;M_n)$ *(we generally omit the subscript* I*).*

Our set I of inference rules is divided into two parts POR and EOR, each operating for a specific task. From an implementation point of view, each mechanism can work separately in a concurrent computation. The first inference rules compute over the logic part of the goal.

Predicate Oriented Rules POR

Delete

$$\frac{(L \leftrightarrow R\{E\}[\![\theta]\!];\ M)}{(\emptyset;\ M)}$$

if $\theta L = \theta R$

Orient

$$\frac{(L \leftrightarrow R\{E\}[\![\theta]\!];\ M)}{(L \rightarrow R\{E\}[\![\theta]\!];\ M)}$$

if $\theta L \succ \theta R$

Predicate Resolution

$$\frac{(p(\overline{t}), L \rightarrow R\{E_2\}[\![\theta_1]\!];\ M)}{(B, L \leftrightarrow B, R\{E_1, E_2, \overline{s} \simeq \overline{t}\}[\![\theta_1 \wedge \theta_2]\!];\ M')}$$

if $p(\overline{s}), B \rightarrow B\{E_1\}[\![\theta_2]\!] \in \psi(P)$ and $M' = M \cup \{p(\overline{t}), L \rightarrow R\{E_2\}[\![\theta_2]\!]\}$

Simplify

$$\frac{(G_1, L \rightarrow R\{E_1\}[\![\theta_1]\!];\ M)}{(D, L \leftrightarrow R\{E_1\}[\![\theta_1 \wedge \sigma]\!];\ M)}$$

if $G_2 \rightarrow D\{E_2\}[\![\theta_2]\!] \in \psi(P) \cup M$ and $\exists \sigma, \sigma\theta_2 G_2 = \theta_1 G_1$ and $\sigma\theta_2 E_2 \subseteq \theta_1 E_1$

Answer Reduction

$$\frac{(Ans(\overline{t}), L \rightarrow R\{E\}[\![\theta_1]\!];\ M)}{(Ans(\overline{t}[\omega \leftarrow r]), B, L \leftrightarrow B, R\{E, E'\}[\![\theta_1 \wedge \theta_2 \wedge \sigma]\!];\ M)}$$

if $l \simeq r, B \rightarrow B\{E'\}]\!]\theta_2]\!] \in \psi(P)^*$ and $\sigma\theta_1\overline{t}_{|\omega} = \sigma\theta_2 l$ and $\theta_1\overline{t}_{|\omega}$ is not a variable

Delete allows to stop useless computation. *Orient* insures that the goal rule is oriented according to the simplification ordering. *Predicate Resolution* differs from the usual inference systems designed for equational logic programs since predicates are resolved as in constraint logic programming : the equality between the arguments of the goal predicate and the program rule predicate will be treated apart by the equational inference system described below. Substitutions are recorded as constraints and no equational unification is required. *Simplify* is a key feature of this mechanism : simplification of a current rule by one of its ancestors allows to avoid some loops. Since *Ans* predicate is not defined by the program rules, a rewrite rule containing only this predicate is a terminal rule as soon as the arguments are reduced : such a rule synthesi-

zes a set of answers. *Answer Reduction* is invoked in order to reduce answers w.r.t. the subjacent equational theory. This inference corresponds in fact to a bottom-up computation step usually performed between a predicate headed clause and equation. In our context, since we transform the initial program into a set of specialized rewrite rules that synthesizes the program for the given goal, this kind of operation can be simulated by this top-down inference.The next inference rules use narrowing to solve the equational part E of the goal (i.e. to solve equations such as $s \simeq t$).
Equational Oriented Rules EOR

Equational Deduction

$$\frac{(L \to R\{s \simeq t, E_1\}[\![\theta_1]\!];\ M)}{(L \to R\{s[\omega \leftarrow r] \simeq t, E_1, E_2\}[\![\theta_1 \wedge \theta_2 \wedge \sigma]\!];\ M)}$$

if $\sigma\theta_1 s_{|\omega} = \sigma\theta_2 l$ and $l \simeq r, B \to B\{E_2\}[\![\theta_2]\!] \in \psi(P)^*$ and $\theta_1 s \not\prec \theta_1 t$ and $\theta_1 s_{|\omega}$ is not a variable

Unification

$$\frac{(L \to R\{s \simeq t, E\}[\![\theta]\!];\ M)}{(L \to R\{E\}[\![\theta \wedge \sigma]\!];\ M)}$$

if $\sigma\theta s = \sigma\theta t$

Equational Deduction corresponds to a narrowing operation. *Unification* allows to transform equations into substitutions. We propose to invoke completion rules only when needed i.e. when no equational resolution step can be performed to solve an equation. The set $\psi(P)^*$ previously defined can be viewed as a blackboard since completion can be involved in different branches of the derivation. The equation to solve is delayed until a usable equational rewrite rule is generated in $\psi(P)^*$ while another branch can be explored.

We implicitly assume that, before going on into a derivation, each side of the current goal rule is fully simplified by the two meta-reduction rules :$X, X \to X$, $X, true \to X$. The strategy Σ is expressed as the following precedence $\gg$ over the set I and over the rules of PCR :
Simplify $\gg$ Delete $\gg$ Orient $\gg$ Predicate Resolution $\gg$ Answer reduction $\gg$ Unification $\gg$ Equational deduction $\gg$ Deduce $\gg$ Tautology deletion.
We see that we delay as much as possible equational resolution to focus on the logical part. A *selection rule* is a function that select an atom to be treated in the current goal rule. Here we choose the maximum atom of the right hand side of the goal. The philosophy of ELC is to transform the initial rewrite program and the associated query rule into a new set of rules containing only Ans predicate and substitutions as constraints, capturing the intended answers. So, the definition of a success goal rule follows :

Definition **3** *A rule g is a* success goal rule *if it is of one of the forms :*

- $Ans(\overline{t}) \rightarrow true[\![\theta]\!]$
- $Ans(\overline{t_1}), ..., Ans(\overline{t_n}) \leftrightarrow Ans(\overline{t_{n+1}}), ..., Ans(\overline{t_p})[\![\theta]\!]$

