Proceedings of Sixth International Workshop on Unification

Schloss Dagstuhl, Germany

Franz Baader Jörg Siekmann Wayne Snyder
(Organizers)

July 29 – 31, 1992

Contents

Conditional Rewriting Modulo a Built-in Algebra

Jürgen Avenhaus and Klaus Becker[1]

1 Motivation

Conditional equational specifications can be regarded as programs of a functional programming language with conditional rewriting as its computation mechanism. Whereas built-in concepts can be used in common programming languages, they usually are not available in rewrite environments.

In the following example, which is intended to define the greatest common divisor function over the natural numbers, it would be desirable to consider '+' and '0' as symbols with a predefined meaning, that agrees with the common interpretation of the symbols over the natural numbers. However, in most rewrite based specification environments these equations would not be "executable".

$$\begin{array}{lrcl}
(1) & g(x,0) & = & x \\
(2) & g(0,y) & = & y \\
(3) & g(x+y,y) & = & g(x,y) \\
(4) & g(x,x+y) & = & g(x,y)
\end{array}$$

Built-in concepts are attractive for several reasons:

- Built-ins allow programming at a higher level of abstraction. The "basic world of interest" can be assumed to be given, it need not be modeled in a bottom-up fashion.
- Built-ins possibly allow to gain efficiency, if appropriate built-in algorithms are available to treat the predefined structures.
- Built-ins can increase the expressive power of equational specification environments. By a built-in algebra we are able to describe "non-constructive" domains, as for instance real numbers, that cannot be characterized equationally.

[1]Fachbereich Informatik, Universität Kaiserslautern, W 6750 Kaiserslautern, Germany, email: `avenhaus@informatik.uni-kl.de`.

The aim of this paper (for an extended version see [AvBe92]) is to present a general approach of how to integrate predefined objects and operations into rewrite based equational reasoning. There has been done some related work by Vorobyov [Vo89] about built-in arithmetic, by Kaplan, Choppy [KaCh89] and Walters [Wa90] about implementational aspects and Kirchner, Kirchner, Rusinowitch [KKR90] and Comon [Co92] about equational specifications with constraints.

2 Syntax

Let $\Sigma_0 + \Sigma_1$ be an "ordinary" flat functional signature enrichment. Σ_0 introduces the basic sorts and objects. The built-in structure over the basic domain is intended to be given by a Σ_0-algebra — the built-in algebra. We assume for simplicity that the sorts in S_0 are not empty. Σ_1 introduces new function symbols over the basic sorts, which are intended to be partially defined by the equations of the specification. We thus mix semantically treated Σ_0-symbols and syntactically treated Σ_1-symbols. In order to allow separation between semantically and syntactically oriented operations, we restrict the range of the variables that occur in the equations: Variables are to be instantiated only by built-in objects resp. Σ_0-terms.

In order to design this kind of restriction, we use a suitable order sorted interpretation Σ of $\Sigma_0 + \Sigma_1$, based on a copying process (see also [SNGM89], where the same approach is used to describe partial functions). Let F_0 be divided into $F_0^{(=0)}$ and $F_0^{(\geq 1)}$, the set of symbols from F_0 with an arity that is equal to 0 — the constants from F_0 — resp. greater or equal than 1.

Definition 1 *Let* $\Sigma_0 + \Sigma_1 = (S_0, F_0, D_0) + (\emptyset, F_1, D_1)$ *be a flat functional signature enrichment.* $\Sigma = (S, F, D)$ *is said to be the* hierarchical signature resp. order-sorted signature induced by Σ_0 and Σ_1 — *written* $\Sigma = \Sigma_0 \oplus \Sigma_1$ — *iff*

- $S = S_0 \cup S_0^\wedge$ *where* $S_0^\wedge = \{s^\wedge \mid s \in S_0\}$ *and*
- $F = F_0 \cup F_1$ *and*
- $D = D_0 \cup D^\wedge \cup D_{sort}$ *where*
  $D^\wedge = \{f : s_1^\wedge, \ldots, s_n^\wedge \to s^\wedge \mid f \in F_0^{(\geq 1)} \cup F_1$ *and* $f : s_1, \ldots, s_n \to s \ \in D_0 \cup D_1\}$ *and*
  $D_{sort} = \{s \lhd s^\wedge \mid s \in S_0\}$.

Variables are introduced only for the base sorts in S_0. Let $V = \bigcup_{s \in S_0} V_s$ be the union of disjoint infinitary sets V_s of variables for the basic sorts. Σ-terms are defined as usual. A *conditional equation over* Σ is a formula $\bigwedge_{i=1}^{n} u_i = v_i \Rightarrow u = v$ such that u_i and v_i resp. u and v are sort compatible Σ-terms (see [SNGM89]).

Definition 2 *A* specification with a built-in algebra $(\Sigma = \Sigma_0 \oplus \Sigma_1, E, \mathcal{A})$ *consists of a hierarchical signature* Σ *induced by a flat functional signature enrichment* $\Sigma_0 + \Sigma_1$*, a set* E *of conditional equations over* Σ *and a term-generated (built-in)* Σ_0*-algebra* $\mathcal{A}$*.*

3 Semantics

We consider a specification $(\Sigma = \Sigma_0 \oplus \Sigma_1, E, \mathcal{A})$ with built-in algebra $\mathcal{A}$. First, the meaning of the specification is characterized denotationally. The built-in algebra $\mathcal{A}$ is taken into account by $E_{\mathcal{A}} = \{u = v \mid \mathcal{A} \models u = v; u, v \in TERM_0(\Sigma, V)\}$, the *set of* Σ_0*-equations induced by* $\mathcal{A}$. The algebras that capture the denotational meaning of the specification are the order-sorted Σ-algebras that are models of $E \cup E_{\mathcal{A}}$. These algebras have a "core", constituted by the base elements, that contain a uniquely defined homomorphic image of the built-in algebra $\mathcal{A}$. Further, they satisfy E, but due to our hierarchical concept in the following weak sense: Only assignments that (correctly) instantiate variables by "base elements" have to be taken into account.

Next we present an inference system, depending on E and an additional set S of Σ-equations, in order to define the semantics of the specification operationally. The rules of this inference system model (R)eflexivity, (S)ymmetry, (T)ransitivity, (C)ongruence and (E)-application (see [Ga91] for a similar approach).

$$(R) \quad \frac{}{u = u}$$

$$(S) \quad \frac{u = v}{v = u}$$

$$(T) \quad \frac{u = v, v = w}{u = w}$$

$$(C) \quad \frac{u_i = v_i}{f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)} \qquad \text{,if } f(u_1, \ldots, u_n), f(v_1, \ldots, v_n) \text{ are well-formed.}$$

$$(E) \quad \frac{\sigma(u_i) = \sigma(v_i)}{\sigma(u) = \sigma(v)} \qquad \text{,if } \bigwedge_{i=1}^{n} u_i = v_i \Rightarrow u = v \in E,\ \sigma(u_i) = \sigma(v_i) \in S.$$

We write $S \vdash_E u = v$ to indicate that $u = v$ can be derived from S by the above inference rules (depending on E).

Let $\sim_{\mathcal{A}} = \overset{*}{\longleftrightarrow}_{E_{\mathcal{A}}}$ be the *congruence relation on* $TERM(\Sigma, V)$ *induced by* $\mathcal{A}$. Note that if e.g. $x + y = y + x \in E_{\mathcal{A}}$ and $f, a, b \in F_1$, then $f(a + b) \sim_{\mathcal{A}} f(b + a)$, but not necessarily $f(a) + f(b) \sim_{\mathcal{A}} f(b) + f(a)$, as x, y cannot be instantiated by $f(a), f(b)$.

According to the intuition that built-in equivalences are given in advance, the inductive definition to follow starts with the congruence relation $\sim_{\mathcal{A}}$. Let $\sim^0_{E,\mathcal{A}} = \sim_{\mathcal{A}}$. For all Σ-terms u, v let $u \sim^{i+1}_{E,\mathcal{A}} v$ iff $u \sim^i_{E,\mathcal{A}} v$ or $\sim^i_{E,\mathcal{A}} \vdash_E u = v$. Finally, let $\sim_{E,\mathcal{A}} = \bigcup_{i \geq 0} \sim^i_{E,\mathcal{A}}$. Obviously, $\sim_{E,\mathcal{A}}$ is a congruence relation on $TERM(\Sigma, V)$, the *congruence relation induced by* E *and the built-in algebra* $\mathcal{A}$.

The following "Birkhoff-theorem" states the equivalence of the denotational and operational semantics of the specification.

Theorem 1 *Let* $(\Sigma = \Sigma_0 \oplus \Sigma_1, E, \mathcal{A})$ *be a specification with built-in algebra* $\mathcal{A}$*. Then for any* $s, t \in TERM(\Sigma, V)$*,* $s \sim_{E,\mathcal{A}} t$ *iff* $E \cup E_{\mathcal{A}} \models s = t$*.*

4 Rewriting

A *conditional rewrite rule over* Σ is a (directed) conditional equation $\bigwedge_{i=1}^{n} u_i = v_i \Rightarrow u = v$, where all variables occuring in v, u_i, v_i also occur in u, and where the left hand side u is a term of a sort in $S_0^\wedge$. The latter condition, meaning that the left hand side of a rewrite rule has to contain a new symbol, is a kind of constructor preserving property, provided we interprete base terms as constructors. In addition, this condition assures that rewrite rules are always sort-decreasing, i.e. $sort(u) \trianglerighteq sort(v)$ (see [SNGM89] for the relevance of this property).

Next we define conditional rewriting modulo the built-in algebra $\mathcal{A}$, which is essentially conditional rewriting of $\sim_{\mathcal{A}}$-equivalence classes.

Definition 3 *Let R be a set of conditional rewrite rules over Σ and $s, t \in TERM(\Sigma, V)$. s* rewrites to *t* modulo *$\mathcal{A}$, written $s \longrightarrow_{R/\mathcal{A}} t$, iff there exist terms $s', t' \in TERM(\Sigma, V)$, an occurence $p \in O(s')$, a substitution σ and a rule $\bigwedge_{i=1}^{n} u_i = v_i \Rightarrow u = v \ \in R$ such that*

- *$s \sim_{\mathcal{A}} s'$, $s'/p = \sigma(u)$, $s'[p \leftarrow \sigma(v)] = t'$, $t' \sim_{\mathcal{A}} t$ and*
- *for any $i \in \{1, \ldots, n\}$ there exist $u_i', v_i' \in TERM(\Sigma, V)$ such that $\sigma(u_i) \overset{*}{\longrightarrow}_{R/\mathcal{A}} u_i' \sim_{\mathcal{A}} v_i' \ {}_{R/\mathcal{A}}\overset{*}{\longleftarrow} \sigma(v_i)$ (resp. $\sigma(u_i) \downarrow_{R/\mathcal{A}} \sigma(v_i)$).*

Let R be a set of conditional rewrite rules over Σ. R is said to be *Church-Rosser modulo $\mathcal{A}$* iff for any $s, t \in TERM(\Sigma, V)$, $s \sim_{R,\mathcal{A}} t$ iff $s \downarrow_{R/\mathcal{A}} t$. R is said to be *confluent modulo $\mathcal{A}$ (resp. locally confluent modulo $\mathcal{A}$)* iff for any $s, s_1, s_2 \in TERM(\Sigma, V)$: if $s_1 \ {}_{R/\mathcal{A}}\overset{*}{\longleftarrow} s \overset{*}{\longrightarrow}_{R/\mathcal{A}} s_2$ (resp. $s_1 \ {}_{R/\mathcal{A}}\longleftarrow s \longrightarrow_{R/\mathcal{A}} s_2$), then $s_1 \downarrow_{R/\mathcal{A}} s_2$.

The following theorem relates the operational and rewrite semantics.

Theorem 2 *Let R be a conditional rewrite system over Σ. Then R is Church-Rosser modulo $\mathcal{A}$ iff R is confluent modulo $\mathcal{A}$.*

By Newman's lemma, if $\longrightarrow_{R/\mathcal{A}}$ is terminating, then R is confluent modulo $\mathcal{A}$ iff R is locally confluent modulo $\mathcal{A}$. Local confluence can be tested as usual by considering critical equations. For that reason we first generalize unification by introducing a suitable notion of constraints in order to be able to represent the solutions of an equation in a finite way.

A substitution σ *satisfies a set S of Σ-equations modulo $\mathcal{A}$ resp. is an $\mathcal{A}$-solution of S* iff $\sigma(s) \sim_{\mathcal{A}} \sigma(t)$ for all $s = t \in S$. A *constraint* γ is either a finite conjunction (or set) of Σ_0-equations or one of the symbols $\top$ ("solved by every σ") and $\perp$ ("solved by no σ").

Lemma 1 *For any finite set S of Σ-equations one can compute a constraint γ_S such that S and γ_S have the same $\mathcal{A}$-solutions.*

Definition 4 *Let $U \Rightarrow u = v$ and $U' \Rightarrow u' = v'$ be two conditional rules over Σ that have no variables in common. Let p be a position in u such that u/p is no variable. Then the conditional equation $\gamma_{u/p=u'} \wedge U \wedge U' \Rightarrow u[p \leftarrow v'] = v$ is called a* (conditional) critical equation between the two rules.

Note that there exist no variable overlaps, since variables are introduced only for the base sorts and since the left hand sides of the rules have to be of a new sort. So we do not have to require R to be decreasing in order to prove local confluence, in contrast to the syntactical case (see [DeOk90]).

Theorem 3 *Let R be a conditional rewrite system over Σ. If all conditional critical equations, that can be built from the rules of R, are joinable modulo $\mathcal{A}$, then R is locally confluent modulo $\mathcal{A}$.*

5 Termination

In this section we only sketch some ideas about how to prove termination of the rewrite relation $\longrightarrow_{R/\mathcal{A}}$. Technical details can be found in the extended paper [AvBe92].

The equations from the example above do not induce a terminating rewrite relation modulo the natural number interpretation $\mathcal{N}$, as $g(x,0) \sim_{\mathcal{N}} g(x+0,0)$ and hence $g(x,0) \longrightarrow_{R/\mathcal{N}} g(x,0)$. We change the rewrite rules (3) and (4) by adding constraints.

$$\begin{array}{llcllcl} (3') & y \succ 0 & = & true & \Rightarrow & g(x+y,y) & = & g(x,y) \\ (4') & x \succ 0 & = & true & \Rightarrow & g(x,x+y) & = & g(x,y) \end{array}$$

Now we may use the well-foundedness of the algebra $\mathcal{N}$ wrt. $\succ$ and $true$ and the fact, that $\mathcal{N} \models y \succ 0 = true \Rightarrow x+y \succ x = true$ and $\mathcal{N} \models x \succ 0 = true \Rightarrow x+y \succ y = true$, to conclude that $\longrightarrow_{R/\mathcal{N}}$ is terminating.

We capture these ideas by a suitable recursive path ordering construction [De87]. First, conditions are divided into a constraint part and a rest part. A rewrite rule then has the form $\gamma \wedge C \Rightarrow u = v$. Further, Σ_0-terms are viewed as semantic units. For that reason we introduce a new constant $[u]$ for any Σ_0-term u. The terms to be considered consist of syntactic operators $f \in F$ and semantic units $[u]$. The precedence, that allows to compare such terms by recursive decomposition, is constituted by the following three parts: (i) $\gtrsim_F$, the given "syntactical precedence" that allows to compare syntactic operators, (ii) $>_H = \{(f,[u]) \mid f \in F, u \in TERM(\Sigma_0, V)\}$, the "hierarchy precedence" that makes every term of a sort in $S_0^\wedge$ greater than every Σ_0-term, and (iii) $\gtrsim_0^{(\gamma)}$, the "semantical base ordering system" that allows to compare semantic units.

The base ordering system provides an ordering for any constraint γ. These orderings are assumed to satisfy some basic properties as well-foundedness and compatibility with the congruence relation $\sim_{\mathcal{A}}$.

In the above example we can define for any γ a base ordering by $\gtrsim_0^{(\gamma)} = \{([u],[v]) \mid \mathcal{N} \models \gamma \Rightarrow u \succeq v\}$, where $\succeq$ is the normal ordering on $\mathcal{N}$.

Let $>_{rpo,\mathcal{A}}^{(\gamma)}$ be the recursive path ordering induced by the union of the above precedences (for any constraint γ). This construction allows us to lift semantic properties as compatibility with the built-in algebra $\mathcal{A}$ from the base orderings (over Σ_0-terms) to the recursive path orderings (over the Σ-terms).

We then obtain the following main result: If $u >_{rpo,\mathcal{A}}^{(\gamma)} v, u_i, v_i$ ($u_i = v_i \in C$) for any rule $\gamma \wedge C \Rightarrow u = v$, then rewriting by $\longrightarrow_{R/A}$ as well as the recursive evaluation of conditions is terminating.

References

[AvBe92] J. Avenhaus and K. Becker. Conditional rewriting modulo a built-in algebra. SEKI Report SR-92-11.

[Co92] H. Comon. Completion of rewrite systems with membership constraints. *ICALP '92*, LNCS 623, (Springer, Berlin, 1992) pp. 392-403.

[De87] N. Dershowitz. Termination of rewriting. *J. Symbolic Computation* 3 (1987) pp. 69-116.

[DeOk90] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science* 75 (1990) pp. 111-138.

[Ga91] H. Ganzinger. Order-sorted completion: the many-sorted way. *Theoretical Computer Science* 89 (1991) pp. 3-32.

[KaCh89] S. Kaplan and C. Choppy. Abstract rewriting with concrete operators. *3rd RTA '89*, LNCS 355, (Springer, Berlin, 1989) pp. 178-185.

[KKR90] C. Kirchner, H. Kirchner and M. Rusinowitch. Deduction with symbolic constraints *Revue d'Intelligence Artificielle* 4(3) (1990) pp. 9-52.

[SNGM89] G. Smolka, W. Nutt, J.A. Goguen and J. Meseguer. Order-sorted equational computation. H. Ait-Kaci and M. Nivat, eds., *Resolution of Equations in Algebraic Structures, Vol. 2* (Academic Press, San Diego CA, 1989) pp. 297-367.

[Vo89] S.G. Vorobyov. Conditional rewrite rule systems with built-in arithmetic and induction. *3rd RTA '89*, LNCS 355, (Springer, Berlin, 1989) pp. 492-512.

[Wa90] H.R. Walters. Hybrid implementations of algebraic specifications. *Algebraic and Logic Programming, Proc. 2nd Int. Conf. 1990*, LNCS 463, (Springer, Berlin, 1990) pp. 40-54.

Associative Commutative Matching Based on the Syntacticity of the AC Theory

Mohamed Adi and Claude Kirchner[1]

We present a new Associative-Commutative matching algorithm. It is based on the syntacticity of associative-commutative theories. As shown by Tobias Nipkow, it is possible to built a matching algorithm from a resolvant presentation of an AC theory and we have shown how this algorithm can be improved in such a way that most redundant computations are avoided. The resulting algorithm has been implemented and, compared to the algorithms that solve the AC-matching problem using solving of inhomogeous linear Diophantine equations, it gives much better performance, in particular a first match is computed several orders of magnitude faster.

[1]INRIA Lorraine & CRIN, 615 rue du jardin botanique, BP 101, 54602 Villers les Nancy Cedex, France, email: `Claude.Kirchne@loria.fr`.

Order-Sorted Feature Theory Unification

Hassan Ait-Kaci[1]

Order-sorted seature (OSF) terms generalize first-order rational terms whereby constructors become partially ordered sorts, and argument positions become symbolic feature symbols. Matching and unification are extended accordingly making OSF terms a more versatile data structure for symbolic programming than first-order terms, where unification is akin to inheritance among objects in object-oriented programming. In order to obtain the functionality of the latter more adequately, one also needs the notion of class or object template. This can be rendered very precisely as an OSF theory in the form of a monotonic mapping from sorts to OSF terms imposing structural constraints on objects using certain sorts. We will show how this may be done quite precisely and simply by achieving the template effect of a class hierarchy through a remarkably efficient unification algorithm modulo such an OSF theory. This algorithm does in effect lazy inheritance, importing a minimal amount of information from the theory to the OSF formula being normalized. Recursive classes, self-referring classes, and class attribute coreference are readily accommodated. The scheme extends also to disjunctive theories.

[1]Digital, Paris Research Laboratory, 85 Avenue Victor Hugo, F-92563 Rueil Malmaison Cedex, France, email: `hak@prl.dec.com`.

Combination Techniques and Decision Problems for Disunification

Franz Baader,[1] Klaus U. Schulz[2]

Abstract. Previous work on combination techniques considered the question of how to combine unification algorithms for disjoint equational theories $E_1, \ldots, E_n$ in order to obtain a unification algorithm for the union $E_1 \cup \ldots \cup E_n$ of the theories. Here we want to show that variants of this method may be used to decide solvability and ground solvability of disunification problems in $E_1 \cup \ldots \cup E_n$. Our first result says that solvability of disunification problems in the free algebra of the combined theory $E_1 \cup \ldots \cup E_n$ is decidable if solvability of disunification problems with linear constant restrictions in the theories E_i $(i = 1, \ldots, n)$ is decidable. In order to decide ground solvability (i.e., solvability in the initial algebra) of disunification problems in $E_1 \cup \ldots \cup E_n$ we have to consider a new kind of subproblem for the particular theories E_i, namely solvability of disunification problems with linear constant restriction under the additional constraint that values of variables are not E_i-equivalent to variables. The correspondence between ground solvability and this new kind of solvability holds, *(1)* if one theory E_i is the free theory with at least one function symbol and one constant, or *(2)* if the initial algebras of all theories E_i are infinite. Our results can be used to show that the existential fragment of the theory of the (ground) term algebra modulo associativity of a finite number of function symbols is decidable; a similar result follows for function symbols which are associative and commutative, or associative, commutative and idempotent.