3.2 Pointing out Advantages

Equational Linear Completion mechanism benefits from the loop avoiding properties of Linear Completion. In [12], deletion rules are introduced that avoid some redundant completion steps. The problem is due to the fact that the completion mechanism generates tautologies of two types: $t \simeq t \leftarrow B$ and $A \leftarrow A, B$. This is highlighted by the following example (extracted from [12]):

$$\begin{array}{l} p(c,b,b). \\ b \simeq c \leftarrow p(c,c,b), p(c,b,c). \\ p(x,y,x) \leftarrow p(x,y,y). \\ p(x,x,y) \leftarrow p(x,y,y). \end{array}$$

The goal is $\leftarrow p(c,c,c)$. With the usual mechanisms, every inference over this set of clauses leads to a tautology or an existing clause. But these tautologies are necessary to prove the goal and may not be removed. With ELC, we do not even need to complete the initial set of clauses since there is only one clause with an equational head. Starting from the query $p(c,c,c) \rightarrow Ans$, we generate $Ans \rightarrow true$. No completion inference is needed to prove that goal and the unproductive inference are stopped using simplification and deletion.
Consider now the equational logic program:

1. $p(a)$.
2. $p(f(x)) \leftarrow p(x)$.
3. $f(x) \simeq a \leftarrow p(x)$.

For the query $\leftarrow p(x)$, the set of answer is only $\{a\}$ since each answer is of the form $f^n(a)$ and can be reduced to a with the clause 3. Usual evaluation mechanisms provide an infinite computation. In the introduction, the example has shown the ability to synthesize by a finite representation an infinite set of answers. Here the problem is quite different since there is an infinity of answers but which are all equationally equivalent. Considering our transformed program:

1. $p(a) \rightarrow true$.
2. $p(f(x)), p(x) \rightarrow p(x)$
3. $f(x) \simeq a, p(x) \rightarrow p(x)$

we get the following derivation tree:

$$\begin{tikzcd}
& p(x) \to \mathit{true} \arrow[dl, dash, "\text{Predicate Resolution with 1}"'] \arrow[dr, dash, "\text{Predicate Resolution with 2}"] & \\
\mathit{Ans}(a) \to \mathit{true} & & p(y), \mathit{Ans}(x) \to p(y)\{f(y) = x\} \arrow[d, dash, "\text{Simplify with G}"] \\
& & \mathit{Ans}(y), \mathit{Ans}(x) \to \mathit{Ans}(y)\{f(y) = x\} \arrow[d, dash, "\text{Equational Deduction with 3}"] \\
& & \mathit{Ans}(y), \mathit{Ans}(x), p(a) \to \mathit{Ans}(y), p(a)\{x = a\} \arrow[d, dash, "\text{Unification and Simplify with 1}"] \\
& & \mathit{Ans}(y), \mathit{Ans}(x) \to \mathit{Ans}(y)\llbracket x = a \rrbracket
\end{tikzcd}$$

These examples highlight the loop avoiding and synthesis aspects of our mechanism.

4 Declarative Semantics

Logic programming with equality has a clearly defined semantics ([8], [6], [2]) we briefly recall here. The reader is supposed to be familiar with the interpretation, model and logical consequence notions. In [8, 9], it is assumed that the set of clauses in an equational program P decomposes into disjoint sets of conditional equations and of predicate-headed clauses. But, as in [12], there is no essential difficulties to omit this hypothesis.

4.1 Model-theoretic and Fixpoint Semantics of Equational Logic Programs

Given an equational logic program P, the *Herbrand universe* $U(P)$ associated to P is the set of ground terms built over the set $\mathit{Func}(P)$ of function symbols appearing in P. The *Herbrand base* $B(P)$ is the set of ground atoms built over the set $\mathit{Pred}(P)$ of predicate symbols appearing in P : thus $B(P)$ includes the set of ground equations $s \simeq t$ where s and t belong to $U(P)$. We denote by $Gr(P)$ the set of ground instances of clauses of P. In standard logic programming, a Herbrand interpretation I is any subset of $B(P)$, but in the case of equational programs, a more specific notion is necessary. As usual, we shall identify a congruence $\sim$ with the set of equations $s \simeq t$ such that $s \sim t$. If I is a subset of $B(P)$, we denote $I^{\sim}$ the smallest congruence on $U(P)$ such that $s \sim t$ whenever $s \simeq t \in I$.

Definition **4** *A* (Herbrand) E-interpretation *I is a subset of $B(P)$ with the additional properties :*

- *$I^{\sim} \subseteq I$,*
- *if $p(t_1, \cdots, t_n) \in I$ (p distinct from $\simeq$) and $t_1 \simeq s_1 \in I, \cdots, t_n \simeq s_n \in I$ then $p(s_1, \cdots, s_n) \in I$.*

In the following, interpretation will mean Herbrand E-interpretation. Thus we have a simple notion of truth.

Definition **5** *Given an interpretation I, we say a ground atom is* E-true *in I if it is in I,* E-false *otherwise. Similarly, a conjunction of ground atoms is* E-true *in I iff all of the atoms are E-true in I,* E-false *otherwise.*

Definition **6** *Given an interpretation I, a ground clause C is* E-true *in I iff the head of C is E-true in I or its body is E-false in I. A clause C is* E-true *in I iff all its ground instances are E-true in I. In that case, I is a Herbrand* E-model *of C.*

In the following E-model will mean Herbrand E-model.