1 Introduction and Formal Preliminaries

In recent years the rôle Robinson unification—and later unification modulo equational theories—played in theorem proving, term rewriting, and logic programming has more and more been taken on by constraint solving (see e.g., [Bü90, KK89, JL87, Col90]). One advantage of constraint approaches is that it is no longer necessary to compute (a complete set of) solutions; deciding satisfiability of the constraints is usually sufficient. Thus one can, for example, work modulo non-finitary equational theories such as associativity.

[1] DFKI, Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany,
Email: `baader@dfki.uni-sb.de`.
[2] CIS, University of Munich, Leopoldstr. 138, 8000 Munich 40, Germany
Email: `schulz@sun1.cis.uni-muenchen.de`.

Another benefit of using a constraint approach is that it is rather natural to enhance the expressive power by considering more general constraints than the equality constraints of unification problems. One of the earliest of these generalizations was Colmerauer's use of equations and negated equations in PROLOG II [Col84]. In the present paper we shall consider solvability of this kind of equational problems (subsequently called disunification problems) modulo equational theories.

We shall introduce a method to decide solvability of disunification problems which contain function symbols whose properties are defined by equational theories with disjoint signatures. For pure unification problems, such combination methods have been extensively studied (see, e.g., the introduction of [BS91a] for a brief overview), but until now these approaches have not been generalized to the disunification case. We shall consider two notions of solvability, solvability in the initial algebra (called *ground solvability*) and solvability in the free algebra (simply called *solvability*). Both types of solvability are used in the literature (see [Com90, Bü88]), but ground solvability seems to be more interesting for most applications.

In order to stress the signature over which the terms in the formulation of a disunification problem and in the solutions of the problem may be built we shall talk about (E, Σ)-*disunification* problems, where Σ is a finite superset of $sig(E)$, the signature of E. Such a problem is a finite set of equations and disequations

$$\Gamma = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\} \cup \{s_{n+1} \not\doteq t_{n+1}, \ldots, s_{n+m} \not\doteq t_{n+m}\},$$

where $s_1, \ldots, t_{n+m}$ are Σ-terms. A *solution* of the (E, Σ)-disunification problem Γ is a Σ-substitution σ such that $s_i\sigma =_E t_i\sigma$ $(i = 1, \ldots, n)$ and $s_{n+j}\sigma \neq_E t_{n+j}\sigma$ $(j = 1, \ldots, m)$. A *ground solution* is a solution that maps all variables occurring in Γ to variable-free Σ-terms.

The (E, Σ)-disunification problem is called *elementary*, if $\Sigma = sig(E)$; it is a disunification problem *with constants*, if $\Sigma \setminus sig(E)$ is a finite set of constants; and it is a *general* disunification problem, if no such restrictions hold.

In order to formulate our results we have to consider the following generalization of a disunification problem with constants: an *E-disunification problem with linear constant restriction* consists of two parts:

1. An $(E, sig(E) \cup C)$-disunification problem Γ, where C is a finite set of constant symbols not occurring in $sig(E)$, and

2. a linear ordering $<$ on $C \cup X$, where X is a finite superset of the set of variables occurring in Γ.

For a given problem of this kind, the sets V_c of *variables which may not use c* are defined as $V_c = \{x \in X;\ x < c\}$, for every $c \in C$. A *solution* of this problem is a substitution σ which assigns terms $x\sigma$ built with variables, symbols from $sig(E)$, and constants in C to the variables $x \in X$, solves all equations and disequations of Γ modulo E, and has the additional property that c does not occur in $x\sigma$ for all $c \in C$ and $x \in V_c$. A solution σ is called *restrictive* if for all variables $x \in X$ the value $x\sigma$ is not E-equivalent to a variable.

2 Main Results

Our first result says that solvability of disunification problems in the combination of disjoint equational theories can be reduced to solvability of disunification problems with linear constant restriction in the single theories.

Theorem 2.1 (Solvability) *Let $E_1, \ldots, E_n$ be equational theories over disjoint signatures such that solvability of disunification problems with linear constant restriction is decidable for $E_1, \ldots, E_n$. Then solvability of elementary disunification problems is decidable for the combined theory $E_1 \cup \ldots \cup E_n$.*

This result is analogous to the one for unification given in [BS91a], and it depends on a decomposition algorithm (given in [BS92]) which is very similar to the algorithm presented in that paper. However, the proof of soundness of the method is more complex.

Corollary 2.1 *(1) Let E be an equational theory such that solvability of disunification problems with linear constant restriction is decidable. Then solvability of general E-disunification problems is decidable.*
(2) The assumptions of Theorem 2.1 are sufficient to get decidability of general disunification problems in the combined theory.
(3) If, for E_1 and E_2, solvability of disunification problems with linear constant restriction can be decided by an NP-algorithms, then solvability of disunification problems in the combined theory is also NP-decidable.

For ground solvability we arrive at a distinct characterization.

Theorem 2.2 (Ground Solvability) *Let $E_i, i = 1, \ldots, n$, be equational theories over disjoint signatures Σ_i, and suppose that the initial algebras $T(\Sigma_i, \emptyset)/{=_{E_i}}$ are infinite. If restrictive solvability of E_i-disunification problems with linear constant restriction is decidable for $i = 1, \ldots, n$, then ground solvability of disunification problems is decidable for $E_1 \cup \ldots \cup E_n$.*

Sometimes the condition that the initial algebras are infinite can be dropped: we call an equational theory F the free theory with signature Σ iff $sig(F) = \Sigma$ and $=_F$ is just the syntactic equality of terms.

Corollary 2.2 *Let $\Sigma_1, \ldots, \Sigma_n$ be disjoint signatures, $E_1, \ldots, E_{n-1}$ be equational theories over $\Sigma_1, \ldots, \Sigma_{n-1}$, and let E_n be the free theory with signature Σ_n. Assume that Σ_n contains at least one function symbol of arity greater zero and one constant. Then ground solvability of disunification problems in $E_1 \cup \ldots \cup E_n$ is decidable if restrictive solvability of E_i-disunification problems with linear constant restriction is decidable for $i = 1, \ldots, n-1$.*

The method can be applied to the combination of A, AC, ACI, and free theories.[3]

[3]An equational theory is called an A-theory iff its signature consists of a binary function symbol h, and it contains the single axiom $h(h(x, y), z) = h(x, h(y, z))$ (associativity). For AC-theories, one has an additional axiom $h(x, y) = h(y, x)$ (commutativity), and for ACI-theories there is a third axiom $h(x, x) = x$.

Theorem 2.3 *Solvability of disunification problems is decidable for every theory which is a disjoint combination of finitely many A-, AC-, and ACI-theories and a free theory. To get decidability of ground solvability by our method we have to assume that the free theory contains at least one constant symbol and one function symbol of arity greater than* 0.

Since existential equational formulae can be seen as disjunction of disunification problems we have the following immediate consequence of the theorem.

Corollary 2.3 *Let* Σ *be a signature consisting of* $n \geq 1$ *binary function symbols* $h_1, \ldots, h_n$, *and at least one constant and one additional non-constant function symbol. Let* A_n, AC_n, *and* ACI_n *respectively stand for associativity, associativity and commutativity, and associativity, commutativity and idempotency of the function symbols* h_i. *(1) The existential theories of the free algebra* $T(\Sigma, Y)/{=_{A_n}}$ *and the initial algebra* $T(\Sigma, \emptyset)/{=_{A_n}}$ *are decidable. (2) The existential theories of the free algebra* $T(\Sigma, Y)/{=_{AC_n}}$ *and the initial algebra* $T(\Sigma, \emptyset)/{=_{AC_n}}$ *are NP-decidable. (3) The existential theories of the free algebra* $T(\Sigma, Y)/{=_{ACI_n}}$ *and the initial algebra* $T(\Sigma, \emptyset)/{=_{ACI_n}}$ *are NP-decidable.*

For AC, decidability has already been shown by Comon [Com88]. The result for A seems to be new. There is no real hope to extend these decidability results to equational formulae with more complex quantifier prefix. A recent result by Treinen [Tr92] shows that already the Σ_2 fragment[4] of the theory of the ground term algebra modulo A is undecidable. For AC, Treinen shows that the Σ_3-fragment is undecidable, both for the free algebra and the initial algebra.

References

[BS91a] F. Baader, K.U. Schulz. Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures. *DFKI-Research Report RR-91-33*, also in *Proceedings of the 11th International Conference on Automated Deduction*, subseries on AI 607, 50-65, 1991.

[BS91b] F. Baader, K.U. Schulz. General A- and AX-Unification via Optimized Combination Procedures. To appear in the *Proceedings of Second Workshop on Word Equations and Related Topics IWWERT '91*, *LNCS*, Rouen 1991.

[BS92] F. Baader, K.U. Schulz. Combination Techniques and Decision Problems for Disunification. To appear as *DFKI-Research Report*, German Research Center for Artificial Intelligence, Saarbrücken 1992.

[Bü88] H.J. Bürckert. Solving Disequations in Equational Theories. *Proc. 9th Conf. on Automated Deduction, Argonne*, *LNCS*

310, 1988.

[Bü90] H.-J. Bürckert. A Resolution Principle for Clauses with Constraints. *Proceedings of the 10th International Conference on Automated Deduction*, *LNCS* **449**, 1990.

[4] consisting of the closed formulae with quantifier prefix of the form $\exists\vec{x}\forall\vec{y}$

[Col84] A. Colmerauer. Equations and Inequations on Finite and Infinite Trees. In *FGCS'84, Proceedings*, 85-99, November 1984.

[Col90] A. Colmerauer. An Introduction to PROLOG III. *C. ACM*, **33**, 1990.

[Com88] H. Comon. Unification and Disunification. Théorie et Applications. *PhD Thesis, Institut National Polytechnique de Grenoble*, Grenoble, France, 1988.

[Com90] H. Comon. Disunification: a Survey. *Rapport de Recherche 540, L.R.I*, Universite de Paris-Sud, France, 1988.

[DJ87] N. Dershowitz, J.P. Jouannaud. Rewrite System. *Handbook of Theoretical Computer Science, Vol. B*, North-Holland, 244-320, 1990.

[JK90] J.P. Jouannaud, C. Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. Preprint, 1990. To appear in the Festschrift to Alan Robinson's birthday.

[JL87] J. Jaffar, J.L. Lassez. Constraint Logic Programming. *Proceedings of 14th POPL Conference*, Munich, 1987.

[KN91] D. Kapur, P. Narendran. Complexity of Unification Problems with Associative-Commutative Operators. Preprint, 1991. To appear in *J. Automated Reasoning*.

[KK89] C. Kirchner, H. Kirchner. Constrained Equational Reasoning. *Proceedings of SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, ACM Press, 1989.

[SS89] M. Schmidt-Schauß. Combination of Unification Algorithms. *J. Symbolic Computation* **8**, 1989.

[Tr92] R. Treinen. A New Method for Undecidability Proofs of First Order Theories. To appear in the *J. Symbolic Computation.*.

A Complete and Decidable Feature Theory

Rolf Backofen and Gert Smolka[1]

Various feature descriptions are being employed in constrained-based grammar formalisms. The common notational primitive of these descriptions are functional attributes called features. The descriptions considered in this paper are the possibly quantified first-order formulae obtained from a signature of features and sorts. We establish a complete first-order theory FT by means of three axiom schemes and construct three elementarily equivalent models.

One of the models consists of so-called feature graphs, a data structure common in computational linguistics. The other two models consist of so-called feature trees, a record-like data structure generalizing the trees corresponding to first-order terms.

Our completeness proof exhibits a terminating simplification system deciding validity and satisfiability of possibly quantified feature descriptions.

[1]German Research Center for Artificial Intelligence Stuhlsatzenhausweg 3, D-6600 Saarbrücken 11, Germany, email: `backofen,smolka@dfki.uni-sb.de`.

Complexity Results for Difference Unification

David A. Basin[1] and Toby Walsh[2]

1 Introduction

In [4] we present a general procedure called *difference unification* for identifying differences between two terms or formulas. Difference unification extends unification in that it decides if terms are syntactically equal not only by giving assignments for variables but also by computing what incompatible term structure must be removed. This incompatible term structure, called *wave-fronts* is marked by sets of *annotations* which are used to direct a special kind of rewriting called *rippling*. Rippling seeks to reduce the differences between the terms by moving the wave-fronts "out of the way" while not disturbing the unannotated parts of the terms.

Difference unification and rippling have proved very successful in several different areas of mathematics: in inductive theorem proving [1, 2], summation [7], inductive completion and normalization [4]. For example, if we wish to prove $p(x)$ for all natural numbers using induction, we assume $p(n)$ and attempt to show $p(s(n))$. The hypothesis and the conclusion are identical except for the successor function $s(.)$ applied to the induction variable n. This wave-front is marked by placing a box around it and underlining the *wave-hole* the subterm contained in the hypothesis: $p(\boxed{s(\underline{n})})$. Rippling then attempts to move this difference out of the way leaving behind the induction hypothesis.

2 Difference unification

To specify difference unification we must be more precise about the representation of wave-fronts and annotations. As in [3] wave-fronts are assumed to be in a normal form in which every wave-front has an immediate subterm deleted (i.e. all wave-fronts are one functor thick). In addition, rather than superimposing a particular representation on terms (like the "box-and-hole" notation used in the introduction and in [2]), annotations

[1]Max-Planck-Institut für Informatik, Im Stadtwald, Saarbrücken, Germany. Email:basin@mpi-sb.mpg.de

[2]Department of AI, University of Edinburgh, 80 South Bridge, Edinburgh, Scotland. Email:tw@aisb.ed.ac.uk

will be abstracted out and represented separately; this makes it much easier to specify difference unification. Annotations will therefore be represented by the set of positions of the wave-holes; as the wavefronts are always one functor thick, the position of the wave-hole uniquely determines the wavefront. Positions are defined recursively as follows: Λ is the root, and the set of positions in the term t is $Pos(t)$ where,

$$Pos(f(s_1, ..., s_n)) \quad = \quad \{\Lambda\} \cup \{i.p \mid 1 \leq i \leq n \ \wedge \ p \in Pos(s_i)\}$$

By recursion on terms it is simple to define a function $skeleton(t, A_t)$ which takes a term t and a set of annotations for that term A_t, and returns the unannotated part of the term. For example, the skeleton of $f(g(\boxed{f(\underline{a}, b)}), \boxed{g(\underline{b})})$ is $f(g(a), b)$.

Difference unification is then a relation written $du(s, t, A_s, A_t, \sigma)$ that satisfies the property

$$\sigma(skeleton(s, A_s)) \quad = \quad \sigma(skeleton(t, A_t)),$$

where σ is a most general unifier. Note that this is rather different from the much harder homomorphic embedding problem [6] where the substitution is applied before deleting function symbols possibly including those introduced by the substitution. Trivially, difference unification is a generalisation of matching and unification. In [4] we give a set of transformation rules for difference unification which we prove are sound, complete and terminating.

3 Complexity

A naive implementation of difference unification which generates (annotations) and tests (for unifiability) is exponential since a term of size n (n function symbols) has $O(n)$ interior (neither constants or variables) function symbols that can be hidden in $O(2^n)$ different ways. It is natural to ask whether this the best that we can do, and whether there are more tractible cases. In asking such questions we must distinguish between the problem of generating all solutions and that of generating a solution or knowing if one exists. The first problem is easily seen to require exponential size and space even in the very restricted of case of ground difference matching.

Theorem 1 *There are difference matching problems requiring exponential time and space.*

Proof: Consider difference matching $\langle s, t\rangle$ where $s = f^n(a)$ and $t = f^{2n}(a)$ (f^m(a) is the m-fold application of f to a). The solutions correspond to choosing n out of $2n$ occurrence of f in t to hide. That is, $(2n!)/((n!)^2)$ which is $O(2^n)$. $\Box$

Problems generating exponential numbers of solutions are exceptional as they involve unusual amount of repeated structure. In general, there are far fewer matches and unifiers; so it is interesting to investigate the complexity of returning a single solution, or determining if one exists. Below we show that, in the ground case, determining the existence of a solution to difference matching or difference unification is polynomial time decidable. The algorithms given are based on dynamic programming and can be easily modified to return solutions as well as indicate their existence or non-existence. After, we show that when variables are admitted, the problem becomes NP complete.

3.1 Ground difference matching and unification

Theorem 2 *Given terms s and t we can determine if s difference matches t (s may be annotated with skeleton t) or s difference unifies with t in polynomial time.*

Proof: For difference unification, consider the following rewrite rules on pairs of terms.

$$
\begin{array}{lll}
\langle t,t\rangle & \rightarrow & TRUE \\
\langle a,b\rangle & \rightarrow & FALSE \text{ (for } a \neq b \text{ and } a,b \text{ atoms)} \\
\langle h(s_1,....,s_k),h(t_1,...,t_k)\rangle & \rightarrow & \langle s_1,h(t_1,...,t_k)\rangle \vee ... \vee \langle s_k,h(t_1,...,t_k)\rangle \vee \\
 & & \langle h(s_1,...,s_k),t_1\rangle \vee ... \vee \langle h(s_1,...,s_k),t_k\rangle \vee \\
 & & (\langle s_1,t_1\rangle \wedge ... \wedge \langle s_k,t_k\rangle) \\
\langle h(s_1,....,s_k),t\rangle & \rightarrow & \langle s_1,t\rangle \vee ... \vee \langle s_k,t\rangle \\
\langle s,h(t_1,...,t_k)\rangle & \rightarrow & \langle s,t_1\rangle \vee ... \vee \langle s,t_k\rangle
\end{array}
$$

If we apply these rewrite rules deterministically, given priority in the order listed above, it is not difficult to see see that there is a rewriting of $\langle s,t\rangle$ to $TRUE$ iff s difference unifies with t.

Whilst naive application of these rewrite rules results in an exponential amount of computation we can do better. In particular, if we assume some fixed maximal arity for the functions in the signature, each reduction can be computed in constant time given truth values for the subproblems. Without this assumption, the bound becomes linear in the size of the two terms. Notice that there are only $|s| * |t|$ subproblems, corresponding to the cartesian product of subterms from s and t. If we use dynamic programming to compute these in a sensible order, or alternatively use a memo function with constant lookup time, then the overall complexity is $O(|s| * |t|)$.

The rules are easily modified for the ground difference matching problem: change b to a term t in the second rule, delete the $\langle h(s_1,...,s_n),t_i\rangle$ disjuncts in the third rule, and delete the final rule entirely. Note that this ground difference matching algorithm can be used to solve the homeomorphic embedding problem of one ground term into another in polynomial time and is similar to the algorithm of [6] □

As a side note, observe that while the above ground difference unification algorithm can be easily modified to yield minimal answers[3], there is a trivial linear time algorithm for determining difference unifiability although it does not give minimal answers; two terms will difference unify iff they share at least one constant (of arity 0). In the non-ground case, two terms will difference unify iff they share one constant or if either contains a variable. In this respect, difference unification is, perhaps surprisingly, easier than difference matching.

3.2 Difference unification with variables

Difference unification and all its subproblems are trivially in NP since we can guess annotations and then unify or match resulting skeletons in polynomial time. To show NP completeness, we prove that when variables are added determining the existence of a solution is NP hard.

[3]That is, those with the least amount of annotation necessary to ensure unifiability

Theorem 3 *Difference unifying s and t, with annotation on one and only one side is NP hard.*

Proof: Assume we only allow annotation on the second term t (i.e. delete one of the two hiding rules). We reduce 3SAT to this restricted difference unification problem by the following construction which has similarities to one used in [5]. Let $C = \{c_1, c_2, ..., c_m\}$ be an instance of 3SAT over the boolean variables $x_1, ..., x_n$. We construct two terms s and t to difference unify where s represents the clauses and t the satisfying assignments.

The term signature is as follows. Corresponding to each clause c_i is the distinct ternary function g_i. We also employ the m-ary function h and the 7ary f and the constants 0 and 1. Further, let V, the set of variables be $\{x_1, ..., x_n\}$. We intend that the truth or falsity of a boolean variable x_i is simulated by $x_i = 1$ and $x_i = 0$ respectively. For each clause c_j, do the following: let x_1, x_2, x_3 be the variables in c_j. There are exactly 7 sets of truth-value-assignments that make clause c_j true. Define 7 distinct terms $q_{j1}, ..., q_{j7}$ as follows: $q_{ji} = g_j(b_1, b_2, b_3)$ where $b_i \in \{0, 1\}$ and the assignment (b_1, b_2, b_3) satisfies c_j. Now let $s_j = g_j(x_1, x_2, x_3)$ and $t_j = f(q_{j1}, ..., q_{j7})$. Note that if s_j difference unifies with t_j than a solution yields a substitution that is a satisfying assignment for clause c_j.

Now form terms s and t as follows: $s = h(s_1, ..., s_m)$ and $t = h(t_1, ..., t_m)$. Observe that the solution to difference matching must pick, for each j, exactly one q_{ji} to unify with the variables in clause c_j and the combination of these is a satisfying assignment. Hence if there exists a solution, the clauses are satisfiable.

Since the reduction is trivially polynomial time computable we have shown NP-hardness. □

4 Conclusions

We have shown that difference unification with variables is NP complete. Fortunately this result is not as disappointing as it first appears. In [4], we show how minimal difference unifications can often be found quickly (using a new search strategy called *left-first search*), and that these answers are usually the ones required for rippling. The exponential factor in difference unification is in the potentially exponential amount of annotation required; with minimal difference unifications, we attempt to minimize the amount of annotation returned.

Acknowledgements

This research was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102 and a SERC postdoctoral fellowship. The responsibility for the contents of this chapter lies with the authors. We thank Paliath Narendran and the Edinburgh Mathematical Reasoning Group for their encouragement and criticism.

References

[1] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. newblock Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, 1991.

[2] Alan Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference On Automated Deduction*, pages 111–120, Argonne, Illinois, 1988.

[3] David Basin and Toby Walsh. Difference matching. In *Proc. of 11th International Conference On Automated Deduction (CADE-11)*, pages 295–309, Saratoga Springs, New York, June 1992. Springer-Verlag.

[4] David Basin and Toby Walsh. Difference unification. Technical Report MPI-I-92-247, Max-Planck-Institute für Informatik, 1992.

[5] Deepak Kapur and Paliath Narendran. NP-completeness of the set unification and matching problems. In *8th International Conference On Automated Deduction*, pages 489–495, Oxford, UK, 1986.

[6] Paliath Narendran and Jonathan Stillman. In *Fifth International Conference on Applied Algebra and Error Correcting Codes*, Menorca, Spain, 1987.

[7] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 325–339. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.

Ordered Theory Resolution and Partial Unification

Peter Baumgartner[1]

1 Introduction

Theory reasoning ([Sti85]) means to relieve a calculus from explicit reasoning in some domain (e.g. equality, partial orders) by taking apart the domain knowledge and treating it by special inference rules. In an implementation, this results in a universal "foreground" reasoner that calls a specialized "background" reasoner for theory reasoning. More technically, the background reasoner has to implement a unification procedure for the theory. Since for the theory we want to allow arbitrary Horn theories, the theory unification procedure operates on the *literal* level rather than on the term level. We are interested in the design of such unification procedures, which moreover take advantage of ordering restrictions as commonly applied in the term rewriting paradigm (see [Bac91]).

2 Ordered Theory Resolution

For the foreground calculus we will use resolution. Additionally a partial ordering on the literals is used to disallow resolving two clauses which violate certain maximality constraints of the selected literals. The partial ordering on literals must contain a total — but not neccessarily noetherian — ordering '$\succ$' on ground literals.

Then the main inference rule of our *total ordered theory resolution calculus* in its ground version is as follows:

Total ordered theory resolution:

$$\frac{L_1 \vee R_1 \quad \dots \quad L_n \vee R_n}{R_1 \vee \dots \vee R_n} \qquad \left\{ \begin{array}{ll} \text{if 1.} & \{L_1, \dots, L_n\} \text{ is} \\ & \text{theory-unsatisfiable.} \\ \text{and 2.} & L_i \text{ is the biggest literal} \\ & \text{wrt. } \succ \text{ in } L_i \vee R_i \\ & (\text{for } i = 1 \dots n) \end{array} \right.$$

[1]Institut für Informatik, Universität Koblenz, Rheinau 3-4, W-5400 Koblenz, Germany, Email: peter@informatik.uni-koblenz.de.

Lifting of the inference rules to the first order level is done with respect to the following adaption of theory unification to our needs:

Definition 2.1 Let $\mathcal{T}$ be the axioms of a theory, given by a consistent set of clauses. Let $S = \{L_1, \ldots, L_n\}$ be a literal set. S is called *$\mathcal{T}$-complementary* iff the clause $\overline{L_1} \vee \ldots \vee \overline{L_n}$ is $\mathcal{T}$-valid. A set of substitutions is called a *complete set of $\mathcal{T}$-unifiers for S* (or short: $CSU_{\mathcal{T}}(S)$) iff

1. for all $\sigma \in CSU_{\mathcal{T}}(S)$: $S\sigma$ is $\mathcal{T}$-complementary. *(Correctness)*
2. for all substitutions θ such that $S\theta$ is $\mathcal{T}$-complementary: there exists a $\sigma \in CSU_{\mathcal{T}}(S)$ and a substitution σ' such that $\theta|var(S) = (\sigma\sigma')|var(S)$ *(Completeness)*

This notion of theory unifier generalizes the notion of *rigid E-unifier* ([GNPS90]) to more general theories than equality (see ([Bau92])).

When an additional factoring rule (it suffices to factor on maximal literals) is added, the resulting calculus is refutational complete (see [Bau92] for a full first order version).

The problem in implementing (ordered) total theory resolution is that in general it cannot be predicted what literals and how many variants of them constitute a contradictory set. As a solution, partial theory reasoning tries to break the "big" total steps into more managable smaller steps. For this purpose the *partial ordered theory resolution calculus* needs an additional inference rule:

Partial ordered theory resolution:

$$\frac{L_1 \vee R_1 \quad \ldots \quad L_n \vee R_n}{Res \vee R_1 \vee \ldots \vee R_n} \quad \left\{ \begin{array}{rl} \text{if 1.} & \{L_1, \ldots, L_n, \overline{Res}\} \text{ is} \\ & \text{theory-unsatisfiable.} \\ \text{and 1'.} & L_i \succ Res \text{ for some} \\ & i \in \{1 \ldots n\}. \\ \text{and 2.} & L_i \text{ is the biggest literal} \\ & \text{wrt. } \succ \text{ in } L_i \vee R_i \\ & (\text{for } i = 1 \ldots n) \end{array} \right.$$

Condition 1 guarantees the soundness of the rule: the "residue" Res is a logical consequence of $\{L_1, \ldots, L_n\}$. Operationally, the residue is a new subgoal to be proved. For example, if $L_1 = a \leq b$, $L_2 = b \leq c$ then by transitivity of $\leq$ (assume $\leq$ is interpreted as a partial ordering) one possible residue is $Res = a \leq c$ if additionally the ordering restriction 1.$'$ is satisfied.

3 Partial Unification

In the *total* calculus a theory-unsatisfiable literal set $S = \{L_1, \ldots, L_n\}$ has to be found in every inference step; the strategy for refutations in the *partial* calculus is to refute this set S in a chain of partial steps, concluded by a "trivial" total step.

In every partial step a residue is computed from a subset of the input set S which is then joined to S. There is a suitable framework at hand for expressing such refutations, namely *unit resulting resolution* ([MOW76])[2]. In order to apply unit resulting resolution for our purposes we take the following view: an axiom of the theory $L_1 \wedge \ldots \wedge L_n \rightarrow L_{n+1}$ is mapped to a clause $\overline{L_1} \vee \ldots \vee \overline{L_n} \vee L_{n+1}$. Additionally the resulting clause set $\mathcal{C}$ has to be closed under resolution with unit literals from $\mathcal{C}$, i.e. if there exists a literal $L \in \mathcal{C}$ and a clause $\overline{L} \vee C \in S$ then $C \in \mathcal{C}$. This closure has to be carried out since in unit resulting steps with a clause $L_1 \vee \ldots \vee L_n$ from $\mathcal{C}$ the $n-1$ literals shall be saturated by input literals only, but not by literals from $\mathcal{C}$. Thus, the potential use of literals from $\mathcal{C}$ in unit resulting steps has to be carried out "before". It should be noted that this closure can result in infinite clause sets, as e.g. $\{P(a), P(x) \rightarrow P(f(x))\}$ yields $P(f(a)), P(f(f(a))), \ldots$.

The resulting (possibly infinite) clause set does not allow for order restricted refutations. By an order restricted refutation we mean a refutation where (speaking ground) the resulting unit literal is smaller than the biggest premise literal. As a typical example consider[3]

$$\mathcal{C} = \{A \rightarrow B, B \wedge C \rightarrow D\}$$

Let $S = \{A, C, \neg D\}$. S is unsatisfiable in the theory given by $\mathcal{C}$. Furthermore suppose an ordering with $B \succ A \succ C \succ D$. There exists a unit-resulting refutation where from A and $A \rightarrow B$ the literal B is derived. However this inference violates the ordering restriction. Now, this "peak" in the derivation can be detoured by a new clause that stems from a critical pair (much like in Knuth-Bendix completion): one has to consider all overlaps of $A \rightarrow B$ with clauses containing B in the premise, which is here $B \wedge C \rightarrow D$. Then we generate a new rule in the following way:

$$\begin{array}{cccccccc} A & \rightarrow & B & & & & \\ & & B & \wedge & C & \rightarrow & D \\ \hline A & & & \wedge & C & \rightarrow & D \end{array}$$

In general all such order-violating situations have to be considered and turned into new rules. In order to come to finite systems more often, one needs redundancy criterion that allow to delete clauses. Besides tautology and subsumption tests the most powerful redundancy criterion is the following: a clause is redundant if its use in a derivation can be simulated by a (finite) derivation using the other clauses.

The generalization to the first order level is straightforward. Concerning the ordering, additionally to the requirements from above it must be noetherian.

In summary, by this completion technique we can combine an ordering strategy together with a unit-resulting strategy for Horn clauses — two strategies that are not complete when naively combined. The technique means for the partial theory reasoning calculus a combination of ordered unit resulting resolution and ordinary ordered resolution. As an example for a finite system we can mention a system for the theory of strict orderings together with an equivalence relation.

[2]In unit-resulting resolution a resolvent from a n-literal clause and $n-1$ or n literals is built by simultaneously resolving upon the $n-1$ or n literals against the n-literal clause. Thus the resolvent is either a single literal or the empty clause. Unit resulting resolution is complete for Horn clauses

[3]Clauses are written as implications again for easier reading

References

[Bac91] L. Bachmair. *Canonical Equational Proofs.* Progress in Theoretical Computer Science. Birkhäuser, 1991.

[Bau92] P. Baumgartner. An Ordered Theory Resolution Calculus. In A. Voronkov, editor, *Logic Programming and Automated Reasoning (Proceedings)*, pages 119–130, St. Petersburg, Russia, July 1992. Springer. LNAI 624.

[GNPS90] J. Gallier, P. Narendran, D. Plaisted, and W. Snyder. Rigid E-unification: NP-Completeness and Applications to Equational Matings. *Information and Computation*, pages 129–195, 1990.

[MOW76] J. McCharen, R. Overbeek, and L. Wos. Complexity and related enhancements for automated theorem-proving programs. *Computers and Mathematics with Applications*, 2:1–16, 1976.

[Sti85] M. E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, pages 333–356, 1985.

On Unification of Terms with Integer Exponents (Abstract)

Hubert Comon [1]

1 Introduction

H. Chen and J. Hsiang proposed in [1] a unification algorithm for what they call ρ-terms (with intended applications to logic programming). These terms allow to iterate terms with one hole along fixed paths. The number of iterations is part of the syntax of the terms and may include integer variables.

Example 1.1
Let the alphabet of functions symbols be f (binary), g (unary) and a (constant). Let N be an integer variable. Then, in Chen and Hsiang's formalism, $\Phi(f(\diamond, a), N, g(a))$ is a typical ρ-term whose instances (obtained by replacing N with an actual non-negative integer) are $g(a), f(g(a), a), f(f(g(a), a), a), \ldots$. The term $f(\diamond, a)$ has one "hole" (the $\diamond$) and its iteration along the path 1 (which is the position of the hole) is allowed.

Such constructions can be useful in expressing infinite sets of terms which occur in logic programming (see [1]) or in the Knuth-Bendix completion procedure (see [2]). In both cases, the ability to express iterated terms can prevent the non termination of the deduction process. Other applications are currently under investigation, for example in unification theory (see [5])

Constructions involving more than one occurrence of an integer variable such as

$$f(\Phi(f(\diamond, a), N, g(a)), \Phi(f(\diamond, a), N, f(a, a)))$$

are also allowed in [1] which shows that the ρ-terms can schematize non regular sets of ground terms and hence they are not encompassed by the study of terms with context variables investigated in [3]. On the other hand, ρ-terms do not have the power of regular languages; there are two restrictions in these ρ-terms: nested ρ-terms are forbidden and the iterated part should not itself contain ρ-terms. Finally, in [1], the iterated parts and the terms in the holes should not contain variables.

[1]Laboratoire de Recherche en Informatique, Bâtiment 490, Université de Paris-Sud 91405 ORSAY cedex, France, Email: `comon@lri.fr`.

Example 1.2 (see also the figure on the following page)

$\Phi(f(\diamond, a), N, \Phi(g(\diamond), M, a))$ is not a ρ-term. (Nested ρ-terms are forbidden)

$\Phi(f(\diamond, \Phi(f(\diamond, a), N, a)), M, g(a))$ is not a ρ-term (the iterated part itself should not contain a ρ-term).

$\Phi(f(x, \diamond), N, g(a))$ is not a ρ-term (variables are not allowed in the iterated part)

$\Phi(f(a, \diamond), N, g(x))$ is not a ρ-term (variables are not allowed "below" the iterated part, i.e. in the third positions of the Φ construction).

In this note we give another unification algorithm where these restrictions are dropped. We keep however the condition that, along an iterated path there should not occur any ρ-term. (i.e., for example, $\Phi(\Phi(f(\diamond, a), N, g(\diamond)), M, g(a))$ is still not allowed). Indeed, without this restriction, unification becomes undecidable.

2 I-terms

Missing definitions can be found in [JK91, 4]. We assume that F is a (finite or infinite) set of function symbols together with the arity function a. F is assumed to contain at least one constant (i.e. a symbol of arity 0). X is an infinite set of constants (disjoint from F) called *variables*. $V_{\mathbf{N}}$ is a fixed set of symbols denoting integer variables. The set of *I-terms* T (also called "terms" for short) is the least set that satisfies:

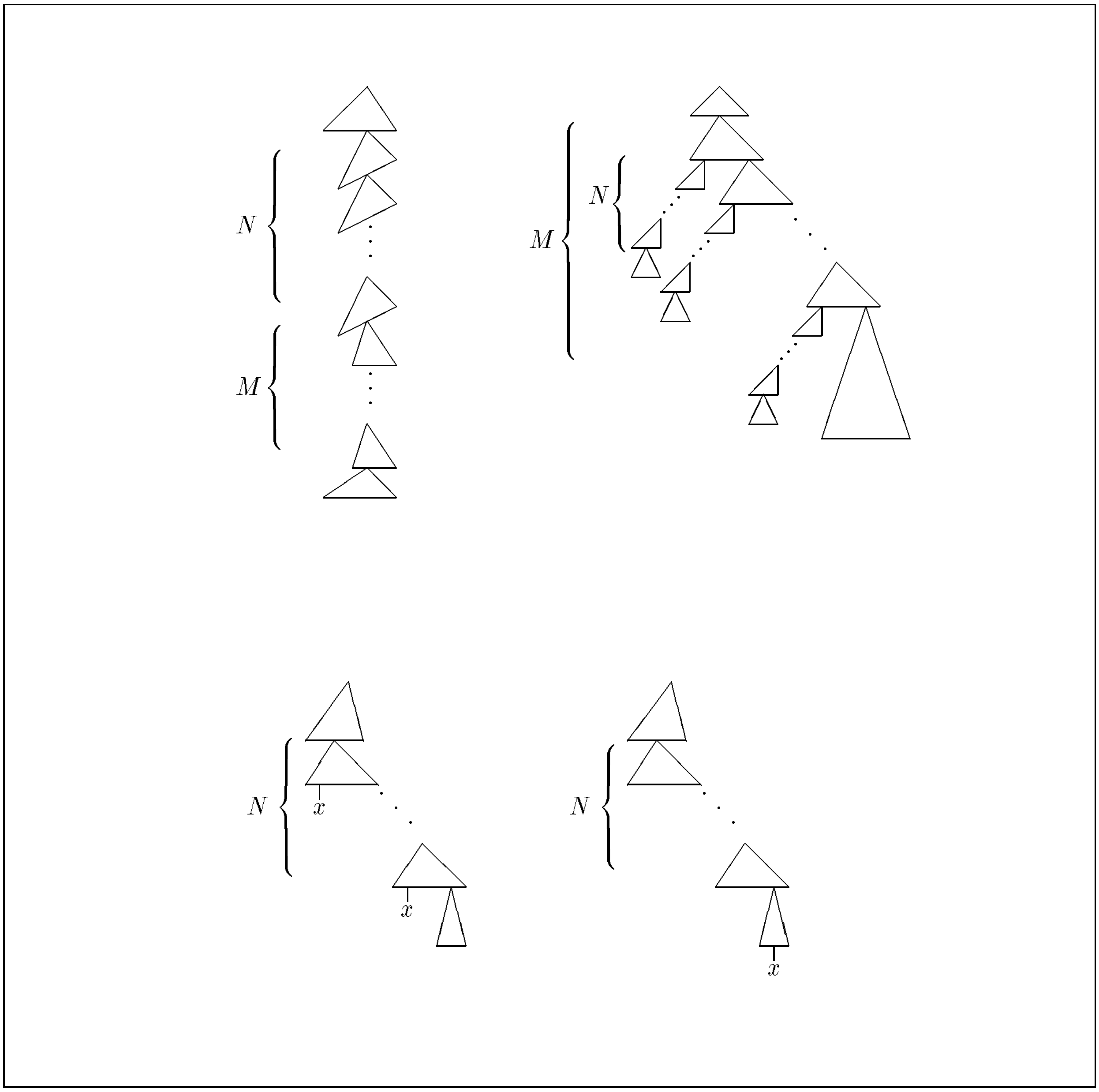
$$\begin{cases} f(\vec{s}) \in T & \Leftarrow & \vec{s} \in T^{a(f)} \\ x \in T & \Leftarrow & x \in X \\ s[]_p^N \cdot t \in T & \Leftarrow & s, t \in T,\ p \in Pos(s), N \in V_{\mathbf{N}} \\ Pos(x) & \stackrel{\text{def}}{=} & \{\Lambda\} \\ Pos(f(\vec{s})) & \stackrel{\text{def}}{=} & \{\Lambda\} \cup 1 \cdot Pos(s_1) \cup \ldots \cup a(f) \cdot Pos(s_{a(f)}) \\ Pos(s[]_p^N \cdot t) & \stackrel{\text{def}}{=} & \emptyset \end{cases}$$

For example, the constructions of figure 1.2 are allowed here. Of course the intended meaning of the construction $s[]_p^N$ (which is defined precisely in [4]) is to iterate the context $s[]_p$ (i.e. the term s in which the subterm at position p has been erased) Nth times.

The main result (which can be found in [4]) is the following:

Theorem 2.1 *Unification of I-terms is decidable (and finitary).*

However, if iterated parts themselves are allowed to contain integer exponents *along the path which is iterated* (this corresponds to constructions of the form $(s[]_p^N)^M$), then unification becomes undecidable.



Examples of terms in T which are not ρ-terms

References

[1] H. Chen and J. Hsiang. Logic programming with recurrence domains. In *Proc. ICALP 91, LNCS*, Madrid, 1991.

[2] H. Chen, J. Hsiang, and H.-C. Kong. On finite representations of infinite sequences of terms. In S. Kaplan and M. Okada, editors, *Proc. CTRS 90*, Montreal, 1990.

[3] H. Comon. Completion of rewrite systems with membership constraints. In W. Kuich, editor, *Proc. ICALP 92*, Vienna, 1992. Springer-Verlag. An extended version is available as LRI Research Report number 699, Sept. 1991.

[4] H. Comon. On unification of terms with integer exponents. Research Report 770, L.R.I, Univ. Paris-Sud, Orsay, Aug. 1992.

[5] E. Contejean. *Eléments pour la décidabilité de l'unification modulo la distributivité.* PhD thesis, Univ. Paris XI, Orsay, Apr. 1992.

[6] J.-P. Jouannaud and C. 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.* MIT-Press, 1991.

Negation Elimination in Equational Formulae

Hubert Comon, Maribel Fernández [1]

Terms with variables serve as a basis in many areas of computer science to denote the (possibly infinite) set of their ground instances. This is for example the case in all functional or logic programming languages. Therefore basic operations on terms are fundamental in many programming activities. Examples of such operations are the computation of the greatest lower bound (*unification*) and the computation of the smallest upper bound (*anti-unification*) of two terms. Let $[\![t_1, \ldots, t_n]\!]$ be the set of all ground instances of $t_1, \ldots, t_n$. The first operation (unification) can be reflected at the level of ground instances: to unify s and t (which are defined up to literal similarity) corresponds to the computation of $[\![s]\!] \cap [\![t]\!]$.

Another (less well-known) operation on terms has revealed to be fundamental: the *term complement*, i.e. the computation of the difference set $[\![t]\!] - [\![u_1, \ldots, u_m]\!]$. Some applications of such computations are described in [Com91, LMM91]. This difference cannot always be expressed as a finite set $[\![v_1, \ldots, v_k]\!]$ and Lassez and Marriott gave in [LM87] a precise characterization of the cases in which the difference is expressible within the same formalism.

The computation of term complement, as well as unification and other operations are actually nothing but particular cases of solving *equational formulae*. An equational formula is a first-order formula constructed over a finite alphabet $\mathcal{F}$ of function symbols and only one relational symbol: the equality. It has been shown that the validity of an equational formula in the free term algebra $T(\mathcal{F})$ is decidable [Mal71, CL89, Mah88], leading to a complete axiomatization of $T(\mathcal{F})$. The technique of [CL89] consists in reducing the formula (according to a set of rewrite rules) until a *solved form* is reached. In the unification case, a solved form can be for example a formula $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$ where $x_1, \ldots, x_n$ are variables which occur only once. Such a solved form represents a most general unifier in a straightforward way (see [JK91] for more information about solved forms of unification problems). In this view, solving equational formulae encompasses unification, as unification consists only in solving an equation $s = t$, but also encompasses term complement as the computation of $[\![t]\!] - [\![u_1, \ldots, u_m]\!]$ consists in solving the formula

$$\exists \mathcal{V}ar(t), \forall \mathcal{V}ar(u_1, \ldots, u_m) : \; x = t \wedge x \neq u_1 \wedge \ldots \wedge x \neq u_m$$

[1] CNRS and LRI, Bat. 490, Université de Paris Sud, 91405 ORSAY cedex, France, Email: {comon,maribel}@lri.lri.fr.

where $\mathcal{V}ar(t), \mathcal{V}ar(u_1, \ldots, u_m)$ denote the set of variables in t and $u_1, \ldots, u_n$ respectively.

However there are good reasons for which we would like the solved forms to have an additional property: to involve only equations, if this is possible. More precisely, let $[\![\phi]\!]$ be the set of ground assignments σ to the free variables of ϕ such that $T(\mathcal{F}) \models \phi\sigma$. We would like that a solved form ϕ' of ϕ does not contain any disequation whenever there is such a formula ψ satisfying $[\![\psi]\!] = [\![\phi]\!]$. Indeed, for a formula ϕ that satisfies these requirements, there are finitely many substitutions $\theta_1, \ldots \theta_k$ such that $[\![\phi]\!]$ is exactly the set of ground instances of $\theta_1, \ldots, \theta_k$. In other words, we would like to compute a finite set of generating substitutions $\theta_1, \ldots, \theta_k$ as soon as such a finite set exists. In the case of complement problems, this implies that we compute a finite set of terms $v_1, \ldots v_k$ such that $[\![t]\!] - [\![u_1, \ldots, u_m]\!] = [\![v_1, \ldots, v_k]\!]$ as soon as such a set exists.