Definition **7** *An interpretation I is an* E-model *of an equational program P iff I is an E-model of each clause of P.*

There is a natural relationship between this notion of (Herbrand) E-model and the usual notion of Herbrand model. Let us consider the following set $Eq(P)$ of universally quantified first order axioms :
$x \simeq x \leftarrow$
$x \simeq y \leftarrow y \simeq x$
$x \simeq z \leftarrow x \simeq y, y \simeq z$
$f(x_1, \cdots, x_n) \simeq f(y_1, \cdots, y_n) \leftarrow x_1 \simeq y_1, \cdots, x_n \simeq y_n$ for each $f \in Func(P)$
$p(y_1, \cdots, y_n) \leftarrow p(x_1, \cdots, x_n), x_1 \simeq y_1, \cdots, x_n \simeq y_n$ for each $p \in Pred(P) \setminus \{\simeq\}$

We have the following property relating E-models and standard Herbrand models :

Proposition **1** *Let P an equational logic program. I is an E-model of P iff I is a Herbrand model of $P \cup Eq(P)$.*

Since the previous notion meets the model-intersection property and that $B(P)$ is an E-model of P, there is a *least E-model* : $M_E(P)$. This is the meaning of P. We have the following characterization of $M_E(P)$:

Proposition **2** *Given a ground atom A, $A \in M_E(P)$ iff $P \cup Eq(P) \models A$ (i.e. A is a logical consequence of $P \cup Eq(P)$).*

We recall now the definition of the immediate consequence operator. Since we have equations, the process is divided up into two steps. Let I be an E-interpretation of P.

Definition **8** $T_P(I) = TE_P(I) \cup TL_P(I)$ *with*
$TE_P(I) = \{s \simeq t \in B(P) \mid s \simeq t \leftarrow B \in Gr(P), B \text{ is E-true in } I\}^{\sim}$
and
$$\begin{array}{ll} TL_P(I) = & \{p(s_1, ..., s_n) \in B(P) \mid p(t_1, ..., t_n) \leftarrow B \in Gr(P), \\ & \bigwedge_i s_i \simeq t_i \wedge B \text{ is E-true in } I \cup TE_P(I)\} \end{array}$$

Informally, $TE_P(I)$ corresponds to the equational consequences of the program while $TL_P(I)$ concerns the logic part. The set of E-interpretations with subset inclusion forms a complete lattice : bottom element is $\{s \simeq s \mid s \in U(P)\}$ and top element is $B(P)$. T_P maps E-interpretations into E-interpretations and we have the following properties :

Proposition **3**
- *T_P is a continuous operator,*
- *I is an E-model of P iff $T_P(I) \subseteq I$,*
- *$M_E(P) = lfp(T_P) = T_P \uparrow \omega$.*

As in the standard case, the model-theoretic and the fixpoint semantics coincide.

4.2 A Fixpoint Semantics for Equational Rewrite Programs

We define now a fixpoint semantics for an equational rewrite program, $\psi(P)$. We must add to $B(P)$ the symbol *true*. As in the logic programming case, it is easy to define an operator similar to T_P. Each E-interpretation is augmented with the element *true* : the set of augmented E-interpretations is yet a complete lattice for subset inclusion with bottom element $\{s \simeq s \mid s \in U(P)\} \cup \{true\}$ and top element $B(P) \cup \{true\}$.

Definition **9** $T_{\psi(P)}(I) = TE_{\psi(P)}(I) \cup TL_{\psi(P)}(I)$ *with*
$TE_{\psi(P)}(I) = \{s \simeq t \in B(P) \mid s \simeq t, B \rightarrow B\{E\} \in Gr(P), B \wedge E \text{ E-true in } I\}^{\sim}$
and
$$\begin{aligned} TL_{\psi(P)}(I) = \ & \{p(s_1, ..., s_n) \in B(P) \mid p(t_1, ..., t_n), B \rightarrow B\{E\} \in Gr(P), \\ & B \wedge E \textstyle\bigwedge_i s_i \simeq t_i \text{ E-true in } I \cup TE_{\psi(P)}(I)\} \end{aligned}$$

There is no difficulty to prove :

Proposition **4** *$T_{\psi(P)}$ is a continuous operator.*

We have thus an equivalence between semantics of a logic program P and its associated rewrite program $\psi(P)$: this result claims the soundness of our transformation function ψ.

Proposition **5** *Given an equational logic program P, $lfp(T_P) \cup \{true\} = lfp(T_{\psi(P)})$.*

4.3 Equivalence with Operational Semantics

The (ground) operational semantics of our mechanism is naturally defined as follows :

Definition **10** *Let be $\psi(P)$ an equational rewrite program :*
$$\begin{aligned} \mathcal{O}(\psi(P)) = \ & \{p(\overline{s}) \in B(P) \mid p(\overline{s}) \rightarrow Ans \vdash Ans \rightarrow true\} \\ & \cup \{s \simeq t \in B(P) \mid Ans \rightarrow true\{s \simeq t\} \vdash Ans \rightarrow true\} \end{aligned}$$

In this section, we show that our mechanism is sound and complete with regard to the declarative semantics (the complete proofs can be found in [14]). In the equational case, a completeness result differs from the usual completeness result of Logic Programming, since we are only interested in finding the smallest solution w.r.t. the ordering. For instance, if we consider the two rules program : $f(x) \simeq x \leftarrow p(a)$ and $p(a) \leftarrow$, although $p(f^n(x))$ is valid for all n, we only generate the solution $p(a)$.

Proposition **6** *Given an equational program P, we have :*

$$\mathcal{O}(\psi(P)) = M_E(P)$$

5 Future Works and Conclusion

In this paper, we propose a scheme for the execution of equational logic programs based on an extension of Linear Completion. We transform the initial equational program into a set of rewrite rules and we define a fixpoint semantics for this equational rewrite program. The transformation is thus proved semantics preserving since the semantics of the logic program coincides with the fixpoint semantics of the corresponding rewrite program. The evaluation mechanism is described by an inference system which is shown sound and complete. The strategy is focused on a goal-oriented computation and restricts completion as far as possible. Thanks to Linear Completion, we gain w.r.t. the others mechanisms ([3, 6, 12]) synthesis ability and loop avoidance.

The method proposed here to compute in equational logic can be improved in several ways taking advantages of previous researches. Some works proposes an approach to include negation in equational logic programming [3] and [10]. Linear completion has already been extended to logic programs with negation [1] and this extension could be applied to the mechanism presented here. Work is in progress to include such a negation : this would offer a kind of constructive negation for equational logic programming.

On another hand, the constraint used here are only symbolic constraints. [13] shows that general constraint domains can be combined with linear completion methods. In an equational framework, it is tempting to deal with equations as constraints. But, in a standard constraint framework, the constraints are built-in predicates resolved apart by a specific solver, independent of the user-defined predicated or rules. In the case of equational programming, the equations (and thus the subjacent theory) could be user defined and could not be solved apart from the user program. A kind of dynamic solver is necessary here. This solver should probably cooperate with built-in solver that allows to use non symbolic constraints.

References

[1] S. Anantharaman and G. Richard. A Rewrite Mechanism for Logic Programs with Negation. In *Proceedings of RTA'95*, number 914 in LNCS, pages 163–178, KaisersLautern (Germany), 1995. Springer-Verlag.

[2] L. Bachmair and H. Ganzinger. Completion of first-order clauses with equality by strict superposition. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems*, number 516 in LNCS, pages 162–193. Springer-Verlag, 1990. 2nd International CTRS Workshop, Montreal, Canada, June 1990.

[3] L. Bachmair and H. Ganzinger. Perfect Model Semantics for Logic Programs with Equality. In Koichi Furukawa, editor, *Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*, pages 645–659. MIT Press, 1991.

[4] M.P. Bonacina and J. Hsiang. On Rewrite Programs : Semantics and Relationship with Prolog. *Journal of Logic Programming*, 14:155–180, 1992.

[5] N. Dershowitz and J.P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter Rewrite Systems, pages 243–309. J. Van Leeuwen, 1990.

[6] L. Fribourg. Oriented Equational Clauses as a Programming Language. *Journal of Logic Programming*, 1(2):165–177, August 1984.

[7] M. Hanus. The Integration of Functions into Logic Programming: from Theory to Practice. *Journal of Logic Programming*, 19,20:583–628, May/July 1994.

[8] S. Holldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Artificial Intelligence*. J. Siekman, springer-verlag edition, 1989.

[9] J. Jaffar, J.L. Lassez, and M. Maher. A Theory of Complete Logic Programs with Equality. *Journal of Logic Programming*, 3:211–223, 1984.

[10] S. Kaplan. Positive/Negative Conditional Rewriting. In S. Kaplan and J.P. Jouannaud, editors, *Conditional and Typed Rewriting Systems, 1srt Int. Workshop, Orsay (France)*, number 308 in LNCS, pages 129–141. Springer-Verlag, 1987.

[11] J.W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation series. Springer Verlag, 1987 (revised version).

[12] C. Lynch. Oriented Equational Logic Programming is Complete. 1995. submitted.

[13] G. Richard and F. Saubion. A Rewrite Approach to Transform Constraint Logic Programs. *Journal of Computing and Information*, Proceedings of ICCI'95, IEEE International Conference on Computing and Information, Trent University, Canada:184–200, 1995. http://www.phoenix.trentu.ca/jci.

[14] G. Richard and F. Saubion. Linear Completion for Equational Logic Programs. Technical report, University of Orleans (France), 1996. (to appear).

[15] J. Siekmann. An Introduction to Unification Theory. In R.B. Banerji, editor, *Formal Techniques in Artificial Intelligence*. Elsevier Scinces Publishers, 1990.

Program inversion in functional languages

Marko Schütz, Manfred Schmidt-Schauß
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
e-mail: {marko,schauss}@cs.uni-frankfurt.de

May 24, 1996

Combinator reduction systems with algebraic data types are the foundations for non-strict functional programming languages. Proving properties of functional programs is essential for verification, compilation and efficient execution of functional programs.
In contrast to strict functional programming languages that have a fixed (static) strategy to evaluate expressions, non-strict functional languages do not have such a prescribed simple evaluation strategy. They typically use so-called lazy evaluation, which combines call-by-need with sharing. A compiler for non-strict functional languages requires information about the cases in which a static evaluation strategy can be used in order to compile efficient code. This information is also helpful for implicit parallelisation.It is usually called "strictness" information. A function f is strict in its k^{th} argument, iff non-termination of a_k implies that evaluation of $(f a_1 \dots a_n)$ does not terminate. This is more or less equivalent to: The evaluation of a $(f a_1 \dots a_n)$ requires the evaluation of a_k.
A generalisation is context analysis, which also extracts information on deeper evaluation. For example, the function *length* will always evaluate the spine of the list, whereas the function *sum* will always evaluate the list to normal form.
Strictness analysis is usually done using abstract interpretation based on denotational semantics. We shall use abstract reduction, a kind of abstract interpretation using the operational instead of the denotational semantics. It is also related to top-down narrowing. However, we have the more general situation that at the function position, an expression is permitted.
The calculus is non-deterministic and constructs a tableau using expansion rules and rules testing for loops. The final labels at the leaves are used to represent the set of all solutions.
The presented calculus for strictness context analysis is able to solve constraints of the form.

$$t \in C$$

where t is an expression including free variables, and C is a context describing a set.

If we have defined lists using the constructors `Nil` and `Cons`, written using infix colons(:), and using a combinator *listcase* with the following definition:

> `listcase` `Nil` $f\ g = f$
> `listcase` (a:as) f g = g a as

then we define `length` as follows:

> `length` $xs =$ `listcase` xs 0 `lengthcons` ,
> `lengthcons` $y\ ys = 1 +$ (`length` ys)

Now consider the problem `length` $xs \in$ `Bot`, where `Bot` is a representation of undefined expressions and `Top` is a representation of all expressions.
The calculus will return the solution $xs =$ `INF`, where the context `INF` is recursively defined as `INF` $:= \{$`Bot`$\} \cup$ `Top` : `INF`.
The interpretation is that (`length` xs) is undefined if xs is an infinite list or a list with undefined tail. From the view of a compiler, this can be interpreted as follows: if (`length` xs) is to be evaluated, then it is safe to evaluate the spine of the list xs before applying `length` to xs.