Good reasons for these additional requirements on solved forms are described in [LMM91], but let us mention another motivation. Constraints systems are very useful in logic programming languages because they provide with adequate notations for specific computation domains. In particular, strategies can be expressed using constraints over $T(\mathcal{F})$. In the same way constraints enhance both efficiency and expressiveness in equational logic computations [KKR90]. For example, a single constrained equation can be used in place of a huge (or even infinite) set of equations. However, the drawback is the failure of the critical pair lemma: in any constrained version of rewriting, there are constrained rules without critical pairs (in the classical sense) which do not define a locally confluent rewrite system. For example, consider the constrained rewrite system R which contains only one rule:

$$x \neq b \ : \ f(x, y) \rightarrow b$$

where f is a binary function symbol, b is a constant and x, y are variables. There is no critical pair, but the system is not confluent: take for instance the term $f(f(a, b), b)$ where a is also a constant. There are two possible reductions leading to different terms in normal form:

$$f(f(a,b),b) \rightarrow b$$
$$f(f(a,b),b) \rightarrow f(b,b)$$

Recent trends in rewrite systems theory undertake this problem[2]. A simple way for handling the problem is to assume that every constraint can (eventually) be turned into equations only. Indeed, this means that, when necessary, it is possible to turn the constrained equations into unconstrained ones. In the previous example, if the signature $\mathcal{F}$ contains only the function symbols a, b and f, then $x \neq b$ is equivalent in $T(\mathcal{F})$ to $x = a \vee \exists z_1, z_2 : x = f(z_1, z_2)$, where z_1, z_2 are variables. Then R is equivalent to the system R' containig two unconstrained rules:

$$f(a,y) \rightarrow b$$
$$f(f(z_1,z_2),y) \rightarrow b$$

where there is a critical pair.

[2]There is not any published paper yet, but the problem of constrained completion of rewrite systems was one of the major themes of the 1st Int. Workshop on Constraint Theorem Proving, Dagstuhl, Oct. 1991.

Of course, for constraints systems where all constraints can be turned into equations the expressivity of equational logic is not really increased, but it is still possible to delay expensive computations until they are necessary (with the hope that they will never be necessary). This is widely demonstrated in [KKR90]. Now, since equational formulae are a possible constraint language, it is worthwhile to study the ability of turning such formulae into equations only, which is the subject of this paper.

For all these reasons, the main goal of our work is to give a terminating set of rules on equational formulae for which the irreducible formulae do not contain disequations if they are equivalent in $T(\mathcal{F})$ or $T(\mathcal{F})/_{=_E}$ to a formula which does not contain any disequation.

We solved the problem in the following cases:

1. Equational formulae in $T(\mathcal{F})$. See [CF92] where a restricted version of this problem is studied, and [Taj92] where another method of negation elimination is presented for the general case.

2. Equational formulae in $T(\mathcal{F})/_{=_E}$ when $=_E$ is the congruence generated by a permutative set of equations (in Mal'cev's sense [Mal71]). See [CF92].

3. Linear complement problems in $T(\mathcal{F})/_{=_{AC}}$ where AC is an associative and commutative theory. See [Fer92].

In the first two cases we use a procedure such as those described in [CL89] to eliminate universal quantifiers: it is possible to transform any equational formula into a (semantically equivalent) purely existential formula. Therefore, we have only to consider purely existential formulae. Then we use a set of transformation rules $\mathcal{R}$ that transforms any existential formula ϕ into a disjunction of *simple* formulae. This set of rules is correct (it preserves the set of solutions of ϕ) and terminating. The last step uses a set of transformation rules (which is inspired in the ideas of [Taj92]) to perform the inter-reduction of simple formulae. We proved that this set of rules is terminating, correct and complete (that is, the resulting formula does not have negation whenever there exists a positive formula equivalent to ϕ).

Our results for $T(\mathcal{F})$ extend easily to some quotient algebras. In order to preserve decidability we assume that the congruence $=_E$ is generated by a permutative set of equations (in Mal'cev's sense [Mal71]). With such a restriction, all previous results which were established for $T(\mathcal{F})$ carry over $T(\mathcal{F})/_{=_E}$.

Linear complement problems modulo Associativity and Commutativity (AC) have been shown decidable [KLP91]. We cannot hope much more powerful results for equational formulae in the AC case because the first order theory of a single AC function symbol is undecidable [Tre90]. But we proved, using the same approach as for the previous cases (systems of transformation rules), that also negation elimination is decidable for linear complement problems in AC theories.

References

[CF92] H. Comon and M. Fernández. Negation elimination in equational formulae. In *Proceedings 17th Mathematical Foundations of Computer Science, Praha 1992, LNCS*, 1992.

[CL89] Hubert Comon and Pierre Lescanne. Equational problems and disunification. *J. Symbolic Computation*, 7:371–425, 1989.

[Com91] Hubert Comon. Disunification: a survey. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.

[Fer92] Maribel Fernandez. Validity and negation elimination in linear ac complement problems. Research Report to appear, Laboratoire de Recherche en Informatique, Univ. Paris-Sud, 1992.

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT-Press, 1991.

[KKR90] Claude Kirchner, Helene Kirchner, and Michael Rusinowitch. Deduction with symbolic constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on automatic deduction.

[KLP91] E. Kounalis, D. Lugiez, and L. Pottier. Complement problems modulo associativity and commutativity. In *Proc. 16th Mathematical Foundations of Computer Science 91, Warsaw*. Springer-Verlag, 1991.

[LM87] J.-L. Lassez and K. G. Marriott. Explicit representation of terms defined by counter examples. *J. Automated Reasoning*, 3(3):1–17, September 1987.

[LMM91] J.-L. Lassez, M. Maher, and K. Marriott. Elimination of negation in term algebras. In *Proc. 16th Mathematical Foundations of Computer Science 91, Warsaw*. Springer-Verlag, 1991.

[Mah88] M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd IEEE Symp. Logic in Computer Science, Edinburgh*, pages 348–357, July 1988.

[Mal71] A. I. Mal'cev. Axiomatizable classes of locally free algebras of various types. In *The Metamathematics of Algebraic Systems. Collected Papers. 1936-1967*, pages 262–289. North-Holland, 1971.

[Taj92] M. Tajine. Representation explicite de certains langages de termes: théorie et applications. Thèse de l'université Louis Pasteur de Strasbourg, 1992.

[Tre90] Ralf Treinen. A new method for undecidability proofs of first order theories. In K. V. Nori and C. E. Veni Madhavan, editors, *Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 48–62. Springer Lecture Notes in Computer Science, vol. 472, 1990.

A Partial Solution for D-unification Based on a Reduction to $AC1$-Unification

Evelyne Contejean [1]

1 Introduction

It is very well-known that the notion of equation is crucial in mathematics as well as in computer science. Unification is solving equations in some particular domains, namely free term algebras or term algebras modulo an equational theory. Unification was first introduced by Hebrand [2], and rediscovered by Robinson as a basic mechanism for the resolution in first order logic [6].

On the other hand, the very old and famous 10^{th} Hilbert problem of solving Diophantine equations has been proved undecidable by Matijasevic [5]. The challenge now is to find the maximal subsets of Peano arithmetic for which unification is decidable. A candidate is actually provided by the two axioms of left and right distributivity of a symbol $*$ over a symbol $+$. Arnborg and Tidén have shown that one-sided (left *or* right) distributivity has a decidable unification, whereas adding a unit element for $*$ and associativity of $+$ makes unification undecidable [8]. Szabo has proved the same result when associativity of $+$ is added to both-sided distributivity [7].

We give a partial solution for both-sided distributivity. More precisely, we show that if a problem does not contain any $+$ function symbol, solvability modulo D and solvability in the free algebra are equivalent.

Describing all solutions of a problem P , however, is more difficult. In a first step, we show that all solutions (modulo D) are instances of the linearized form λ of the most general unifier of P in the free algebra. Not all of them, however, are solutions. The appropriate instances have a "top" which is an $AC1$ solution of the problem $P\lambda$ in the theory where $*$ is $AC1$.

2 Definitions

D is the equational therory presented by

$$\begin{aligned} x*(y+z) &= (x*y)+(x*z) && (D_l)\\ (x+y)*z &= (x*z)+(y*z) && (D_r)\end{aligned}$$

[1] LRI, CNRS URA 410, Université Paris-Sud, Bât 490
91405 ORSAY Cedex, France, Email: `contejea@lri.lri.fr`.

and the axioms of D are usually oriented from left to right, yielding developped normal forms for the terms in $\mathcal{T}(\{+,*\},\mathcal{X})$.

Definition 1 ($*$-Problems and Well-balanced $*$-Problems) *A* $*$-equation *in $\mathcal{T}(\{+,*\},\mathcal{X})/D$ is an equation $s =^?_D t$, where s and t are terms of $\mathcal{T}(\{*\},\mathcal{X})$. A* well-balanced $*$-equation *in $\mathcal{T}(\{+,*\},\mathcal{X})/D$ is an equation $t\sigma =^?_D t\tau$, where t is a term of $\mathcal{T}(\{*\},\mathcal{X})$ and σ and τ are two variables identifications. A* $*$-problem *(respectively a* well-balanced $*$-problem*) is a unification problem in the algebra $\mathcal{T}(\{+,*\},\mathcal{X})/D$ in which all equations are $*$-equations (respectively well-balanced $*$-equations).*

3 General $*$-Problems

This section is devoted to the proof of our first main result: if P is a unification $*$-problem then P is solvable modulo D if and only if it is solvable in the free algebra. Moreover all its solutions modulo D are instances of the linearized form of its most general unifier in the free algebra. The proof of this theorem needs some technical tools, some of which are based on rewriting techniques. To record some information about the application of distributive steps, we will mark "distinct" $*$ function symbols by distinct indexes. Two occurences of $*$ in a term will be considered identical if they come from the dupplication of a same $*$ symbol when using distributivity.

Definition 2 (Marked term) *Let $\mathcal{X}$ be a set of variables, and $\mathcal{I}$ be a set of indexes. A* marked term *with respect to $\mathcal{I}$ is either a variable x in $\mathcal{X}$, or $s+t$ where s and t are marked terms with respect to $\mathcal{I}$, or $s *_i t$ where s and t are marked terms with respect to $\mathcal{I}$ and i is an index of $\mathcal{I}$.*

Indexed distributivity, called DI in the following, is a version of distributivity which cops with marked terms. DI is presented by the indexed axioms:

$$\begin{array}{rcll} x_I *_i (y_J + z_K) & = & (x_I *_i y_J) + (x_I *_i z_K) & \qquad (DI_l) \\ (x_I + y_J) *_i z_K & = & (x_I *_i z_K) + (y_J *_i z_K) & \qquad (DI_r) \end{array}$$

where i is any index and the variables x_I, y_J and z_K can be instanciated by marked terms. Indexed distributivity has been introduced in order to study distributivity. Hence we would like that D and DI have the "same" equivalence classes. Unfortunately this is not always true, but this is the case for the *compatible indexations*. Any term in $\mathcal{T}(\{+,*\},\mathcal{X})$ can be marked in such a way that the obtained term has a compatible indexation. The obvious but useless solution is to mark all $*$ function symbols with the same index. We introduce an ordering on indexations. Now, any term in $\mathcal{T}(\{+,*\},\mathcal{X})$ can be marked in such a way that the obtained term has a minimal (finest) compatible indexation. Moreover, this indexation is unique up to renaming. Minimal compatible indexations are very usefull in order to decide when a term can be "factorized", that is, when an axiom of distributivity can be applied from right to left. Indeed when two $*$ function symbols are marked with the same index, such a factorization is possible, and we show that there is no need to look at the subterms below them. This is important for unification since the terms that have to be unified are known, but the substitution that has to be applied in order to make them equal is *a priori* unknown.

Thanks to this factorization result, we can state proposition 1 about the solutions of $*$-problems:

Proposition 1 *Let P be a $*$-problem. P has a solution modulo D if and only if it has a solution modulo the empty equational theory. Moreover if P is solvable, and if σ is a a solution of P modulo D, then σ is an instance of the linearized most general unifier of P in the empty equational theory λ_0: $\sigma =_D \lambda_0\rho$.*

4 Well-balanced $*$-Problems

Well-balanced $*$-problems always admit solutions. The most trivial one is a substitution which maps all variables on a single variable. Hence, the question is not the solvability of such problems, but a full description of their solutions. In [3], Kirchner and Klay have proved that the distributivity is not a syntactic theory, by giving an infinite set of uncomparable solutions for the very simple equation $x * y =^?_D u * v$. We introduce a representation of the solutions which enables us to represent such an infinite set by a single scheme. Actually, we cannot represent all solutions, but only the "upper part" of each solution. Such an upper part is also a solution for a well-balanced $*$-problem.

In the following, $\mathcal{T}(\{+, \Box\})$ is called the algebra of structures. The upper part of a term in $\mathcal{T}(\{+, *\}, \mathcal{X})$, which is in developped normal form, is the structure obtained by replacing all $*$-headed subterms and variables by the $\Box$ symbol. A term can have several developped normal forms. If we extend our notion of upper part to terms which are not in developped normal form, we cannot ensure the uniqueness property. This problem is solved by considering classes of structures modulo an equivalence relation $\sim_D$ derived from the equational theory D: Two structures s and t are D-equivalent if $s\{\Box \mapsto x * y\} =_D t\{\Box \mapsto x * y\}$, and this is denoted by $s \sim_D t$. An important remark is that if t_1 and t_2 are in developped normal form, there is a normal from of $t_1 * t_2$ the upper part of which is equal to $\widehat{t_1}\{\Box \mapsto \widehat{t_2}\}$, which is denoted by $\widehat{t_1}@\widehat{t_2}$. The operator @ is called *composition*. The composition has some very nice properties: it is assocative, has a unit element equal to $\Box$, and is compatible with D-equivalence. Moreover it is commutative up to D-equivalence.

The quotient algebra $\mathcal{T}(\{+, \Box\})/\sim_D$ has a unique decomposition property with respect to the composition @, similar to the situation with natural numbers. This does not imply, however, that it is a factorial ring. If s and t are two D-equal terms in $\mathcal{T}(\{+, *\}, \mathcal{X})$ which are in developped normal form, then their upper parts are D-equivalent. As a consequence, two distinct developped normal forms of a term t have the same upper part modulo $\sim_D$, which is called the *top* of t.

All previous definitions of upper part and top extend naturally to substitutions. Thanks to the property of uniqueness for the maximal decomposition of a structure, we can describe all tops of the solutions of a well-balanced $*$-problem as instances of the most general unifier of an $AC1$ problem.

Proposition 2 *Let P be a well-balanced $*$-problem and σ a solution of P. Let ρ_{AC1} be the most general solution of P modulo the equational theory $AC1$ of $*$. Then there exists* *a semi-group homomorphism h from $\mathcal{T}(\{*\},\mathcal{X})/AC1(*)$ to $\mathcal{T}(\{+,\square\})/\sim_D$ such that*

$$\widehat{\sigma} \equiv \rho_{AC1} h$$

Conversly, let h' be any semi-group homomorphism from $\mathcal{T}(\{*\},\mathcal{X})/AC1(*)$ to $\mathcal{T}(\{+,\square\})/\sim_D$. Then $\rho_{AC1}h'\{\square \mapsto x * y\}$ is a solution of P.

There are infinitely many semi-group homomorphisms from $\mathcal{T}(\{*\},\mathcal{X})/AC1(*)$ to $\mathcal{T}(\{+,\square\})/\sim_D$, but ρ_{AC1} is unique and can be used to schematize all tops of solutions of P. By putting together proposition 2 and proposition 1, we get a description of the solutions of any $*$-problem P.

5 Conclusion

We have introduced new tools (indexed distributivity and the algebra of structures) which enable us to check very easily the solvability modulo D of a $*$-problem and to represent its infinite set of solutions with a single scheme. The most surprising result is that D-unification of $*$-problems boils down to $AC1$-unification. The question of course, is wether this can be generalized to arbitrary problems. We believe it can, and based on the previous tools, we have solved the general problem under the technical assumption that there exists no cycle of some special form called "cycles of non-null weight" [1]. We therefore conjecture that unification modulo distributivity is decidable, but the algorithm appears to be even more complex than Makanin's algorithm [4]. Moreover, it is interesting to notice that both of them use the solutions of Diophantine equations, for solving positions equations in Makanin's algorithm, and for computing $AC1$-solutions in our algorithm.

References

[1] Evelyne Contejean. Éléments pour la décidabilité de l'unification modulo la distributivité. Thèse de Doctorat, Université de Paris-Sud, France, Avril 1992.

[2] J. Herbrand. Recherches sur la théorie de la démonstration. Thèse d'Etat, Univ. Paris, 1930. Also in: Ecrits logiques de Jacques Herbrand, PUF, Paris, 1968.

[3] Claude Kirchner and Francis Klay. Syntactic theories and unification. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[4] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Akad. Nauk. SSSR*, 233(2), 1977.

[5] J. V. Matijasevic. Enumerable sets are diophantine. *Soviet Mathematics (Dokladi)*, 11(2):354–357, 1970.

[6] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[7] P. Szabo. Unifikationstheorie erster ordnung. Technical Report Thesis, Fakultat fur Informatik, University Karlsruhe, Karlsruhe, West Germany, 1982.

[8] Erik Tiden and Stefan Arnborg. Unification problems with one-sided distributivity. *Journal of Symbolic Computation*, (3):183–202, 1987.

Conditional Rewriting Presentations for General E-Unification

Bertrand Delsart[1]

1 Preliminaries

Many efficient algorithms have been devised to solve E-unification problems in particular theories. As regards general E-unification, there are three basic approaches: very efficient *narrowing* algorithms[7][10][14][13], complete procedures based on *lazy-unification* [8][9][6] and more efficient ones for *syntactic* theories[12]. This work is concerned with the definition of general transformation rules that subsume the topmost approaches, thus allowing the different strategies to be used simultaneously. The mutation rule is based on strictly resolvent conditional rewriting presentations of the theories, defined later.

Our notations are consistent with [5]. However, since we allow new variables in the right term of the conditional rewriting systems, we must define the corresponding rewriting rule. In our approach, conditions are restricted to conjunction of E-unification problems. A (given) term t is a rewriting of s by the conditional rule $c_i \,|\, l_i \to r_i$ at the position p by the substitution σ if and only if $p \in \mathcal{P}os(s)$, $s|_p = l\sigma$, $t = s[r\sigma]_p$, $\sigma \in U\Sigma_E(c)$ and $\mathcal{V}ar(l) \cup \mathcal{V}ar(r) \subseteq \mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(c) \cup \mathcal{V}ar(l) \cup \mathcal{V}ar(r)$ for some variants c, l and r of c_i, l_i and r_i (i.e. $\mathcal{V}ar(c \,|\, l \to r) \cap \mathcal{V}ar(s,t) = \emptyset$).

2 Transformation rules for E-unification

Our approach is based on the classic rules **Delete**, **Decompose**, **Coalesce** and **Eliminate**[11]. We must also consider **Imitate**, defined by Gallier and Snyder[9] for collapsing theories. As for the mutation rule, the efficiency of the syntactic approach comes from the resolventness of the equational presentations. We have defined a similar property for conditional term rewriting systems. A *rewriting presentation* of a theory is a (conditional) rewriting system R so that $\overset{+}{\longleftrightarrow}_E = \overset{+}{\longrightarrow}_R$. In fact, we always choose R such that $\overset{+}{\longrightarrow}_R \subseteq \longrightarrow_R \overset{*}{\longrightarrow}_{R,\neq\Lambda}$. It means that at most one rewriting can be done before decomposing. Such a presentation is said to be *strictly resolvent* and gives interesting properties to the rules given in the first figure.[2] The soundness of **Delete**, **Decompose**,

[1]LIFIA, Institut IMAG, 46 Avenue Felix Viallet, F-38031 GRENOBLE
e-mail : `delsart@imag.fr`.

[2]Where $\underline{\sigma} = \{x =^?_E x\sigma / x \in \mathcal{D}om(\sigma)\}$

Mutate	$P \cup \{s =_E^? t\}$ $\Longrightarrow P\sigma \cup \{r\sigma =_E^? t\sigma\} \cup c\sigma \cup \underline{\sigma}$ if $c \,\vert\, l \rightarrow r$ is a variant of a rule of R and σ is the most general unifier of s and l
Imitate	If $x \in \mathcal{V}ar(f(s_1, \ldots, s_n))$ then $P \cup \{x =_E^? f(s_1, \ldots, s_n)\}$ $\Longrightarrow P\sigma \cup \{x_1 =_E^? s_1, \ldots, x_n =_E^? s_n\} \cup \underline{\sigma}$ where $x_1, \ldots, x_n$ are new variables and $\sigma = \{x \mapsto f(x_1, \ldots, x_n)\}$

Coalesce, **Eliminate** and **Imitate** is well known. The proofs of the following results can be found in [4] and [3].

Theorem 2.1 *If R is a rewriting presentation of E then* **Mutate** *is sound.*

Theorem 2.2 Delete, **Decompose**, **Coalesce**, **Eliminate**, **Imitate** *and* **Mutate** *preserve the set of unifiers*[3] *when R is strictly resolvent and the control verifies the following property: if* **Decompose**, **Imitate** *or* **Mutate** *is applied on a given pair of terms then every transformation of this pair by one of these three rules must be tried concurrently.*

Actually, the control is refined thanks to the two following properties:
- Only one orientation of $s =_E^? t$ need to be considered for **Mutate** .
- **Mutate** need not be applied to the problem $r\sigma =_E^? t\sigma$ generated by its previous application (note that it implies $t \in \mathcal{X}$, $r \in \mathcal{X}$ or $\mathcal{H}ead(t) = \mathcal{H}ead(r)$).