As a further example consider

$$\begin{array}{lcl} \texttt{append}\ xs\ ys & = & \texttt{listcase}\ xs\ ys\ (\texttt{appendcons}\ ys) \\ \texttt{appendcons}\ ys\ z\ zs & = & z : (\texttt{append}\ zs\ ys) \end{array}$$

for which our calculus will produce the following result if asked how the arguments of `append` $xs\ ys$ can be evaluated if the application is evaluated.
The calculus starts with `append` $xs\ ys \in$ `Bot` and results in the following representation of all solutions:

$$(xs, ys) = (\texttt{Bot}, \texttt{Top}) \cup (\texttt{Top} : xs, \texttt{Top}) \cup (\texttt{Nil}, \texttt{Bot})$$

Essentially, it says that the first argument needs to be evaluated, but the second argument does not need to be evaluated as long as we do not reach the end of the spine of the first list.

Higher Order Unification as a Typed Narrowing

Daniel Briaud

Centre de Recherche en Informatique de Nancy (CNRS)
and INRIA-Lorraine
Campus Scientifique, BP 239,
F54506 Vandœuvre-lès-Nancy, France Fax: (33) 83 41 30 79

email: briaud@loria.fr

Abstract

We show how higher order unification (HOU) can be considered as a typed narrowing in a suitable first order equational theory. The theory in question is presented by a simple calculus of explicit substitutions, $\lambda\upsilon$, due to Lescanne. The main task consists in embedding HOU in $\lambda\upsilon$-unification and decoding $\lambda\upsilon$-unifiers as β-unifiers. Since $\lambda\upsilon$ is ground confluent and strongly normalizing on the set of simply typed terms, a typed narrowing, called $\lambda\upsilon$-narrowing is readily proved to be ground complete. This work may be seen as a combination of two previous ones. Dougherty used such an embedding in combinatory logic. A drawback is that the structure of the λ-terms is lost. More recently, Dowek, Hardin and Kirchner translated Huet's preunification algorithm in $\lambda\sigma$, another calculus of explicit substitutions. Beside the differences between HOU and preunification, $\lambda\upsilon$-narrowing analyzes in finer detail the process of building solutions.

A detailed version with proofs can be found at

`http://www.loria.fr/~briaud`

Introduction

Higher-order unification (HOU) solves equations where the unknowns may be functions, or, formally, equations between simply typed λ-terms. For instance,

$$x(f) == f$$

whose solutions are $x \mapsto \lambda u.u$ and $x \mapsto \lambda u.f$. As the simply typed λ-calculus is confluent and strongly normalizing, a simple and natural idea comes to mind, namely to unify via narrowing. The fact that β reduction is not a first-order rewrite relation constitutes the main difficulty.

A second difficulty is that HOU is impractical. Specifically, expressed as transformation rules [SG89], the search tree leading to a set of unifiers is infinitely branching. To overcome this problem, [Hue76] introduced the notion of preunifier. A given HOU problem has a preunifier if and only if it admits an

HO-unifier. As a consequence, preunification, like HOU, is undecidable. Nevertheless, this is a practical and basic procedure of higher-order theorem provers, such as Isabelle [Pau90], or higher-order logic programming languages, such as λProlog [MN86]. Still, due to β-reduction and the way substitutions are usually handled, preunification is thought to be difficult to implement.

Dougherty [Dou93] avoids the first mentioned problem by means of a translation to simply typed combinatory logic (CL), which he called C-unification. This requires a trick since CL involves only weak β-reduction. Dougherty proves that a variant of typed narrowing is complete with respect to C-unification, but in his approach, he looses the structure of the λ-terms.

Most of λ-calculi with explicit substitutions, such as $\lambda\sigma_{\Uparrow}$ and $\lambda\upsilon$, aim at expressing β-reduction by means of a first order term rewriting system. Thus, they give another way of translating HOU into a first order setting and of unifying via narrowing. These calculi are quite different. For instance, $\lambda\upsilon$ tries to introduce as few operators as possible, whereas $\lambda\sigma_{\Uparrow}$ introduces derived operations such as the composition of explicit substitutions; $\lambda\sigma_{\Uparrow}$ is confluent on open terms whereas $\lambda\upsilon$ is only ground confluent and PSN.

The work described in [DHK95] reduces HOU to a first-order equational unification in a theory presented by $\lambda\sigma_{\Uparrow}$. Then, although unification in such a calculus can be performed by narrowing, they present a specialized algorithm that computes $\beta\eta$-preunifiers for greater efficiency. Their approach could be improved in three respects: $\lambda\sigma_{\Uparrow}$ is rather complex, the $\lambda\sigma_{\Uparrow}$-unification rules can be made more elementary, and their rules for β-unification are incomplete.

Departing from [DHK95], our approach consists of two parts. First, the study in a simple setting of what we think is the heart of problem: how to decompose *in small steps* HOU by means of explicit substitutions. In other words, compute β-unifiers by means of $\lambda\upsilon$-narrowing. Second, the use of this study to design a preunification algorithm that makes small steps, easy to implement. In other words, compute $\beta\eta$-preunifiers in the spirit of $\lambda\upsilon$-narrowing. The work described here concerns the first part. Its first aim is to show that a system simpler than λ'*s* is sufficient to do the whole job. In particular, since $\lambda\upsilon$ does not include the composition of substitutions, we show that such a compositionis not mandatory to express HOU in an explicit substitutions framework. The second aim is to prove that $\lambda\upsilon$-narrowing is complete with respect to HOU and, as it is very simple to understand, to show that it provides an interesting alternative to build, at least manually, HO-unifiers.

1 Higher Order Unification

If M and N are two λ-terms of the same type then a *β-unifier* of M and N is a *substitution* θ such that $\theta(M) =_\beta \theta(N)$. A substitution is presented as a finite set of pairs (x_i, M_i). Applying such a substitution to a term N consists in replacing the free variables x_i of N by the terms M_i avoiding free variable of any M_i to be bound in the resulting term. Renaming bound variables in N prevents that capture. Such a renaming is part of the substitution process. Applying θ to M is also:

$$\theta(N) \equiv (\lambda x_1 \dots x_n.N)M_1 \dots M_n$$

For example, substituting x by y in $\lambda y.x$ yields $\lambda z.y$. On the contrary, first-order substitution, called *grafting* in this paper, does not take into account bound variables, hence, does not prevent the capture of free variables. For instance, grafting y for x in $\lambda y.x$ gives $\lambda y.y$. To sum up, substitution is grafting with renaming.
Let us illustrate HOU with an example. We consider a basic type i, $\tau(M)$ denotes the type of M and the following τ function,

$$\begin{array}{ll} \tau(g) = G = Y = i \rightarrow i \rightarrow i & \begin{array}{l} \tau(x) = G \rightarrow Z \rightarrow Y \rightarrow i \\ \tau(y) = Y \\ \tau(z) = Z \end{array} \\ \tau(a) = A = Z = i & \end{array}$$

The two λ-terms $xgzy$ and gaz admit as a β-unifier the substitution

$$\{(x, \lambda u_1 u_2 u_3.u_3(u_2, u_2)), (y, g), (z, a)\}$$

Generally, given a β-unification problem $M == N$, one is interested in a complete set of β-unifiers of $M == N$. Such a set, denoted $CSU_\beta(M == N)$, is a set of β-unifiers of M and N such that for any β-unifier θ of $M == N$, there is a substitution in $CSU_\beta(M == N)$ which subsumes θ.

We show now how to embed HOU into the $\lambda\upsilon$ framework. To ease the correspondence between β-unification and $\lambda\upsilon$-unification, it is convenient to abstract the free variables of the λ-terms to be unified as follows: let $\vec{x}$ be a ordering of $FV(MN)$, θ is a β-unifier of M and N iff θ is a β-unifier of $(\lambda\vec{x}.M)\vec{x}$ and $(\lambda\vec{x}.N)\vec{x}$. The interest of taking an ordering of $FV(MN)$ comes from the equivalences:

$$\begin{array}{rcl} \theta(M) & =_\beta & \theta(N) \\ \theta((\lambda\vec{x}.M)\vec{x}) & =_\beta & \theta((\lambda\vec{x}.N)\vec{x}) \\ (\lambda\vec{x}.M)\theta(\vec{x}) & =_\beta & (\lambda\vec{x}.N)\theta(\vec{x}) \end{array}$$

$\lambda\vec{x}.M$ and $\lambda\vec{x}.N$ which contain no free variables are easily translated into terms with de Bruijn indices according to the following syntax where A is a type, f is a constant and $c_x \in C_V$.

$$\begin{array}{lll} Nat & n ::= & 0 \mid n+1 \\ Pure\ Term & a ::= & f \mid c_x \mid \underline{n} \mid aa \mid \lambda A.a \end{array}$$

The set C_V is a mirror of the set of variables V. We do not detail here its motivation, which is quite technical.

The function $DB_{\vec{x}}$, where $\vec{x}$ is a sequence of variables, translates a λ-term to a pure term:

$$\begin{array}{lcl}
DB_{\vec{x}}(x) & = & j \text{ if j is the index of the first occurrence of x in } \vec{x} \\
DB_{\vec{x}}(y) & = & c_y \text{ if } y \notin \vec{x} \\
DB_{\vec{x}}(f) & = & f \\
DB_{\vec{x}}(MN) & = & DB_{\vec{x}}(M)\ DB_{\vec{x}}(N) \\
DB_{\vec{x}}(\lambda y.M) & = & \lambda A.\ DB_{y\cdot\vec{x}}(M) \text{ if } y \text{ is of type } A
\end{array}$$

If $\vec{x}$ is void, then we shall write DB instead of $DB_{\vec{x}}$. For example,

$$\begin{array}{rcl}
DB(\lambda x.\lambda y.\lambda z.xgzy) & = & \lambda X.\lambda Y.\lambda Z.\ \underline{2}\ g\ \underline{0}\ \underline{1} \\
DB(\lambda x.\lambda y.\lambda z.gaz) & = & \lambda X.\lambda Y.\lambda Z.\ g\ a\ \underline{0}
\end{array}$$

The function $BD_{\vec{x}}$, extracting a λ-term from a pure term can be defined likewise and will be used to decode $\lambda\upsilon$-unifiers. $DB_{\vec{x}}$ and $BD_{\vec{x}}$ are converse.

2 $\lambda\upsilon$-unification

Before introducing $\lambda\upsilon$-unification, we simply type $\lambda\upsilon$ and include variables of V.

$$\begin{array}{llll}
Nat & n & ::= & 0 \mid n+1 \\
Term & a & ::= & x \mid f \mid c_x \mid \underline{n} \mid aa \mid \lambda A.a \mid a[s] \\
Subst & s & ::= & a/ \mid \uparrow \mid \Uparrow(s)
\end{array}$$
where $x \in V$, f is a constant, $c_x \in C_V$,

Moreover we type B and $Lambda$ and we add two rules to $\lambda\upsilon$ as follows:

$Beta$	$(\lambda A.x)b$	$\to$	$x[b/]$
App	$(xy)[s]$	$\to$	$x[s]y[s]$
$Lambda$	$(\lambda A.x)[s]$	$\to$	$\lambda A.x[\Uparrow(s)]$
$FVar$	$\underline{0}[b/]$	$\to$	b
$RVar$	$\underline{n+1}[b/]$	$\underline{n}$	
$FVarLift$	$\underline{0}[\Uparrow(s)]$	$\to$	$\underline{0}$
$RVarLift$	$\underline{n+1}[\Uparrow(s)]$	$\underline{n}[s][\uparrow]$	
$VarShift$	$\underline{n}[\uparrow]$	$\to$	$\underline{n+1}$
$Const$	$f[s]$	$\to$	f
$ConstVar$	$c_x[s]$	$\to$	c_x