Theorem 2.3 *Any non transformable problem is either an insoluble problem or a solved form.*

Theorem 2.2 and theorem 2.3 ensure that this approach results in a complete set of E-unifiers when it terminates. In section 4, we give particular rewriting presentations that simulate Gallier and Snyder's complete topmost approach and Kirchner's mutation rule for syntactic theories. Moreover, this more general framework is suitable for theoretical researches. For instance, the imitation rule is not efficient and does not allow emulation of the cycle-syntactic approach to collapsing shallow theories[1].

3 Improving the imitation rule

When occur-check fails, **Imitate** generates a loop. Fortunately, imitation cycles are avoided without losing completeness since they lead to equivalent problems. Moreover, **Imitate** is needed only to apply axioms above an occurrence of the initial x in Gallier and Snyder's algorithm. This result does not hold in a non-lazy approach[4]. However, we conjecture that the same restriction can be used with strictly resolvent presentations. We proved it only when any rewriting sequence is equivalent to a strictly resolvent one in which the highest step is done at the same position[5]. It is a direct consequence of the following theorem.

[3] i.e. each unifier solves at least one of the concurrently generated problems
[4] for instance, unification of x and $f(x, x)$ in $E = \{f(h(a), h(b)) \simeq h(a), a \simeq b\}$
[5] property verified by all the presentations we considered

Var-Mutate	$P \cup \{x =^?_E t[x]_q\}$ $\Longrightarrow P \cup \{t[r]_p =^?_E x\} \cup \{t\|_p =^?_E l\}$ if $p < q$ and $l \rightarrow r$ is a variant of a rule of R so that $l \in \mathcal{X}$ or $\mathcal{H}ead(l) = \mathcal{H}ead(t\|_p)$

Theorem 3.1 *Let σ be a solution of $x =^?_E t$ so that $t|_q = x$, for each rewriting presentation R of the theory, there is a position p strictly above q and a more general solution τ so that $t\tau \xrightarrow{*}{}^{>p}_{R} \longrightarrow^{p}_{l\rightarrow r} \xrightarrow{*}{}^{\geq p}_{R} \xrightarrow{*}{}^{\|p}_{R} x\tau$*

The previous theorem also ensures[6] the completeness of the mutation rule in the second figure when every rule of a rewriting presentation R is applied concurrently at every position p above a chosen occurrence of x.

In addition, **Decompose** can be applied to $t|_p =^?_E l$ when l is not a variable. When dealing with shallow theories, completeness is preserved even if considering only the subterm-collapsing rules[7] of a cycle-syntactic presentation. We are trying to restrict the set of rules considered in the general case. Unfortunately, even rules like $x \rightarrow f(x)$ are needed for some theories[8].

4 E-unification algorithms

The most important point is that any theory E=$\{l_i \simeq r_i\}$ can be represented by

$$R \;=\; \{\{x \underset{E}{\overset{?}{=}} l_i\} \,|\, x \rightarrow r_i\} \cup \{\{x \underset{E}{\overset{?}{=}} r_i\} \,|\, x \rightarrow l_i\}.$$

This strictly resolvent presentation simulates Gallier and Snyder's topmost approach. As regards the syntactic theories, R=$\{\{X =^?_E U\} \,|\, f(X) \rightarrow g(V), \{Y =^?_E V\} \,|\, g(Y) \rightarrow f(U) \ldots\}$ is a rewriting presentation of the theory E=$\{f(U) \simeq g(V)\ldots\}$, where capital letters denote lists of arguments. Moreover, R is strictly resolvent if E is resolvent. Thus, **Mutate** is more general than Gallier and Snyder's topmost approach and Kirchner's mutation rule. For example, a syntactic theory can easily be merged with a non-syntactic one if they are disjoint.

We can also use the conditions to enable mutation of the generated pair $r\sigma =^?_E t\sigma$ for a given rule ($c \,|\, l \rightarrow r$ becomes $c \cup \{x =^?_E r\} \,|\, l \rightarrow x$). This is particularly useful to extend a syntactic theory. Let R be the previously defined strictly resolvent presentation of a syntactic theory E_0 and E be $E_0 \cup \{l \approx r\}$.

Theorem 4.1 *$R \cup \{\{x =^?_E l, y =^?_E r\} \,|\, x \rightarrow y\} \cup \{\{x =^?_E l, y =^?_E r\} \,|\, y \rightarrow x\}$ is a strictly resolvent presentation of E.*

The corresponding E-unification algorithm can be expressed as follow: if topmost applications of the new axiom are needed then introduce them without decomposition else

[6] when allowing any rewriting after the first one at position p

[7] $\forall \theta \in \Sigma \; |l\theta| > |r\theta|$

[8] for instance to solve $h(x) =^?_E x$ in $E=\{f(x) \simeq x, f(h(x)) \simeq x\}$

apply the unique topmost step of the resolvent proof. A deterministic algorithm should add the fact that the rightmost application of this new axiom at the root can be chosen, thus restricting the mutations of $r =^?_E t$ to conditional rules of R. A more complex presentation ensures this behavior. Let $\mathrm{R}_{l\to r}$ be $\{\{x=^?_E l\} \mid x \to r\} \cup \{c\sigma \cup \{x=^?_E l\sigma\} \mid x \to d\sigma$ so that $c \mid g \to d \in \mathrm{R}$ and σ is the most general unifier of r and $g\}$.

Theorem 4.2 *$R \cup R_{l\to r} \cup R_{r\to l}$ is a strictly resolvent presentation of E.*

More generally, algorithms are build by searching conditions that ensure the existence of equivalent strictly resolvent proofs. The previous example shows that bigger sets of simpler rules often leads to more efficient algorithms.

Since unification is used to recognize the left parts, efficiency mainly depends on the complexity of the conditions. Some theories admit a strictly resolvent presentation the conditions of which include only E-equalities between variables. For instance, Transitivity with Commutativity of the innermost symbol can be expressed by the following strictly resolvent presentation:

(1) $xRy \to yRx$
(2) $\{x=^?_E z\} \mid (xRy)*(zRt) \to (xRy)*(yRt)$
(3) $\{x=^?_E t\} \mid (xRy)*(zRt) \to (xRy)*(yRz)$
(4) $\{y=^?_E z\} \mid (xRy)*(zRt) \to (xRy)*(xRt)$
(5) $\{y=^?_E t\} \mid (xRy)*(zRt) \to (xRy)*(xRz)$
(6) $\{x=^?_E z, x=^?_E t\} \mid (xRy)*(zRt) \to (xRy)*(yRy)$
(7) $\{y=^?_E z, y=^?_E t\} \mid (xRy)*(zRt) \to (xRy)*(xRx)$

It must be compared to the presentation simulating the syntactic approach:

(1') $\{x_1=^?_E x, x_2=^?_E y\} \mid x_1 R x_2 \to yRx$
(2') $\{x_1=^?_E xRy, x_2=^?_E yRz\} \mid x_1 * x_2 \to (xRy)*(xRz)$
(3') $\{x_1=^?_E xRy, x_2=^?_E yRy\} \mid x_1 * x_2 \to (xRy)*(xRx)$

However, useless bindings of the variables may occur when one of the arguments of '$*$' is a variable. We have implemented an algorithm which detects the following redundancy and ensures that the search space is at most equal to the syntactic one:

(2)≡(4),(3)≡(5) and (6)≡(7) when the first argument is a variable
(2)≡(3) and (4)≡(5) when the second one is a variable

5 Conclusion

Mutate and **Imitate** define a new approach to general E-unification. Algorithms depend mainly upon the chosen conditional rewriting presentation of the theory. An algorithm generating efficient strictly resolvent presentation has been developed and can be found in [3]. Conditions are restricted to equalities between variables but termination of the completion leads to "restricted completeness". Interesting practical results have been obtained with the imitation rule. Improving the behavior when occur-check fails should lead to a really efficient algorithm. We are also trying to mix the different strategies to preserve completeness (using more complex conditions).

References

[1] H. Comon, K. Haberstrau and J.-P. Jouannaud. Decidable Problems in Shallow Equational Theories In *Proc. of the 7th IEEE Symposium on Logic in Computer Science*, 1991.

[2] B. Delsart. General E-unification : A realistic approach. Presented at UNIF'91, 1991. (extended abstract).

[3] B. Delsart. Efficient E-unification algorithms. Technical report, INPG, 1992.

[4] B. Delsart. A New Approach to General E-Unification based on Conditional Rewriting Systems. To appear in *Proc. of CTRS'92*, Springer-Verlag, 1992.

[5] N. Dershowitz and J.-P. Jouannaud. Notations for rewriting. *EATCS*, 43:162–172, 1991.

[6] D. Dougherty and P. Johann. An improved general E-unification method. In *Proc. of the 10th CADE*, pages 261–275. Springer-Verlag, LNCS 449, 1990.

[7] M. Fay. First order unification in equational theories. In *Proc. of the 4th Workshop on Automated Deduction*, pages 162–167, 1979.

[8] J.H. Gallier and W. Snyder. A general complete E-unification procedure. In *Proc. of the 2nd RTA*, pages 216–227. Springer-Verlag, LNCS 256, 1987.

[9] J.H. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *TCS*, 67(2,3):203–260, 1989.

[10] J.-M. Hullot. Canonical forms and unification. In *Proc. of the 5th CADE*, pages 318–334. Springer-Verlag, LNCS 87, 1980.

[11] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. Technical Report 561, LRI, 1990.

[12] C. Kirchner. Computing unification algorithms. In *Proceedings of the first IEEE Symposium on Logic in Computer Science*, pages 206–216, 1986.

[13] W. Nutt, P. Rety, and G. Smolka. Basic narrowing revisited. *JSC*, 7(3,4):295–318, 1989.

[14] P. Rety. Improving basic narrowing. In *Proc. of the 2nd RTA*, pages 226–241. Springer-Verlag, LNCS 256, 1987.

A Unification- and Object-Based Symbolic Computation System[1]

Georgios Grivas[2]

1 Introduction

While pattern matching is not a necessary requirement in symbolic or functional programming, it is however nowadays a part of modern symbolic and functional languages such as Mathematica [14], Axiom [11], Reduce [7], Macsyma [9], Standard ML [10] and Miranda [12]. Actually, Mathematica delivers a more sophisticated pattern-matcher than any of the conventional symbolic and functional languages.

AlgBench [8] is an object-oriented symbolic computation system. It provides an interpreter for a symbolic language and a skeleton for implementing data types and algorithms in all areas of symbolic computation. It supports, like Mathematica, a fully functional language with pattern-matching and term rewriting facilities. Both systems provide type-checking at the user level as a special type of pattern matching.

On the other hand there are also relatively efficient systems that do not support the declarative kind of programming (e.g. Maple [4]). This gives the indication that sometimes the non-declarative way is more efficient. The main reason for this is the relatively bad efficiency of pattern matching algorithms used in existing symbolic computation systems.

The purposes presented in this paper are: First to extend the one-way pattern matching to (ac-) unification for *AlgBench* and therefore to increase the expressive power of its language. Second, to extend the type-constrained pattern matching by taking into account inheritance information in the unification process from a user-defined hierarchy of object types.

2 Extending Matching to Unification

All function definitions in *AlgBench* are rewrite rules, each of which has a pattern on the left-hand side and a replacement on the right-hand side. The idea to replace in a

[1]Research supported by the Swiss National Science Foundation

[2]Institute for Theoretical Computer Science, ETH Zürich, Switzerland,
Email: grivas@inf.ethz.ch.

term rewriting environment the pattern matching with unification, i.e. to implement *narrowing* [6], is legitimate, in order to gain both the advantages of functional and logic languages. In the unification process the pattern variables can be unified and bound to other pattern variables. When a pattern variable is instantiated, all the variables bound to it see its instantiation value. This is a powerful programming technique: incrementally instantiating partial data structures.

An improved version of the *Huet algorithm* for standard unification presented in [13] is implemented. Its basic concept is to build equivalence classes for subexpressions through union-find trees [1]. Its advantage, that works with both DAG and non-DAG representation of terms, was decisive for our choice. We realized the abstract data structure of the union-find tree as a class and the functions `union` and `find` as its methods. This is an important conceptual difference compared to the procedural solution of [1]. The Huet algorithm in [13] is formulated for functions and variables. Therefore we view all symbols, strings, numbers and pattern objects as variables. When they have to be unified with an expression, the specialized unifier for the corresponding class is called. As a result the algorithm works also for new coming classes of pattern objects. All other composite expressions are viewed as functions.

The improved version of the Stickel's algorithm [5] was chosen for implementation of associative commutative unification. For the declaration of the ac-functions serves the command `SetAttributes[`*symbol*`,` *attribute*`]`. The attribute is called -like in Mathematica- `Flat` for the associative case and `Orderless` for the commutative case.

A mechanism selects the right unification algorithm to be called. We distinguish at the moment three cases: complicated expressions, simple expressions and expressions with `Flat` and `Orderless` attributes. Later we want to support expressions that contain subtype information. The criteria for simple versus complicated terms include the length and the number of nested expressions within the term. In the simple case we choose the *mark* and *retract* algorithm proposed by R. Maeder, in the case of complicated expressions the Huet algorithm and in the case of ac-functions the Stickel algorithm.

3 Unification based on Subtyping

Subtyping is a substitutability relationship. Intuitively one type is a subtype of another if an object of a subtype can stand in for an object of a supertype. In object-oriented programming a subtype has all fields and operations of its supertype as well as additional fields and operations. In other words, *subtyping* is the distribution of types into a generalization/specialization hierarchy. P. America [3] shows that *inheritance* (also called *subclassing*), which deals with code sharing among classes, is not always subtyping, which has to do with specialization in behaviour of objects. In this sense subclasses inherit the implementation and subtypes inherit the interface.

H. Ait-Kaci and R. Nasr [2] presented a logic programming language *Login*, which incorporates inheritance-based information (in form of an IS-A taxonomy) immediately into the Huet unification algorithm. Because of our object-oriented design, we do not need to integrate inheritance information into the matching process for the data types provided by the system. The type checking is done automatically by the virtual function mechanism.

The heads of rewrite rules are interpreted in Mathematica inter alia as object types. We implemented an *AlgBench* package, that creates with the subtype command an inheritance hierarchy and then takes into account this information for the object type unification. The non-typed objects are unified with the kernel unifiers. We give the user the opportunity to define inheritance hierarchies of types via a subtype command: `Subtype[`*list1* , *list2*`]`, where *list1*, *list2* are lists of object types. Consider the following example:

```
In[1]:= Subtype[f,{g, h, i}]
In[2]:= Subtype[h, j]
In[3]:= Subtype[{g, i}, k]
In[4]:= f[x_] := x
In[5]:= g[y_] := x+y
```

The type extension is expressed in the definition of g, where the arguments of f are inherited implicitly. With the subtype command we allow user-defined type hierarchies within composite expressions, a user-defined *lattice*.

Definition: *Two object types F,G unify iff there exists a non empty H such that H is the greatest lower bound of F and G with the following binding list:* $\{F \rightarrow H,\ G \rightarrow H\}$.

We put at the user-level the command `Unify[expr1, expr2]`. This means in the previous example:

```
In[1]:= Unify[i[j_], g[h_]]
Out[1]= {i -> k, g -> k, h -> j}
```

The heads of the composite subexpressions (object types) are unified as a GLB (*greatest lower bound*) lattice operation. We can do the following:

```
In[1]:= k[ x_f ] := x + 1
In[2]:= i[g[a,b]]
Out[2]= a + b + 1
```

Since there is no rule for i, the types of k, i are unified with the GLB-operation to k. The types f, g are unified for their part to g. As there is a rule for g, we get after a couple of rewrite steps a+b+1.

All rules of the supertype are inherited after the rules of the subtype as in the example of the abstract logarithm function:

```
In[1]:= log[x_, y_] := log[x] + log[y]
In[2]:= log[x_^n_] := n log[x]
In[3]:= log[1] = 0
```

We specialize the logarithm function with the logarithms of base 2 (ld) and e (ln):

```
In[4]:= Subtype[log, {ld, ln}]
In[5]:= ld[2] = 1
In[6]:= ln[e] = 1
```

If we set the query:

```
In[7]:= ld[2^k]
```

Since there is no rule for matching the pattern `ld[2^k]` the inherited rules are examined, and following rewrite steps are done: ld[2^k] => k ld[2] => k 1 => k.

Reexamining the example at the beginning of this section, we see that we need an extension of the subtype command, where it is specified which arguments of the subtype are projected to the supertype: `Subtype[h1[`*expr1*`], h2[`*expr2*`]]`. The user gives the arguments that have to be inherited, otherwise we select the arguments from left to right and cut the rest:

```
In[1]:= f[x_] := x^2
In[2]:= Subtype[f,g]  /* there is no rule for g */
In[3]:= g[a,b]   /* select from left-to-right a and cut b */
Out[3]= a^2
In[4]:= Subtype[f[y_], g[x_, y_]]
In[5]:= g[a,b]        /* select b and cut a */
Out[5]= b^2
```

We realized the above extension by producing and storing a new global rule.

4 Conclusions

The availability of several unifiers and of their selection mechanism makes the system ampler and faster. Both one-way and two-way pattern matching are supported. The improvement of the pattern matching operation in an object-oriented way seems to be very appropriate.

Acknowledgements

I would like to thank R. Maeder for his useful comments and discussions. My thanks also go to S. Missura for reviewing an early draft of this paper. Parts of the mentioned unifier implementations were realized by M. Ochsner and M. Werner.

References

[1] A. Aho, J. Hopcroft, J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built–in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[3] P. America. Inheritance and subtyping in a parallel object-oriented language. *Proc. of ECOOP'87*, Paris, France, 1987.

[4] B. Char, K. Geddes, G. Gonnet, M. Monagan, S. Watt. *Maple Reference Manual.* 5th edition, Waterloo Maple Publishing, Ontario, Canad, 1990.

[5] F. Fages. Associative Commutative Unification. *Journal of Symbolic Computation*, 3, 257-275, 1987.

[6] M. Fay. First order unification in equational theories. *Proc. 4th CADE'79*, 1979.

[7] A. Hearn. *Reduce-3 User's Manual.* The RAND Corp., Santa Monica CA, 1979.

[8] R. Maeder. AlgBench: An Object-Oriented Symbolic Computation Core System. *Proc. of DISCO'92*, Bath, England, April 1992.

[9] *Macsyma Reference Manual*, Lab. for Comp. Sci., MIT, 1983.

[10] R. Milner, M. Tofte, R. Harper. *The definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[11] R. Sutor. *AXIOM User Guide.* 2nd Draft Edition, 1991.

[12] D. Turner. Miranda: A non-strict functional language with polymorphic types. *Proc. of Functional Programming Languages and Computer Architecture FPCA'85*, Nancy, France, September 1985.

[13] J. Vitter and R. Simons. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Transactions on Computers*, Vol. C–35, No. 5, May 1986.

[14] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* 2nd Edition, Prentice Hall, 1991.

Counterexamples to Completeness Results for Basic Narrowing

Eric Hamoen[1]

Narrowing is a generalization of term rewriting. It can be used as an algorithm to determine whether two terms unify modulo a certain (C)TRS R. It can also be used as the operational semantics for a language which integrates functional and Horn-clause programming. Basic narrowing is a more efficient form of narrowing.

It has been conjectured that basic narrowing is complete for semi-complete TRS's (Yamamoto) and that basic conditional narrowing is complete for semi-complete orthogonal CTRS's (Giovannetti & Moiso). We have found counterexamples for these conjectures. Furthermore, we show that one of the assumptions in Hoelldobler's completeness proof for basic conditional narrowing is incorrect and we give a way of repairing this problem. We will give certain syntactical restrictions that make basic narrowing complete for semi-complete TRS's. Finally, we show that narrowing is complete for level-confluent CTRS's that may contain variables in the right-hand side of a rule that do not appear in its left-hand side.

[1]Vrije Universiteit Amsterdam, Faculteit Wiskunde en Informatica, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands, email: `erik@cs.vu.nl`.

AC1-Unification/Matching in Linear Logic Programming

Steffen Hölldobler,[1] Josef Schneeberger,[2] Michael Thielscher[3]

Linear logic programming is a new deductive approach which can be used to formalize planning problems, deductive databases, object-oriented programming, etc. ([4]). Recently, two other approaches for solving planning problems deductively were proposed, namely the linear connection method [1] and a fragment of the linear logic [7]. Both approaches are shown to be equivalent to the linear logic programming approach applied to planning problems [9, 5]. All these approaches do not require to state frame axioms [8, 3] explicitely. In addition, the linear logic programming approach has a well defined semantics as it deals with standard first order logic [6].

The linear logic programming approach requires unification under an equational theory which consists in the axioms of associativity, commutativity, and the existence of a unit element. In the past twenty years, some algorithms solving unification under associativity and commutativity have been developed (cf. [2]). However, the problems concerning AC1-unification encountered in the linear logic programming approach show a special structure, and the general unification algorithms perform a lot of unnecessary and redundant computation if applied to these problems. In this paper, we define a restricted class of terms containing an AC1-function and give efficient unification algorithms solving AC1-matching and unification on these terms.

[1]Intellektik, Informatik, TH Darmstadt, Alexanderstr. 10, D-6100 Darmstadt, Germany, Email: steffen@intellektik.informatik.th-darmstadt.de

[2]FORWISS, Am Weichselgarten 7, D-8520 Erlangen-Tennenlohe, Germany, Email: jws@forwiss.uni-erlangen.de

[3]Intellektik, Informatik, TH Darmstadt, Alexanderstr. 10, D-6100 Darmstadt, Germany, Email: thielscher@intellektik.informatik.th-darmstadt.de

1 Introduction

The most thouroughly investigated application of linear logic programming is the field of deductive planning. As an example, consider the situation where you are in a shopping centre and, since you are really thirsty, you look for a lemonade. After a while you find a vending machine where a lemonade costs 75c. In your pocket, you find a dollar note as well as a quarter, i.e. it seems to be no problem to get the lemonade. Unfortunately, the vending machine only accepts quarters. Nevertheless, beside the vending machine there is a cashier which changes a dollar note into four quarters. Therefore, the desired result can be achieved by first changing the dollar note and then to get the lemonade.