Like higher-order unification, $\lambda\upsilon$-unification is defined on the set $\Lambda\Upsilon$ of simply typed $\lambda\upsilon$-terms. $\lambda\upsilon$ is confluent on *ground* $\lambda\upsilon$-terms of $\Lambda\Upsilon$ and $\lambda\upsilon$ is strongly normalizing on $\Lambda\Upsilon$ [LRD94, BBLRD96].
To illustrate how variables are mixed with ground terms, consider for instance

$$(\lambda x.\lambda y.\lambda z.xgzy)x \xrightarrow[\beta]{} \lambda y.\lambda z.xgzy$$

With $\lambda\upsilon$, the combinator $\lambda X.\lambda Y.\lambda Z.\underline{2}\ g\ \underline{0}\ \underline{1}$ applied to the *variable* x can be rewritten:

$$\begin{array}{rcl}(\lambda X.\lambda Y.\lambda Z.\underline{2}\ g\ \underline{0}\ \underline{1})x & \xrightarrow[Beta]{} & (\lambda Y.\lambda Z.\underline{2}\ g\ \underline{0}\ \underline{1})\ [x/] \\ & \xrightarrow[\lambda\upsilon]{*} & \lambda Y.\lambda Z.((\underline{2}\ g\ \underline{01})\ [\Uparrow^2(x/)]) \\ & \xrightarrow[\lambda\upsilon]{*} & \lambda Y.\lambda Z.\underline{2}\ [\Uparrow^2(x/)]\ g\ [\Uparrow^2(x/)]\ \underline{0}\ [\Uparrow^2(x/)]\ \underline{1}\ [\Uparrow^2(x/)] \\ & \xrightarrow[\lambda\upsilon]{*} & \lambda Y.\lambda Z.\underline{0}\ [x/][\uparrow][\uparrow]\ g\ \underline{0}\ \underline{1} \\ & \xrightarrow[\lambda\upsilon]{} & \lambda Y.\lambda Z.x\ [\uparrow][\uparrow]g\ \underline{0}\ \underline{1}\end{array}$$

In the last $\lambda\upsilon$-term, a grafting replaces x by a $\lambda\upsilon$-term. Unlike in the $\lambda\upsilon$-term $\lambda A.x$, no capture may happen in $\lambda Y.\lambda Z.x\ [\uparrow][\uparrow]\ g\ \underline{0}\ \underline{1}$, as the two $\uparrow$ act as renaming operators.
Precisely, a *grafting* is a function $\theta : V \to Term$, identified with its unique homomorphic extension, such that $x\theta \neq x$ for only finitely many $x \in V$.
Graftings play for $\lambda\upsilon$-unification the same role as substitutions for HOU. Specifically, if a and b are two $\lambda\upsilon$-terms of the same type then a *$\lambda\upsilon$-unifier* of a and b is a grafting θ such that $a\theta =_{\lambda\upsilon} b\theta$. Such an equation is denoted $a == b$. For example, one may check that $xgzy == gaz$ admits the grafting $\{x \mapsto \lambda GZY.\underline{0}\ \underline{1}\ \underline{1}, y \mapsto g, z \mapsto a\}$ as a $\lambda\upsilon$-unifier.

3 Embedding HOU into $\lambda\upsilon$-unification

H embeds HOU into $\lambda\upsilon$-unification. It associates a $\lambda\upsilon$-unification problem to a β-unification problem.
Let M, N be two λ-terms of the same type, and $\vec{x}$ an ordering of $FV(MN)$. The $\lambda\upsilon$-unification problem associated with $M == N$ is

$$H(M) == H(N)$$

where

$$H(M) = DB(\lambda\vec{x}.M)\vec{x}, H(N) = DB(\lambda\vec{x}.N)\vec{x}$$

At first sight, the associated problem depends on $\vec{x}$, but in fact, this is not the case.
If $M = \lambda u.z$ and $N = \lambda u.a$ are of type $i \to i$, then

$$\begin{array}{rcccl}H(M) & = & (\lambda Z\lambda i.\underline{1})z & =_{\lambda\upsilon} & \lambda i.z[\uparrow] \\ H(N) & = & (\lambda Z\lambda i.a)z & =_{\lambda\upsilon} & \lambda i.a\end{array}$$

$\uparrow$ in $H(M)$ prevents z from being captured by a grafting like $[\underline{1}/z]$.

Theorem 1 *Let $M == N$ be a β-unification problem. The β-unifiers of $M == N$ are exactly the* ground *$\lambda\upsilon$-unifiers of $H(M) == H(N)$.*

4 $\lambda\upsilon$-narrowing

Consider the equation $xa == ay$ with the following types:

$$\begin{aligned}\tau(a) &= Y \to B = A\\ \tau(x) &= (Y \to B) \to B\\ \tau(y) &= Y\end{aligned}$$

Expressed in the $\lambda\upsilon$-unification framework, the equation $xa == ay$ is unchanged. Here is a branch of the search space explored by $\lambda\upsilon$-narrowing on that equation:

$$\begin{tikzcd}
xa == ay \arrow[d, "Beta"', "x\mapsto\lambda A.x_1 \quad \tau(x_1)=B"] \\
x_1[a/] == ay \arrow[d, "App"', "x_1\mapsto x_2x_3 \quad \tau(x_2)=Y\to B,\ \tau(x_3)=Y"] \\
x_2[a/]x_3[a/] == ay \arrow[d, "FVar"', "x_2\mapsto\underline{0}"] \\
a(x_3[a/]) == ay \arrow[d, "Typed\ Unif"'] \\
y == x_3[a/]
\end{tikzcd}$$

The $\lambda\upsilon$-narrowing steps of this branch build the non-ground $\lambda\upsilon$-unifier

$$\sigma = \{x \to \lambda A.(\underline{1}\ x_3),\ y \to x_3[a/]\}.$$

More generally,

Theorem 2 *$\lambda\upsilon$-narrowing is a* ground *complete method of $\lambda\upsilon$-unification.*

5 Translation of non-ground $\lambda\upsilon$-unifiers

The issue is now to translate the graftings produced by $\lambda\upsilon$-narrowing into β-unifiers. Ground graftings are translated by BD. What about non-ground graftings? In the last example, we get the non-ground $\lambda\upsilon$-unifier

$$\sigma = \{x \mapsto \lambda A.(\underline{1}\ x_3),\ y \mapsto x_3[a/]\}$$

We would like to get the β-unifier

$$\{(x, \lambda u.u(x_3(u))), (y, x_3a)\}$$

To translate σ, we just need to compose it with a ground grafting, called σ_B. The subscript B in σ_B means "back translation". The rôle of σ_B is the following:

given a λ-term $\lambda A.\underline{0}\ x_3$, the fact that x_3 depends on $\underline{0}$, i.e. may be replaced by a term containing $\underline{0}$, is implicit. σ_B makes this dependency explicit:

$$\sigma_B = \{x_3 \mapsto c_z\ \underline{0}\}$$

Indeed, the new constant $c_z \in C_V$ is a function of $\underline{0}$, or, put it differently, depends explicitly on $\underline{0}$. Furthening the previous example, we get:

$$\sigma\sigma_B =_{\lambda\upsilon} \{x \mapsto \lambda A.\underline{0}(c_z\underline{0}), y \mapsto c_z a, x_3 \mapsto c_z\ \underline{0}\}$$

$\sigma\sigma_B$ is ground $\lambda\upsilon$-unifier of $xa == ay$. Hence by theorem 1, it can be translated into a β-unifier of $xa == ay$:

$$\{(x, \lambda u.u(zu)), (y, za)\}$$

Theorem 3 *Let $M == N$ be a β-unification problem. Let S the set of graftings produced by $\lambda\upsilon$-narrowing and solving $H(M) == H(N)$. One can extract a $CSU_\beta(M == N)$ out of S.*

6 Conclusion

The main interest of this work is that it provides an *elementary* procedure, $\lambda\upsilon$-narrowing, for a quite complex problem, higher order unification. However, as already said, HOU, and $\lambda\upsilon$-narrowing alike, are impractical due to the presence of infinitely branching nodes in the search space. Our current work consists in designing transformation rules that makes *as small steps as* $\lambda\upsilon$-narrowing and produce $\beta\eta$-preunifiers, still in the framework described here. Such an approach is interesting in the sense that it would give a simple preunification algorithm very near to its implementation.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[BBLRD96] Z. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 1996. à paraître.

[dB72] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen*, 75(5):381–392, 1972.

[DHK94] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. Technical Report 94-R-243, CRIN, December 1994.

[DHK95] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions, extended abstract. In Dexter Kozen, editor, *Proceedings of LICS'95*, pages 366–374, San Diego, June 1995.

[Dou93] D. J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114:273–298, 1993.

[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.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*, pages 60–69. ACM, 1994.

[LRD94] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions $\lambda\upsilon$. Technical Report RR-2222, INRIA-Lorraine, January 1994.

[MN86] D. A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, volume 225 of *Lecture Notes in Computer Science*, pages 448–462. Springer-Verlag, 1986.

[Pau90] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic in Computer Science*. Academic Press, 1990.

[Río93] A. Ríos. *Contributions à l'étude des λ-calculs avec des substitutions explicites*. Thèse de Doctorat d'Université, U. Paris VII, 1993.

[SG89] W. Snyder and J. Gallier. Higher order unification revisited: Complete sets of tranformations. *Journal of Symbolic Computation*, 8(1 & 2):101–140, 1989. Special issue on unification. Part two.

Decidable Linear Second-Order Unification Problems

Jordi Levy
Technical University of Catalonia
Pau Gargallo 5, 08028 Barcelona, Spain
E-mail: levy@lsi.upc.es

Linear Second-Order Unification deals with completely general second-order typed unification problems, where the set of unifiers under consideration is restricted: they instantiate free variables by linear terms, i.e. terms where any λ-abstraction binds one and only one occurrence of a bound variable. This properly extends *context unification*, studied by Comon and Schmidt-Schauß, considering non-unary variables and λ-bindings. This also makes some (trivial) unification problems finitary, which would be infinitary considered as context unification problems. In this talk we present some classes of decidable linear second-order unification problems, trying to enlarge the class defined by Schmidt-Schauß for context unification.

AC-complete Unification and its Application to Theorem Proving[*]

Alexandre Boudet, Evelyne Contejean and Claude Marché
LRI, CNRS URA 410
Bat. 490, Université Paris-Sud, Centre d'Orsay
91405 Orsay Cedex, France
Phone:+33 1 69 41 69 05, Fax: +33 1 69 41 65 86
Email: {boudet,contejea,marche}@lri.fr

The inefficiency of AC-completion is mainly due to the doubly exponential number of AC-unifiers and thereby of critical pairs generated. We present AC-complete E-unification, a new technique whose goal is to reduce the number of AC-critical pairs inferred by performing unification in a extension E of AC (*e.g.* ACU, Abelian groups, Boolean rings, ...) in the process of normalized completion [1, 2]. The idea is to represent complete sets of AC-unifiers by (smaller) sets of E-unifiers. Not only do the theories E used for unification have exponentially fewer most general unifiers than AC, but one can remove from a complete set of E-unifiers those solutions which have no E-instance which is an AC-unifier.

First, we define AC-complete E-unification and describe its fundamental properties. We show how AC-complete E-unification can be done in the elementary case, and how the known combination techniques for unification algorithms can be reused for our purposes. Finally, we give some evidence of the kind of speedup that can be obtained by presenting some experiments with the C*i*ME theorem prover.

References

[1] C. Marché. Normalised rewriting and normalised completion. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 394–403, Paris, France, July 1994. IEEE Comp. Soc. Press.

[2] C. Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 1996. to appear.

[*]This research was supported in part by the EWG CCL, the HCM Network CONSOLE, and the "GDR de programmation du CNRS".