In linear logic programming, planning problems of this kind can be formalized as logic programs by using terms for describing situations and clauses for describing operators. For example, the initial situation of our example can be represented by the term $dollar \circ quarter$ where $\circ$ denotes a binary function.[4] The logic program is based on a ternary predicate $plan(s, p, t)$ with the intended meaning that the plan p applied to situation s leads to situation t. With the help of this predicate, the planning problem can be formulated as a query to the program:

$$\leftarrow\ plan(dollar \circ quarter,\ P,\ lemonade) \tag{0.1}$$

To solve this problem, we need two clauses which formalize the operator to change a dollar note and the operator to use the vending machine:

$$\begin{aligned} &plan(dollar \circ V,\ [change, P'],\ W) \\ &\quad \leftarrow\ plan(quarter \circ quarter \circ quarter \circ quarter \circ V,\ P',\ W). \end{aligned} \tag{0.2}$$

$$\begin{aligned} &plan(quarter \circ quarter \circ quarter \circ V,\ [get\text{-}lemonade, P''],\ W) \\ &\qquad\qquad \leftarrow\ plan(lemonade \circ V,\ P'',\ W). \end{aligned} \tag{0.3}$$

The two clauses (0.1) and (0.2) can be resolved by using the substitution

$$\{V \mapsto quarter,\ P \mapsto [change, P'],\ W \mapsto lemonade\}.$$

The resolvent

$$\leftarrow\ plan(quarter \circ quarter \circ quarter \circ quarter \circ quarter,\ P',\ lemonade) \tag{0.4}$$

describes the reduced planning problem of getting a lemonade when owning five quarters. This problem can be solved by resolving (0.4) and (0.3) with $\{V \mapsto quarter \circ quarter,$ $P' \mapsto [get\text{-}lemonade, P''],\ W \mapsto lemonade\}$ which leads to the resolvent

$$\leftarrow\ plan(lemonade \circ quarter \circ quarter,\ P'',\ lemonade) \tag{0.5}$$

Since the goal situation, i.e. the lemonade, is now contained within the actual situation, we can use a termination clause which states that the planning problem is solved:

$$plan(X \circ Y,\ [],\ X) \leftarrow .^{5} \tag{0.6}$$

[4]Throughout the paper we use PROLOG-syntax, i.e. constant as well as function symbols are written in lower case, whereas variables are in uppercase letters.

[5]The constant [] denotes the *empty* plan.

As resulting substitution for the variable P in (0.1) we receive the term

$$[\textit{get-lemonade}, [\textit{change}, []]].$$

The desired properties of the function $\circ$ follow from the fact that the order of the subterms is irrelevant when describing a situation. Therefore, $\circ$ is required to be associative and commutative. $\circ$ must not be idempotent, since otherwise the terms $\textit{quarter} \circ \textit{dollar}$ and $\textit{quarter} \circ \textit{quarter} \circ \textit{quarter} \circ V$ are unifiable, i.e. one quarter will be enough to settle any amount. Beside the function $\circ$, we will use the constant $\emptyset$ to denote the *empty* situation. More formally, we use an equational theory AC1 which consists in three equations:

$$\begin{array}{rcl} \forall x, y, z.\ x \circ (y \circ z) & = & (x \circ y) \circ z \\ \forall x, y.\ x \circ y & = & y \circ x \\ \forall x.\ x \circ \emptyset & = & x \end{array}$$

The equational theory is used whenever applying SLDE-resolution to the query and the program clauses.

The situation terms used within our example only contain constants together with at most one variable. However, this restriction is not desired in general when using linear logic programming. For example, consider the situation in the blocksworld where block a is on block b, block a is clear, and the robot is holding block c. This can be described by the term $on(a, b) \circ clear(a) \circ holding(c)$. The preconditions of the operator which allows the robot to put down a block, contain the fact that the robot is holding a block as well as the fact that another block is clear, i.e. $holding(X) \circ clear(Y) \circ V$. This operator can be applied in the situation above, since both terms are unifiable under AC1 using the substitution $\{X \mapsto c,\ Y \mapsto a,\ V \mapsto on(a, b)\}$.

2 Problem classes

Our examples motivate a special class of unification problems modulo AC1. A significant property of the terms which appear in our formalism is that there is at most one variable on the level of the AC1-function $\circ$. For this special purpose, the general AC1-unification algorithms which are based on solving diophantine equations[6] are too inefficient. For example, within the blocksworld example stated above, the original algorithm of Stickel [10] generates nine basis solution of the corresponding diophantine equation which will lead to $2^9 = 512$ possible solutions. Since in fact there is only one solution, 511 of them have to be rejected.

Our example also shows that often one of the two terms to unify is ground, i.e. we have the chance to use AC1-matching instead of unification quite often.[7] Beside unification and matching, we define a third problem class which we call *restricted* unification. This problem class appears if both terms contain variables, but one of them does not contain

[6] I.e. linear equations with non-negative integer coefficients.

[7] More precisely, matching suffices in any case when the initial situation is fully specified.

a variable on the level of the AC1-function. For example, an initial situation described by the term

$$block(X) \circ holding(X),$$

represents the fact that it is unknown which block the robot is holding, but beside this uncertanity, the situation is fully specified, i.e. there are no more properties which hold in the situation.

Adding a variable on the level of the AC1-function (see below) requires our general AC1-unification. For example, the initial situation

$$quarter \circ V$$

can be used to ask the query: *What else do I need apart form a quarter to get a lemonade?*

It turns out that for all three problem classes very efficient unification algorithms can be formulated. More formally, we define our problem hierarchy as follows.

- An *$AC1_l$-matching problem*[8] consists of two terms $s_1 \circ \ldots \circ s_m \circ V$ and $t_1 \circ \ldots \circ t_n$ where the t_j, $1 \leq j \leq n$, are ground and V does not occur in s_i, $1 \leq i \leq m$.
- A *restricted $AC1_l$-unification problem* consists of two terms $s_1 \circ \ldots \circ s_m \circ V$ and $t_1 \circ \ldots \circ t_n$ where V does neither occur in s_i nor in t_j.[9]
- An *$AC1_l$-unification problem* consists of two terms $s_1 \circ \ldots \circ s_m \circ V$ and $t_1 \circ \ldots \circ t_n \circ W$ where V and W are different and do neither occur in s_i nor in t_j.

3 Solutions

Our algorithms are based on unification over multisets. In the sequel, we sketch the idea of the solution for each of our problem classes.

AC1$_l$-matching. A substitution σ is a matcher for an AC1$_l$-matching problem iff $\sigma(s_1 \circ \ldots \circ s_m \circ V) =_{\mathrm{AC1}} t_1 \circ \ldots \circ t_n$, where t_j, $1 \leq j \leq n$, are ground. It is easy to prove that if σ is a solution for the AC1$_l$-matching problem then $\{\!\{\sigma s_1, \ldots, \sigma s_m\}\!\} \dot{\subseteq} \{\!\{t_1, \ldots, t_n\}\!\}$.[10] Conversely, if we find a substitution θ such that $\{\!\{\theta s_1, \ldots, \theta s_m\}\!\} \dot{\subseteq} \{\!\{t_1, \ldots, t_n\}\!\}$ then the AC1$_l$-matching problem consisting of the terms $s_1 \circ \ldots \circ s_m \circ V$ and $t_1 \circ \ldots \circ t_n$ is solvable and the matching substitution σ can be constructed from θ as follows. Let

$$\{\!\{r_1, \ldots, r_{n-m}\}\!\} = \{\!\{t_1, \ldots, t_n\}\!\} \dot{-} \{\!\{\theta s_1, \ldots, \theta s_m\}\!\}.$$

Then, $\sigma = \theta|_{\mathcal{V}ar(s_1,\ldots,s_m)} \cup \{V \mapsto r_1 \circ \ldots \circ r_{n-m}\}$.[11]

[8]The index $_l$ is used to stress that we regard special unification problems in the linear logic programming approch.

[9]Note that the subterms t_j may now contain variables.

[10]Multisets are depicted using the modified curly brackets $\{\!\{$ and $\}\!\}$. Furthermore, $\dot{\subseteq}$, $\dot{\cup}$, $\dot{-}$, etc. denote the multiset extensions of the usual set operations $\subseteq$, $\cup$, $-$, etc.

[11]$\mathcal{V}ar(X)$ denotes the set of variables occurring in the syntactic object X and $\sigma|_V$ denotes the restriction of the substitution σ to the variables in V.

Let $\mathcal{S}$ and $\mathcal{T}$ be two multisets. With the previous discussion, we are now interested in computing a complete and, if exists, minimal set Σ of substitutions such that for each $\sigma \in \Sigma$ we find that $\sigma\mathcal{S} \dot{\subseteq} \mathcal{T}$.[12] To receive this, we use the following derivation rule.

$$\frac{\mathcal{S} \dot{\cup} \{\!\{s\}\!\}\ ,\ \mathcal{T} \dot{\cup} \{\!\{t\}\!\})}{\theta\mathcal{S}\ ,\ \mathcal{T}} \quad \theta \;=\; mgu\,(s,\,t)^{13}$$

Starting with the tuple $\mathcal{S}\,,\,\mathcal{T}$ the derivation stops if the first multiset $\mathcal{S}$ is empty. Thus, every successfull derivation leads to a resulting tuple of the form $\{\!\{\}\!\}\,,\,\mathcal{T}'$ and a sequence $\theta_1, \ldots, \theta_m$.[14] Let $\theta = \theta_1 \cdots \theta_m$, then θ is a solution of the subset problem defined above. Moreover, $\mathcal{T}'$ contains the subterms which shall be bound to the variable V in the original $\mathrm{AC1}_l$-matching problem. In this way, all successful derivations lead to a coomplete and minimal set of matcher.

Restricted $\mathbf{AC1}_l$-Unification. Restricted $\mathrm{AC1}_l$-unification differs from $\mathrm{AC1}_l$-matching in the way that both terms may contain variables. However, since again the second term does not contain a variable on the $\circ$-level, the unification algorithm for restricted $\mathrm{AC1}_l$-unification problems is as the matching algorithm except that now substitutions are applied to both multisets, i.e. we try to find substitutions σ such that $\sigma\mathcal{S} \dot{\subseteq} \sigma\mathcal{T}$ holds. Using a modified derivation rule

$$\frac{\mathcal{S} \dot{\cup} \{\!\{s\}\!\}\ ,\ \mathcal{T} \dot{\cup} \{\!\{t\}\!\})}{\theta\mathcal{S}\ ,\ \theta\mathcal{T}} \quad \theta \;=\; mgu\,(s,\,t)$$

all successful derivations lead again to a complete and finite set of solutions. Unfortunately, this set may contain non-minimal substitutions. Therefore, we have to test and, if necessary, to remove subsumed substitutions to receive a minimal set of unifiers. We may use some heuristics to keep the set as small as possible during computation.

$\mathbf{AC1}_l$-Unification. The problem of unifying two terms that both include an AC1-variable cannot be reduced to a subset problem over multisets. However, each solution σ of the AC1-unification problem defined by the terms $s_1 \circ \ldots \circ s_m \circ V$ and $t_1 \circ \ldots \circ t_n \circ W$ can be interpreted as dividing the representing multisets $\mathcal{S} = \{\!\{s_1, \ldots, s_m\}\!\}$ and $\mathcal{T} = \{\!\{t_1, \ldots, t_n\}\!\}$ into two disjunctive parts $\mathcal{S}_1, \mathcal{S}_2$ and $\mathcal{T}_1, \mathcal{T}_2$, respectively, such that we find a most general substitution θ which unifies $\mathcal{S}_1$ and $\mathcal{T}_1$. Let $\theta\mathcal{S}_2 = \{\!\{u_1, \ldots, u_k\}\!\}$ and $\theta\mathcal{T}_2 = \{\!\{v_1, \ldots, v_l\}\!\}$. Then, $\sigma = \theta \cup \{V \mapsto v_1 \circ \ldots \circ v_l,\ W \mapsto u_1 \circ \ldots \circ u_k\}$.

Since within $\mathrm{AC1}_l$-unification, the elements of the first multiset may not be unified with an element of the second, we have to modify our derivation rule and to add a second one. In particular, we use the following derivation rules:

$$\frac{\mathcal{S} \dot{\cup} \{\!\{s\}\!\}\ ,\ \mathcal{S}_1\ ,\ \mathcal{T} \dot{\cup} \{\!\{t\}\!\})}{\theta\mathcal{S}\ ,\ \mathcal{S}_1 \dot{\cup} \{\!\{s\}\!\}\ ,\ \theta\mathcal{T}} \quad \theta \;=\; mgu\,(s,\,t)$$

[12] The notion of a minimal and complete set of substitutions is extended in the obvious way.

[13] mgu denotes the most general unifier under the empty theory, i.e. the standard unification.

[14] Note that a successful derivation stops after exactly m applications of the derivation rule.

$$\frac{\mathcal{S} \dot\cup \{\!\{ s \}\!\} \;,\; \mathcal{S}_1 \;,\; \mathcal{T}}{\mathcal{S} \;,\; \mathcal{S}_1 \;,\; \mathcal{T}}$$

Again, a derivation stops if the first multiset is empty, i.e. the result is a triple of the form $\{\!\{\}\!\}\ ,\ \mathcal{S}_1\ ,\ \mathcal{T}_2$. Analogous to the other problem classes we have that every successful derivation determines a unifier of the corresponding $AC1_l$-unification problem.

The algorithms for $AC1_l$-matching, restricted $AC1_l$-unification, and $AC1_l$-unification are implemented and used successfully within a Prolog-implementation of our linear logic programming approach. They turned out to be very efficient, as they consider the characteristics of the AC1-terms occurring in all applications of our linear logic programming approach. The details of the algorithms as well as a proof of their correctness, completeness, and minimality can be found in [11].

References

[1] Wolfgang Bibel. A Deductive Solution for Plan Generation. *New Generation Computing*, 4:115–132, 1986.

[2] Hans-Jürgen Bürckert, Alexander Herold, Deepak Kapur, Jörg H. Siekmann, Mark E. Stickel, Michael Tepp, and Hanto Zhang. Opening the AC-Unification Race. *Journal of Automated Reasoning*, 4:465–474, 1988.

[3] C. Green. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 219–239, Los Altos, CA, 1969. Morgan Kaufmann Publishers.

[4] Gerd Große, Steffen Hölldobler, Josef Schneeberger, Ute Sigmund, and Michael Thielscher. Equational Logic Programming, Actions, and Change. In *Proceedings of the JICSLP*, 1992. (to appear).

[5] G. Grosse, S. Hölldobler, and J. Schneeberger. On linear deductive planning. Technical Report AIDA-92-08, Intellektik, Informatik, TH Darmstadt, 1992.

[6] Steffen Hölldobler and Josef Schneeberger. A New Deductive Approach to Planning. *New Generation Computing*, 8:225–244, 1990.

[7] M. Masseron, C. Tollu, and J. Vauzielles. Generating Plans in Linear Logic. Technical Report 90-11, Université Paris Nord C.S.P., Département de Mathématiques et Informatique, 93430 Villetaneuse, France, 1990.

[8] John McCarthy. Programs with Common Sense. In Marvin Minsky, editor, *Semantic Information Processing*, chapter 7, pages 403–418. MIT Press, 1968.

[9] Josef Schneeberger. *Plan Generation by Linear Deduction.* PhD thesis, FG Intellektik, TH Darmstadt, 1992.

[10] Mark E. Stickel. A Complete Unification Algorithm for Associative-Commutative Functions. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 71–76, Tbilisi, 1975.

[11] Michael Thielscher. AC1-Unifikation in der linearen logischen Programmierung. Diplomarbeit, Intellektik, Informatik, TH Darmstadt, Germany, September 1992. (in German).

Extensible Unification as Basis for the Implementation of CLP Languages

Christian Holzbaur[1]

1 Introduction

We propose the application of user-defined, extensible unification as the basic formalism for the implementation of constraint logic programming (CLP) languages. The close connection between unification theory and CLP justifies the step to make this link explicit and, particularly, operational.

The idea with extensible unification is that the user identifies the set of interpreted functors through the provision of a signature. The unification semantics of terms built from interpreted functors are specified by predicates, written in the language whose unification part is to be extended.

If CLP languages are implemented via extensible unification, they will inherit the capability of being extended on the very same basis, leading to the attractive construction of towers of (metacircular) CLP languages.

2 Extensible Unification

Extensible unification is realized by allowing for user-defined extensions to the unification algorithm of a unification based language. By means of declarations we identify a set of distinguished, interpreted functors. The unification semantics of interpreted functors is made operational through the provision of a corresponding unification algorithm. To add a catch to this classical situation, we proposed to use the language whose unification semantics we are just about to implement for the realization of the unification algorithm proper [5]. In terms of the low-level implementation, extensible unification can be provided by several means. Each of the two possibilities we relate in [6], has at least the following features:

[1]Austrian Research Institute for Artificial Intelligence, and
Department of Medical Cybernetics and Artificial Intelligence
University of Vienna, Freyung 6, A-1010 Vienna, Austria
Email: `christian@ai.univie.ac.at`.

- We can specify the meaning of interpreted terms in a high level language where implementation approaches specification.
- We have a substitute for 'destructive' updates that is logically sound.
- The declarative maintenance of attributed equivalence classes, which is useful to address global aspects in solvers and/or unification algorithms, leads quite naturally to an object oriented approach to the task of implementing solvers that operate on *systems* of equations.

To substantiate the claim about the fitness of the scheme, we made implementations of the basic mechanism, based on SICStus Prolog, and applications thereof available to the research community via anonymous ftp[2]. The provided implementations of CLP($\Re$) and CLP($\mathcal{Q}$) are at least as complete as other existing implementations we know of: They solve linear equations over rational or real valued variables, cover the lazy treatment of nonlinear equations, and use an incremental decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination) and provides for linear optimization.

3 Towers of (metacircular) CLP Languages

To furnish a language with extensible unification is a decision that has an attractive, transitive aspect: We will implement a level–n CLP language through the application of the level–$(n-1\dots0)$ languages for the implementation (of the unification algorithm) of the level-n language. Besides the straight forward motivation for this construction, there is the more ambitious goal to allow for reflective reasoning in the spirit of [4].

3.1 The Tower applied to AC Unification, for Example

AC and word unification algorithms [1, 8] typically require the solution of diophantine equations or systems thereof:

- Systems of equations over associative, commutative function symbols are abstracted to diophantine equation systems.
- Minimal and complete word unification: Word equations are expressed by generalized equations, which in turn comprise boundary relations, consisting of integer equations and inequalities.

The steps *decomposition, merging, mutation, detection of cycles, flattening, translation to a diophantine system* from [1], for example, are relatively simple symbol processing steps, covered by the level–0 language. In a concrete setting, we could use Prolog as level–0 language, and CLP($\mathcal{Z}$) as the next layer, in order to provide AC unification on level–3. Solving systems of homogeneous, linear diophantine equations is the basic task in CLP($\mathcal{Z}$). If the solutions are required to be positive, however, we can get away with CLP($\mathcal{Q}$), thanks to a result by Domenjoud [3].

[2] Directory sicstus at ftp.ai.univie.ac.at

3.2 Deciding the Satisfiability of Homogeneous Diophantine Systems through Convex Hull Construction in CLP($\mathcal{Q}$)

Given the matrix of a system like

$$A = \left[\begin{array}{rrrrr} 4 & -1 & 5 & 2 & -2 \\ -3 & 5 & 2 & -1 & 3 \end{array} \right]$$

we test whether $\vec{0} \in Conv(A^1, \ldots, A^5)$ in order to decide whether there is a solution to $AX = \vec{0}$. In CLP($\mathcal{Q}$) we can proceed as follows:

```
[Clp(Q)] ?- conv_hull([ [4,-3],[-1,5],[5,2],[2,-1],[-2,3] ], [X,Y]).

X =< 23/5+1/5*Y,
X >= 1-Y,
X >= -7/2+1/2*Y,
X =< 9-2*Y

% the actual test reads:

[Clp(Q)] ?- conv_hull([ [4,-3],[-1,5],[5,2],[2,-1],[-2,3] ], [0,0]).
no
```

Of course we need a CLP($\mathcal{Q}$) program to execute the query against. The convex hull of a set of vectors V is defined as:

$$Conv(V_1, \ldots, V_k) = \{\lambda_1 V_1 + \ldots + \lambda_k V_k | \forall i : \lambda_i \geq 0 \wedge \sum_{j=1}^{k} \lambda_j = 1\}$$

The following CLP($\mathcal{Q}$) program implements this declaration. There is no artificial limit on the number of vectors or their dimension. It can be used to compute the convex hull of a set of vectors, or to test for the membership of points in the hull. The former application is computationally hard because of the (quantifier) elimination of the λ's [7], the latter requires just a decision algorithm for polyhedral sets like Phase-I of the Simplex algorithm [2].

Therefore, Domenjoud's test $\vec{0} \in Conv(V_1, \ldots, V_k)$ runs efficiently in CLP($\mathcal{Q}$). We could even partially evaluate the general convex hull program to formally derive a specialized, possibly more efficient version, that implements only the test.

```
conv_hull( Points, Xs) :-
  colsums( Points, Lambdas, Xs),
  polytope( Lambdas).

colsums( [],         [],     []).
colsums( [S0|Rest], [L|Ls], Sums) :-
  mult_row( S0, L, S1),
  colsums_rest( Rest, Ls, S1, Sums).

mult_row( [],     _, []).
mult_row( [X|Xs], K, [K*X|Xs1]) :-
  mult_row( Xs, K, Xs1).
```

```
polytope( Xs) :-
  positive_sum( Xs, 1).

positive_sum( [],     0).
positive_sum( [X|Xs], X+Sum) :-
  X >= 0,
  positive_sum( Xs, Sum).
```

```
colsums_rest( [],         [],     S1, S1).
colsums_rest( [Ps|Rest], [K|Ks], S1, S3) :-
  colsum_row( Ps, K, S1, S2),
  colsums_rest( Rest, Ks, S2, S3).

colsum_row( [],     _, [],     []).
colsum_row( [P|Ps], K, [S|Ss], [K*P+S|Ss1]) :-
  colsum_row( Ps, K, Ss, Ss1).
```

When a system $AX = \vec{0}$ has solutions, we can enumerate the vertices of the λ polytope. Although the vertices will be rational in general, they can be transformed into integer vertices. The members of this set $S_0(A)$ are known as the minimal solutions with minimal carrier. Example:

$$A = \begin{bmatrix} 6 & 4 & -5 & -4 & -1 \\ 3 & 5 & 2 & -2 & -4 \end{bmatrix}$$

```
[Clp(Q)] ?- conv_hull([[6,3],[4,5],[-5,2],[-4,-2],[-1,-4]],[0,0]).
yes

[Clp(Q)] ?- colsums([[6,3],[4,5],[-5,2],[-4,-2],[-1,-4]],Lambdas,[0,0]),
            polytope( Lambdas),
            vertices( Lambdas, S0).
S0 = [
      [0,1/3,1/6,0,1/2],          % [ 0, 2, 1, 0, 3]
      [0,14/37,0,11/37,12/37],    % [ 0,14, 0,11,12]
      [2/5,0,0,3/5,0],            % [ 2, 0, 0, 3, 0]
      [11/35,0,3/10,0,27/70]      % [22, 0,21, 0,27]
     ]
```

Once we have $S_0(A)$, we can compute the remaining solutions to $AX = \vec{0}$, $X \in N^n$ as positive, rational linear combinations of independent vectors in $S_0(A)$.

Acknowledgements

This work was supported by the Austrian Federal Ministry of Science and Research.

References

[1] Adi M., Kirchner C.: AC-Unification Race: The System Solving Approach, Implementation and Benchmarks, to appear in the Journal of Symbolic Computation, 1992.

[2] Dantzig G.B.: *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[3] Domenjoud E.: Solving Systems of Linear Diophantine Equations: An Algebraic Approach, *in Tarlecki A.(ed): Proc. 16th Int. Symp. on Mathematical Foundations of Computer Science, LNCS520, Springer*, pp.141-150, 1991.

[4] Giunchiglia F., Traverso P.: Reflective Reasoning with and between a Declarative Metatheory and the Implementation Code, in *Proceedings of the 12 International Conference on Artificial Intelligence*, Morgan Kaufmann Publishers, San Mateo, CA, pp.111-117, 1991.

[5] Holzbaur C.: Specification of Constraint Based Inference Mechanisms through Extended Unification, Institute of Medical Cybernetics and AI, University of Vienna, Dissertation, 1990.

[6] Holzbaur C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification, in Bruynooghe M. and Wirsing M.(eds.), Programming Language Implementation and Logic Programming, Springer, LNCS 631, pp.260-268, 1992.

[7] Huynh T., Lassez J.L.: Practical Issues on the Projection of Polyhedral Sets, IBM Research Division, RC 15872 (#70560), 1990.

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

A Complete Transformation System for Polymorphic Higher-Order Unification

Ullrich Hustadt[1]

1 Introduction

We consider unification in the polymorphically typed lambda calculus $\mathcal{L}^{Poly}$. As stated in [2] type variables cause new problems through the possibility of type instantiation. Assume we wanted to use the transformation system $\mathcal{HT}$ defined in [3] for unification in $\mathcal{L}^{Poly}$. Then we have to face the problem that the projection rule is now infinitary branching. So even preunification is not practical anymore.

I propose a transformation system $\mathcal{HPT}$ that is based on the notions of *matrix expanders* and *selectors* instead of the notion of partial bindings. As a result, its search space is only finitely branching.

2 Basic Notions

Let $\mathcal{T}_0$ be a set of *base types* and $\mathcal{V}_\mathcal{T}$ a set of *type variables*. The set of *types* $\mathcal{T}$ is inductively defined as the smallest set containing $\mathcal{T}_0$ and $\mathcal{V}_\mathcal{T}$ such that if $T_1, T_2 \in \mathcal{T}$ then $(T_1 \to T_2) \in \mathcal{T}$.

To define the set $\mathcal{L}^{Poly}$ of polymorphically typed lambda terms, we use the usual rules for term formation in the simply typed lambda calculus but use polymorphic types instead of simple types. The type variables have no special meaning in term formation. They act like type constants. The set of all type variables occurring in M is $FV^{\mathrm{Type}}(M)$. The set of variables occurring in M is $FV(M)$. The β-normal form of a term M in $\mathcal{L}^{Poly}$ is $M\downarrow_\beta$. The η-expanded form of a term M in β-normal from is $\eta[M]$. $\mathcal{L}^{Poly}_\eta$ is the set of polymorphically typed lambda terms in η-expanded form.

A *polymorphic substitution* is a pair $\sigma = \langle \sigma_1, \sigma_2 \rangle$ consisting of a substitution σ_1 from type variables to types and a substitution σ_2 from free variables to terms both defined in the usual way. The domain $\mathcal{DOM}(\sigma)$ of σ is the union of the domains of σ_1 and σ_2. The identity substition ι is the pair consisting of the identity substitution ι_1 on types and the identity substitution ι_2 on terms. A substitution σ is *normalized* if $\sigma(F) = \sigma(F)\downarrow_\beta$ for all

[1]Max-Planck-Institut for Computer Science, Im Stadtwald, 6600 Saarbrücken, Germany, Email: `hustadt@mpi-sb.mpg.de`.

term variables $F \in \mathcal{DOM}(\sigma)$. The *restriction* of a substitution σ to a set of variables Z, denoted $\sigma_{|Z}$, is the substitution

$$\sigma_{|Z}(F) = \begin{cases} \sigma(F), & \text{if } F \in Z \\ F, & \text{otherwise.} \end{cases}$$

A *system* S is a multiset of equation where each equation is a multiset of two terms in $\mathcal{L}^{Poly}$. A *unification problem in* $\mathcal{L}^{Poly}$ is an ordered pair $\langle \sigma, S \rangle$ where σ is a type substitution and S is a system such that $\mathcal{DOM}(\sigma) \cap FV^{\text{Type}}(S) = \emptyset$. The type substitution σ memorizes the type instantiation that has to be done to get a unifier of S.

An equation $M = N$ is in *solved form* in a unification problem $\langle \sigma, S \rangle$ if it is in the form $\eta[F{:}\,T] = N$ for some variable $F{:}\,T$ which occurs exactly once in U, and $F{:}\,T$ and N have the same type. A system S is *solved* if all of its pairs are solved. A unification problem $\langle \sigma, S \rangle$ is *solved* if S is solved.

With a system $S = \{F_1{:}\,T_1 = N_1, \ldots, F_n{:}\,T_n = N_n\}$ in solved form we associate a term substitution $\lceil S \rceil^{\text{SUB}} = \{F_1{:}\,T_1/N_1, \ldots, F_n{:}\,T_n/N_n\}$. This substitution is unique up to variable renaming. With a unification problem $\langle \sigma, S \rangle$ in solved form we associate a substitution $\lceil U \rceil^{\text{SUB}} = \langle \sigma, \lceil S \rceil^{\text{SUB}} \rangle$.

A normalized substitution θ is called a *unifier in* $\mathcal{L}^{Poly}$ of two terms M and N from $\mathcal{L}^{Poly}$ iff $\theta(M) \longleftrightarrow_{\beta\eta} \theta(N)$ holds. θ is a unifier of a system S in $\mathcal{L}^{Poly}$ iff it is a unifier of every equation in S and it is called a unifier of a unification problem $\langle \sigma, S \rangle$ iff it is a unifier of the system S and an instance of $\langle \sigma, \iota_2 \rangle$.

For two types T_1 and T_2 there always exists a unique most general unifier $\text{mgu}(\{T_1 = T_2\})$.

3 The Transformation System $\mathcal{HPT}$

The rules for the transformation system $\mathcal{HPT}$ for unification problems in $\mathcal{L}^{Poly}$ are the following. In the presentation of the rules of $\mathcal{HPT}$ it is assumed that all terms are kept in η-expanded form.

Trivial removal

$$\langle \sigma, \{M = M\} \cup S \rangle \Rightarrow \langle \sigma, S \rangle \qquad \mathcal{HPT}_1$$

Type unification

$$\begin{array}{c} \langle \sigma, \{M = N\} \cup S \rangle \\ \Downarrow \\ \langle \theta \circ \sigma, \theta(\{M = N\} \cup S) \rangle \end{array} \qquad \mathcal{HPT}_2$$

where

- M is a term of type A and N is term of type C,
- $A \neq C$, and
- $\theta = \text{mgu}(\{A = C\})$.

Head type unification

$$\begin{array}{c}\langle \sigma, \{\lambda\, \overline{x_k\colon A_k}.\, a\colon B(\overline{M_m}) = \lambda\, \overline{x_k\colon A_k}.\, a\colon D(\overline{N_m})\} \cup S\rangle \\ \Downarrow \\ \langle \theta \circ \sigma, \theta(\{\lambda\, \overline{x_k\colon A_k}.\, a\colon B(\overline{M_m}) = \lambda\, \overline{x_k\colon A_k}.\, a\colon D(\overline{N_m})\} \cup S)\rangle\end{array} \qquad \mathcal{HPT}_3$$

where

- $B \neq D$,
- $\theta = \mathrm{mgu}(\{B = D\})$.

Decomposition

$$\begin{array}{c}\langle \sigma, \{\lambda\, \overline{x_k\colon A_k}.\, a(\overline{M_m}) = \lambda\, \overline{x_k\colon A_k}.\, a(\overline{N_m})\} \cup S\rangle \\ \Downarrow \\ \langle \sigma, \bigcup_{1\leq i\leq m}\{\lambda\, \overline{x_k\colon A_k}.\, M_i = \lambda\, \overline{x_k\colon A_k}.\, N_i\} \cup S\rangle\end{array} \qquad \mathcal{HPT}_4$$

where a is some arbitrary atom.

Variable elimination

$$\begin{array}{c}\langle \sigma, \{\lambda\, \overline{x_k\colon A_k}.\, F(\overline{x_k}) = N\} \cup S\rangle \\ \Downarrow \\ \langle \sigma, \{\lambda\, \overline{x_k\colon A_k}.\, F(\overline{x_k}) = N\} \cup \{F/N\}(S)\rangle\end{array} \qquad \mathcal{HPT}_5$$

where

- F is a free variable,
- $F \in FV(S)$ and $F \notin FV(N)$, and
- $\mathrm{type}(F) = \mathrm{type}(N)$.

Binder-Expansion

$$\begin{array}{c}\langle \sigma, \{\lambda\, \overline{x_k\colon A_k}.\, F(\overline{M_m}) = \lambda\, \overline{x_k\colon A_k}.\, b(\overline{N_n})\} \cup S\rangle \\ \Downarrow \\ \langle \theta \circ \sigma, \eta[\theta(\{\lambda\, \overline{x_k\colon M_k}.\, F(\overline{M_m}) = \lambda\, \overline{x_k\colon M_k}.\, b(\overline{N_n}) \cup S\})]\rangle,\end{array}$$

where

- F is a free variable of type $(B_1, \ldots, B_m \to A_0)$,
- A_0 is a type variable, and
- $\theta = \{A_0/(C_1 \to C_2)\}$ for type variables C_1 and C_2.

Matrix-Expansion

$$\begin{array}{c}\langle \sigma, \{\lambda\, \overline{x_k\colon A_k}.\, F(\overline{M_m}) = \lambda\, \overline{x_k\colon A_k}.\, b(\overline{N_n})\} \cup S\rangle \\ \Downarrow \\ \langle \sigma, \{F = Q\} \cup \{F/Q\}(\{\lambda\, \overline{x_k\colon A_k}.\, F(\overline{M_m}) = \lambda\, \overline{x_k\colon A_k}.\, b(\overline{N_n})\} \cup S){\downarrow}\rangle,\end{array}$$

where

- F is a free variable of type $(B_1, \ldots, B_m \to A_0)$;
- b is an arbitrary atom of type $\mathrm{type}(a) = (D_1, \ldots, D_n \to A_0)$;
- Q is a variant of a matrix expander of type $(B_1, \ldots, B_m \to A_0)$, i.e.

$$M = \lambda\, \overline{y_m \colon B_m}.\, G(\overline{y_m}, H(\overline{y_l}))$$

  where G is a free variable of type $(B_1, \ldots, B_m, B \to A_0)$, B is a type variable, and H is a free variable of type $(B_1, \ldots, B_l \to B)$ with $l \leq m$.

Selection

$$\langle \sigma, \{\lambda\, \overline{x_k \colon T_k}.\, F(\overline{M_m}) = \lambda\, \overline{x_k \colon T_k}.\, b(\overline{N_n})\} \cup S \rangle$$
$$\Downarrow$$
$$\langle \theta \circ \sigma, \{\theta(F) = Q\} \cup \{\theta(F)/Q\}(\theta(\{\lambda\, \overline{x_k \colon T_k}.\, F(\overline{M_m}) = \lambda\, \overline{x_k \colon T_k}.\, b(\overline{N_n})\} \cup S))\!\downarrow\rangle,$$

where

- F is a free variable of type $(\overline{B_m} \to A_0)$;
- b is some arbitrary atom of type $(\overline{D_n} \to D_0)$;
- a is some arbitrary atom of type $(E_1, \ldots, E_k \to E_0)$, for some k and some l, $k \leq m - l$, $0 \leq l \leq m$, such that $(B_{l+1}, \ldots, B_m \to A_0)$ and $\mathrm{type}(a)$ have a most general unifier θ;
- Q is a variant of a selector appropriate for type $\theta((B_1, \ldots, B_m \to A_0))$, i.e.

$$Q = \eta[\theta(\lambda\, \overline{y_m \colon B_m}.\, a(y_{l+1}, \ldots, y_m))],$$

  such that $l \leq m$.

The transformation system $\mathcal{HPT}$ provides a correct and complete algorithm for unification in the polymorphically typed lambda calculus.

Theorem 1 *For every unification problem U there exists a set*

$$G = \{\lceil U' \rceil^{\mathrm{SUB}}\!\restriction_{FV(U)} \mid U \overset{*}{\Longrightarrow}_{\mathcal{HPT}} U' \text{ and } U' \text{ is in solved form}\}$$

which is a complete set of unifiers for U. Using some renaming substitution away from W this set is a complete set of unifiers for U away from W.

Proof: See [1].

The search space of $\mathcal{HPT}$ is finitely branching. The search space of $\mathcal{HT}$, in contrast, is infinitely branching. However, although we have transformed the infinitely branching search space into a finitely branching one, (unfortunately) we have not reduced the search space. To this end, the idea in [2] of introducing product types should be investigated. This amounts to developing a correct and complete unification algorithm for the λ-calculus with product types.

References

[1] Ullrich Hustadt. A complete transformation system for polymorphic higher-order unification. Technical Report MPI-I-91-228, Max-Planck-Institut for Computer Science, December 1991.

[2] Tobias Nipkow. Higher-Order Unification, Polymorphism and Subsorts. In S. Kaplan and M. Okada, editors, 1990. *Proceedings of the Second International Workshop on Conditional and Typed Rewriting Systems*, LNCS 516, Montreal, Canada. Springer-Verlag, pp. 229–237, 1990.

[3] W. Snyder and J. Gallier. Higher-Order Unification Revisited: Complete Sets of Transformations. *Journal of Symbolic Computation*, Vol. 8, pp. 101–140, 1989.

Sequential Signatures (Extended Abstract)

Delia Kesner [1]

An *order-sorted signature* $\Sigma = (\mathcal{S}, \leq, \mathcal{F})$ consists of a partially ordered set $(\mathcal{S}, \leq)$ of sort symbols and a set $\mathcal{F}$ of function symbols equipped with sort declarations $\{f : \sigma_1 \ldots \sigma_n \to \sigma\}_{f \in \mathcal{F}}$. *Order-sorted* terms are constructed from Σ and sets of $\mathcal{S}$-indexed variables $\{\mathcal{V}^\sigma\}_{\sigma \in \mathcal{S}} = \{x^\sigma, y^\sigma, z^\sigma, \ldots\}_{\sigma \in \mathcal{S}}$ while *partial order-sorted* terms are also constructed from Σ, but each $\mathcal{V}^\sigma$ is now a singleton $\{\Omega^\sigma\}$. The quasi-ordering $\sqsubseteq$ between terms (resp. partial terms) is defined by induction on terms as: $\Omega^\sigma \sqsubseteq t^\eta$ if $\eta \leq \sigma$ and $f(t_1 \ldots t_n) \sqsubseteq f(h_1 \ldots h_n)$ if and only if $\forall i = 1 \ldots n,\ t_i \sqsubseteq h_i$.

Sequentiality is a property of monotonic predicates (w.r.t the partial order $\sqsubseteq$) over partial terms, related to the possibility of *systematically* expanding any term step-by-step in order to turn the predicate true. This work is concerned with the sequentiality of *sort predicates* in order sorted algebras, where each sort predicate $Sort_\delta$ characterizes the partial terms of sort δ. Monotonicity of sort predicates guarantees that each time sorts decrease, there is more and more chance to well type terms.

Substitutions are a very natural mechanism to precise the sort informations when dealing with order-sorted terms. But in order-sorted systems, we do not only perform substitutions in order to refine sorts, but also *reductions* that are the computation steps. While substitutions always decrease the sorts of terms, reductions do not. In that case, sequentiality of sort predicates is no more useful to perform efficient type verifications, and this is the reason we restrict our attention to *sort decreasing systems*, where the sort information of terms may only be refined along the reduction process.

For example, consider the sort-decreasing rewriting system,

$$R : G(x : int) \to H(x : int)$$

where the signature Σ contains the following declarations:

$$\begin{array}{ll} F : nat \times int \to \sigma & G : int \to int \\ H : int \to nat & nat < int \end{array}$$

In this system, every well-sorted term $G(t)$ of sort int can be reduced to a term $H(t)$ of sort nat, refining in this way its sort information.

[1] INRIA Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex and LRI and CNRS UA 410, Bât 490, Université de Paris-Sud, 91405 ORSAY Cedex, France, Email: `kesner@lri.lri.fr`.

The idea of sequentializing the type checking is that terms will need to be evaluated as far as *necessary* in order to satisfy a subsort constraint. In general, a few computation steps could be sufficient, without reducing terms to full normal forms. For example, let $t = F(G(3), G(4))$. Since t is ill-sorted, we have to perform some reductions inside it in order to get a well-typed term. If $G(3)$ and $G(4)$ are both the redexes selected by the reduction strategy, the reduction of $G(4)$ will not give more information in order to type the term (it is not necessary), while the reduction of $G(3)$ will be helpful (and even necessary) for that purpose. So, $G(3)$ has to be selected, $G(3) \rightarrow H(3)$ implies $t \rightarrow t' = F(H(3), G(4))$ and t' is now a well-sorted term, although it is not in normal form.

Sequential sort predicates try to identify positions of terms, called *directions*, where it is strictly necessary to refine the sort information in order to type those terms. Reduction is then performed over subterms at positions that are directions, and so wasteful computations are avoided in this way.

The motivation to consider sort predicates over partial terms rather than terms is that we can represent any term t as a *partial term* t_Ω, where each redex h of sort ρ appearing in t is replaced by an Ω^ρ if it has been selected by the reduction strategy. In our example, $t_\Omega = F(\Omega^{int}, \Omega^{int})$.

Formally, we say that a position u is a direction of a predicate $Sort_\delta$ at a given partial term t if and only if t/u is an Ω (a position where we need to perform a reduction) and for every h such that $t \sqsubseteq h$ and $Sort_\delta(h) = true$, $h/u \sqsubset t/u$. In our example the first occurrence of Ω^{int} in t_Ω is a direction of $Sort_\sigma$ while the second is not. $Sort_\delta$ is said to be sequential if and only if for every partial term t such that $Sort_\delta(t) = false$ but it is compatible with δ, there is a direction of $Sort_\delta$ at t. Finally, a signature Σ is said to be sequential if and only if for every sort δ in Σ, $Sort_\delta$ is sequential.

In order to decide sequentiality of sort predicates, the existence of directions at every partial term has to be verified. In general, there is an infinite set of partial terms and so, decidability of sequentiality becomes a non trivial problem. In this talk we provide a decision procedure for sequentiality of signatures that searches for directions only at some partial terms of height 1. The main theorem guarantees that this is sufficient in order to decide the existence of directions at any term.

Finally, we provide a compilation scheme which allow to efficiently decrease the sort information of any term. Our characterization of signatures becomes in this way a necessary and sufficient condition in order to perform efficient type verifications in order-sorted algebras.

Narrowing and Basic Forward Closures

Stefan Kurtz[1]

We will consider unification problems in equational theories, i.e. theories, which can be axiomatized by a set of universally quantified equational axioms. A very general result in this field has been obtained by Fay (cf. [2]), who describes a universal unification procedure, called narrowing, and proves its completeness for every equational theory that is represented by a convergent (i.e. Church-Rosser and terminating) term rewriting system. Technically, the narrowing algorithm combines syntactic unification and rewriting. To perform a narrowing step on a term means to instantiate it, such that it becomes reducible by a term rewriting rule, and then to reduce it by this rule.

Hullot (cf. [6]) presents an improved unification procedure, called basic narrowing. The idea of this algorithm is to reduce the search space by restricting narrowing steps to subterms not introduced by a substitution. An improvement of basic narrowing was given by Herold (cf. [5]), who noticed, that after a basic narrowing step at a position p one can also discard all positions, which are left of p. Herold calls this unification procedure left-to-right basic narrowing. The corresponding narrowing relation is introduced in Definition 1.

Definition 1 Let $\mathcal{R}$ be a term rewriting system. Let

$$\begin{aligned} shadow &: \mathbb{N}_+^* \longrightarrow \mathcal{P}(\mathbb{N}_+^*) \text{ and} \\ light &: \mathbb{N}_+^* \times \mathcal{T}(\mathcal{F},\mathcal{V}) \longrightarrow \mathcal{P}(\mathbb{N}_+^*) \end{aligned}$$

be functions, such that

$$\begin{aligned} shadow(p) &= \{p' \mid p \leq p' \vee p' \lhd p\} \text{ and} \\ light(p,r) &= \{p.p' \mid p' \in \mathcal{FP}os(r)\}, \end{aligned}$$

where the relation $\lhd$ is the lexicographic ordering on disjoint positions. The inference rule for left-to-right basic narrowing with $\mathcal{R}$ is defined as follows:

$$\frac{(s,\xi,P\cup\{p\})}{(\sigma(s[p\leftarrow r]),\sigma(\xi),(P\setminus shadow(p))\cup light(p,r))} \quad \text{if} \quad \begin{array}{l} \exists l\rightarrow r\in\mathcal{R},\exists\sigma\in\mathcal{S}: \\ \sigma=\mathcal{MGU}(s/p,l)\end{array}$$

[1]Universität Bielefeld, Technische Fakultät, Postfach 100 131, W-4800 Bielefeld 1, Germany. E-mail: kurtz@techfak.uni-bielefeld.de.

Theorem 1 If $\mathcal{R}$ is convergent, then left-to-right basic narrowing with $\mathcal{R}$ is complete. $\square$

This completeness result was first given by Herold (cf. [5]). A detailed proof of Theorem 1 can also be found in [7].

Forward closures are a common notion in the field of term rewriting systems (cf. [1, 4]). They can be seen as a result of a partial evaluation process of the narrowing relation. Our idea is to restrict this process to the basic narrowing relation, which leads to the notion of basic forward closures.

Definition 2 The basic forward closure $\mathcal{R}^+$ of a term rewriting system $\mathcal{R}$ is defined as follows: $\xi(l') \to r \in \mathcal{R}^+$ if and only if there is a term rewriting rule $l' \to r' \in \mathcal{R}$ and a left-to-right basic narrowing derivation with $\mathcal{R}$ from $(r', id_{\mathcal{V}}, \mathcal{FP}os(r'))$ to (r, ξ, P). $\square$

Remarks: For all term rewriting systems $\mathcal{R}$ the following is true:

1. $\mathcal{R}^+$ is a term rewriting system, such that $\mathcal{R} \subseteq \mathcal{R}^+$ and $\to_{\mathcal{R}}^* = \to_{\mathcal{R}^+}^*$.
2. There are term rewriting systems $\mathcal{R}$, such that $\mathcal{R}^+$ is infinite. (Consider for example $\mathcal{R} = \{f(f(x,y),z) \to f(x, f(y,z))\}$.)
3. It is not decidable, if $\mathcal{R}^+$ is finite.
4. In general $\mathcal{R}^+$ is a proper subset of the forward closure of $\mathcal{R}$. For the term rewriting system $\mathcal{R} = \{f(x) \to h(x,x), h(a,y) \to y, a \to b\}$, we have $\mathcal{R}^+ = \mathcal{R} \cup \{f(a) \to a\}$, whereas the forward closure of $\mathcal{R}$ is $(\mathcal{R}^+)^+ = \mathcal{R}^+ \cup \{f(a) \to b\}$. $\square$

If one uses $\mathcal{R}^+$ in a narrowing procedure, one can also discard all the positions, which are introduced by the right-hand sides of the term rewriting rules, since these positions are exhausted in the construction of the basic forward closure. This leads to left-to-right bottom-up narrowing with $\mathcal{R}^+$.

Definition 3 Let $\mathcal{R}$ be a term rewriting system. The inference rule for left-to-right bottom-up narrowing with $\mathcal{R}^+$ is defined as follows:

$$\frac{(s, \xi, P \cup \{p\})}{(\sigma(s[p \leftarrow r]), \sigma(\xi), P \setminus shadow(p))} \quad \text{if} \quad \begin{array}{l} \exists l \to r \in \mathcal{R}^+, \exists \sigma \in \mathcal{S} : \\ \sigma = \mathcal{MGU}(s/p, l) \end{array}$$

$\square$

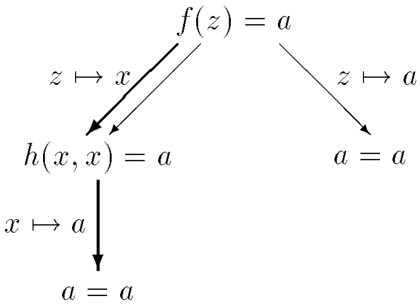
Theorem 2 If $\mathcal{R}$ is convergent, left-to-right bottom-up narrowing with $\mathcal{R}^+$ is complete. $\square$

A detailed proof of Theorem 2 can be found in [7]. It uses the fact that every leftmost-innermost rewriting derivation consists of a sequence of "chains". Since each of these "chains" can be contracted to one application of a rule from $\mathcal{R}^+$, it is easy to prove the completeness, using a lifting lemma for narrowing and rewriting derivations (cf. [8]).

Notice that in a left-to-right bottom-up narrowing derivation step the set of possible narrowing positions is decreased, which implies that left-to-right bottom-up narrowing with $\mathcal{R}^+$ terminates, if $\mathcal{R}^+$ is finite. This termination condition is closely related to the termination condition for basic narrowing, which was given by Hullot in [6]. Hullot proves: If there are no infinite narrowing derivations issuing from the right-hand sides of the term rewriting rules, the basic narrowing algorithm terminates. We conjecture that both termination conditions are equivalent, if $\mathcal{R}$ is terminating.

It is easy to see that left-to-right bottom-up narrowing with $\mathcal{R}^+$ is as efficient as left-to-right basic narrowing with $\mathcal{R}$. This is due to the tradeoff in non-determinism: We can forget the positions introduced by the right-hand sides, but we have to use the extra term rewriting rules in $\mathcal{R}^+$.

Example 1 Let $\mathcal{R} = \{f(x) \to h(x,x), h(a,y) \to y, a \to b\}$. Then we have $\mathcal{R}^+ = \mathcal{R} \cup \{f(a) \to a\}$. If we solve the goal by $f(z) = a$ we get the following "narrowing tree", in which the thick edges are produced by left-to-right basic narrowing and the thin edges are produced by left-to-right bottom-up narrowing.



In both procedures we need two narrowing steps to produce the complete solution set $\{\{z \mapsto a\}\}$. □

Notice that term rewriting systems with a finite basic forward closure are of practical relevance in the field of code generation. In [3], for example, an efficient bottom-up pattern matcher is described, which is based on this notion.

For a more detailed presentation of the results discussed here see [7].

References

[1] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[2] M. Fay. First-Order Unification in Equational Theories. In *Proceedings of the Fourth International Workshop on Automated Deduction*, pages 161–167, 1979.

[3] C.W. Fraser, R.R. Henry, and T.A. Proebsting. BURG—Fast Optimal Instruction Selection and Tree Parsing. User Manual, available via anonymous `ftp` from `kaese.cs.wisc.edu`, 1991.

[4] M. Hermann. Chain Properties of Rule Closures. *Formal Aspects of Computing*, 2:207–225, 1990.

[5] A. Herold. *Combination of Unification Algorithms in Equational Theories*. Ph.D. Thesis, Fachbereich Informatik, Universität Kaiserslautern, 1987.

[6] J.-M. Hullot. Canonical Forms and Unification. In *Proceedings of the Fifth Conference on Automated Deduction*, pages 318–334. Lecture Notes in Computer Science 87, Springer Verlag, 1980.

[7] S. Kurtz. Narrowing and Basic Forward Closures. Report Nr. 5, Abteilung Informationstechnik, Technische Fakultät, Universität Bielefeld, 1992.

[8] A. Middeldorp and E. Hamoen. Counterexamples to Completeness Results for Basic Narrowing. Report, IR-268, CWI, Amsterdam, 1991.

Tree Automata and Complement Problems in AC-like Theories

Denis Lugiez and Jean-Luc Moysset [1]

1 Introduction

Given an equational theory E, a term t, and a set of terms $R = \{t_1, \ldots, t_n\}$, to solve the complement problem $t \neq_E t_1 \ldots t \neq_E t_n$ is to find if there is a ground instance of t which is not a ground E-instance of any of the $t_i's$. We propose a new solution of this problem when some functions are associative and commutative and the $t_i's$ are linear. This solution relies on tree-automata which are powerful tools to recognize regular tree languages. We describe some extensions to other theories and to some non-linear cases. Moreover we extend a proof given in [3] relying on test-set for the following non-linear case: when flattening t and the $t_i's$, all occurrences of a non-linear variable are under the same node.

2 Solving linear AC-complement problems

We take for granted the definitions and notions on terms, equational theories, and tree automata.[2]

Our key result is surprisingly simple to state and prove:

Proposition 1 *Let t be a linear term, then the set of ground AC-instances of t is a regular tree language.*

The next theorem is a straightforward consequence of the previous property.

Theorem 1 *The complement problem $t \neq_{AC} t_1 \wedge \ldots \wedge t \neq_{AC} t_n$ where the t_i 's are linear terms is decidable.*

The inductive reducibility modulo AC of a linear term t for a left-linear term rewriting system is decidable.

[1] CRIN-INRIA BP 539 54506 Vandoeuvres-les-Nancy, FRANCE Email: `lugiez,jmoysset@loria.fr`.

[2] Proofs and missing definitions can be found in [4]

What's the trouble with non-linearity?

Before giving some applications and extensions of the previous result, we show which problem arises when we consider non-linear terms. Let F_{AC} consist of one symbol f , and let F_{NAC} consist of two constants 0 and 1 and let t be $f(x,x)$. Then the ground AC-instances of $f(x,x)$ are extremely difficult to recognize, some of these instances are trees such that the first son and its brother are not equal, even modulo AC. For instance $f(0, f(1, f(1, 0)))$ and $f(0, f(1, f(1, 0)))$ are ground AC-instances of $f(x,x)$. To know that a ground term (with root f) is an instance of $f(x,x)$, one must count $0's$ and $1's$ and there is no way to conclude before all $0's$ and $1's$ have been counted: the ground term is an AC-instance of $f(x,x)$ if there is an even number of $0's$ and an even number of $1's$. There is no way to design a tree automaton which can manage this. Moreover, there is a theoretical results which proves that tree automata cannot recognize ground AC-instances of non-linear terms: the inductive reducibility modulo AC property would be decidable, and this property was proved undecidable [2].

3 Extending the results

Our solution to linear complement problem can be extended in several ways.

3.1 To some non-linear cases

In [1], tree-automata are extended in order to allow equality tests between brothers. This class is closed under boolean operations and the emptiness property is decidable. We can use this result to get decidability results for non-linear complement problems and for the inductive reducibility modulo AC for non-left-linear term rewriting systems.

Definition 1 *The non-linearity of a term is strictly restricted iff for each non-linear variable x, there exist a position p such that all the occurrences of x occur at positions $p.i$ with i an integer, and the symbol at position p is not AC.*

Theorem 2 *The complement problem $t \neq_{AC} t_1 \wedge \ldots \wedge t \neq_{AC} t_n$ (resp. $\neq_A$) where the $t_i's$ are strictly restricted, is decidable.*

The inductive reducibility modulo AC (resp. modulo A) of a strictly restricted term t for a term rewriting system with strictly restricted left-hand sides is decidable.

Proof. Since linearity is strictly restricted, the definition and proofs in [1] need few changes only. The syntactical equality $=$ is replaced by $=_{AC}$, moreover the automata that are used satisfy the property *if t reaches some state, then all AC-variants of t also reach this state*. Therefore the algorithm to decide the emptiness of the language accepted by automata with equality test modulo AC is a straightforward adaptation of that of [1].

3.2 To some other theories

The previous results can be easily extended to other theories. The first one is the AC1 theory, i.e AC with unity: for each AC-symbol f, there exists a constant e such that

$f(x,e) = x$. To handle this new equation, we add the rules $e \to q_e$ and $f(q,q_e) \to q$ and $f(q_e,q) \to q$. The second one is the associativity theory: we simply drop the rules handling commutativity and we introduce the rules for associativity at the right place. Therefore we get the theorem:

Theorem 3 *The complement problem $t \neq_{AC1} t_1 \wedge \ldots \wedge t \neq_{AC1} t_n$ (resp. $\neq_A$) where the t_i's are linear terms is decidable.*

The inductive reducibility modulo AC1 (resp. modulo A) of a linear term t for a left-linear term rewriting system is decidable.

This approach also works for ACI axioms,i.e for functions f which are AC and satisfy the idempotency axioms $f(x,x) = x$.

3.3 Using a abstract criteria

Actually, the previous extensions follow from the general theorem:

Theorem 4 *Let $\mathcal{L}$ be a class of tree-automata closed under the boolean operations and such that the emptiness of the language accepted by an automa of $\mathcal{L}$ is decidable, let E be a equational theory such that the set of ground E-instances of any term is the language accepted by some $\mathcal{A} \in \mathcal{L}$, then the complement problem and the inductive reducibility property in E are decidable.*

This theorem reduces a difficult problem (complement modulo a theory) to a much simpler one (prove that the set of ground E-instances of some term are recognizable) and suggests two fruitful directions of research. The first one is to consider tree automata which are more general than bottom-up tree automata, the second one is to design syntactical criteria on the theory E which will ensure the property required on the ground E-instances of a term.

4 Solving some non-linear AC problems

To conclude, we give the decidability of the complement problem modulo AC in a particular case: all the occurrences of a non-linear variable are under the same node, as described in the following definition.

Definition 2 *A flattened term s is restricted iff all occurrences of a non-linear variable are under the same node.*

And the the main theorem is:

Theorem 5 *Let t and t_i's be restricted flattened terms, then the complement problem $t \neq_{AC} t_1 \wedge \ldots \wedge t \neq_{AC} t_n$ is decidable.*

The proof generalizes the proof given in [3] in the case of one AC-symbol and constants and it relies on test-set techniques. Tree automata provide nice and simple proofs and test-set methods give tedious and difficult proofs. However, at the present time, we have stronger results with test-set methods than with tree automata (but we are currently investigating some new classes of tree automata which may be as powerful as the test-set approach).

References

[1] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *Proceedings of the 9th Symposium on Theoretical Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 161–172, 1992.

[2] D. Kapur, P. Narendran, D.J. Rosenkrantz, and H. Zhang. Sufficient-completeness, quasi-reducibility and their complexity. Technical report, State University of New York at Albany, 1987.

[3] E. Kounalis, D. Lugiez, and L. Pottier. A solution of the complement problem in associative commutative theories. In A.Tarlecki, editor, *16th International Symposium Mathematical Foundation of Computer Sciences*, volume 520 of *Lecture Notes in Computer Science*, pages 287–297. Springer-Verlag, 1991.

[4] D. Lugiez and J.L. Moysset. Complement and tree automata and tree automata in ac-like theories. Technical Report 92-R-175, CRIN, 1992.

Complexity of E-Unification Problems

Paliath Narendran[1]

Complexity issues in unification have been investigated a great deal since Paterson and Wegman published their linear-time algorithm for standard unification. E-unification, or unification in the presence of an equational theory E, is much more complicated, most of the problems being undecidable in general. Research has so far concentrated on two major issues: (i) *E-unifiability* where one only has to check whether there exists a unifier for the input terms, and (ii) computing a *complete* set of E-unifiers for terms especially when these sets are known to be always finite. In Kapur and Narendran (1989) we briefly surveyed the results and presented them in tabular form. The present talk updates that survey and discusses several significant open problems.

[1]Institute of Programming and Logics, Department of Computer Science, State University of New York at Albany, Albany, NY 12222, email: `dran@cs.albany.edu`.

Functional Unification of Higher-Order Patterns

Tobias Nipkow[1]

1 Background

Dale Miller [1] discovered a class of λ-terms which behave almost like first-order terms w.r.t. unification: unification is decidable and unifiable terms have most general unifiers which are easy to compute. His results were taken up by Pfenning [3], who extended them to the Calculus of Constructions, and by Nipkow [2] who reformulated and used them in the context of rewrite systems over simply typed λ-terms.

The unification algorithm presented by Miller applies to quantified λ-terms and is described informally (although rigorously); the algorithms by Pfenning and Nipkow are formulated at a fairly high level. The purpose of this paper is to present an efficient implementation which follows the structure of unification for first-order terms. We use the notation of [2]. In particular the following convention is adhered to: s and t denote terms, θ substitutions, F, G and H free variables, x, y and z bound variables, a and b atoms (i.e. variables or constants), c constants, and ss and ts term lists. The notation $a(\overline{s_n})$ is short for the iterated application $((\ldots(a\ s_1)\ldots)\ s_n)$. Binary application of λ-terms is written $s.t$ in order to avoid confusion with application in the programming language used for expressing the algorithms. We follow the convention of keeping bound and free variables disjoint. Hence the language of λ-terms is defined by the following grammar:

$$t \quad = \quad F \mid x \mid c \mid \lambda x.t \mid (t_1.t_2)$$

Note that this grammar also allows so called "loose" bound variables as in $\lambda x.y$, where y is a bound variable without a corresponding enclosing binder. Such loose bound variables will not normally occur in our terms, except at intermediate stages of a computation.

A term t in β-normal form is a (**higher-order**) **pattern** if every free occurrence of a variable F is in a subterm $F(\overline{u_n})$ of t such that $\overline{u_n}$ is η-equivalent to a list of distinct bound variables.

Examples of higher-order patterns are F, $\lambda x.F(\lambda z.x(z))$ and $\lambda x,y.F(y,x)$, examples of non-patterns are $F(c)$, $\lambda x.F(x,x)$ and $\lambda x.F(F(x))$. In the sequel all terms are either implicitly assumed to be patterns or are patterns by construction.

[1]Institut für Informatik, Technische Universität München, Postfach 20 24 20, W-8000 München 2, Germany. Email: `Tobias.Nipkow@Informatik.TU-Muenchen.De`.

We present the development of a functional program for unification of patterns. Starting with a high-level description we gradually eliminate all features which have no counterpart in ordinary functional languages. Eventually we arrive at a formulation which is directly executable. A translation of this algorithm into Standard ML is contained in an extended version of this abstract, which is available directly from the author and which also treats de Bruijn's representation of λ-terms.

All algorithms are written in a mixture of simplified Standard ML and mathematics. In particular we rely on the following basic combinator:

$$
\begin{array}{lcl}
foldl\ f\ a\ [] & = & a \\
foldl\ f\ a\ (x :: xs) & = & foldl\ f\ (f(a,x))\ xs
\end{array}
$$

2 The first version

We start the development with two basic assumptions:

α : α-equivalent terms are identified.

η : Terms are simply typed and in η-expanded form, except for the arguments of free variables, which are in η-normal form, i.e. bound variables.

Both assumptions are relaxed during the development.

The first algorithm is a serialization of the inference rules in [2] and is close to Miller's algorithm.

$$
\begin{array}{lcl}
unif\ \theta\ (s,t) & = & \texttt{case}\ (s\theta{\downarrow_\beta}, t\theta{\downarrow_\beta})\ \texttt{of} \\
& & \quad (\lambda\overline{x_k}.F(\overline{y_m}), \lambda\overline{x_k}.G(\overline{z_n})) \Rightarrow flexflex(F,\overline{y_m},G,\overline{z_n},\theta) \\
& & \quad (\lambda\overline{x_k}.t, \lambda\overline{x_k}.F(\overline{y_m})) \Rightarrow flexrigid(F,\overline{y_m},t,\theta) \\
& & \quad (\lambda\overline{x_k}.F(\overline{y_m}), \lambda\overline{x_k}.t) \Rightarrow flexrigid(F,\overline{y_m},t,\theta) \\
& & \quad (\lambda\overline{x_k}.a(\overline{s_m}), \lambda\overline{x_k}.b(\overline{t_n})) \Rightarrow rigidrigid(a,\overline{s_m},b,\overline{t_n},\theta)
\end{array}
$$

$$
\begin{array}{lcl}
flexflex(F,\overline{y_m},G,\overline{z_n},\theta) & = & \texttt{if}\ F = G\ \texttt{then}\ flexflex1(F,\overline{y_m},\overline{z_n},\theta) \\
& & \texttt{else}\ flexflex2(F,\overline{y_m},G,\overline{z_n},\theta)
\end{array}
$$

$$
\begin{array}{lcl}
flexflex1(F,\overline{y_m},\overline{z_n},\theta) & = & \texttt{if}\ m \neq n\ \texttt{then fail} \\
& & \texttt{else}\ \{F \mapsto \lambda\overline{y_m}.H([y_i \mid 1 \le i \le m \wedge y_i = z_i])\} \circ \theta
\end{array}
$$

$$
\begin{array}{lcl}
flexflex2(F,\overline{y_m},G,\overline{z_n},\theta) & = & \texttt{let}\ \{\overline{x_k}\} = \{\overline{y_m}\} \cap \{\overline{z_n}\} \\
& & \texttt{in}\ \{F \mapsto \lambda\overline{y_m}.H(\overline{x_k})\} \circ \{G \mapsto \lambda\overline{z_n}.H(\overline{x_k})\} \circ \theta
\end{array}
$$

$$
\begin{array}{lcl}
flexrigid(F,\overline{y_m},t,\theta) & = & \texttt{if}\ F \in \mathcal{FV}(t)\ \texttt{then fail} \\
& & \texttt{else}\ proj\ \{\overline{y_m}\}\ (\{F \mapsto \lambda\overline{y_m}.t\} \circ \theta)\ t
\end{array}
$$

$$
\begin{array}{lcl}
rigidrigid(a,ss,b,ts,\theta) & = & \texttt{if}\ a \neq b\ \texttt{orelse}\ |ss| \neq |ts|\ \texttt{then fail} \\
& & \texttt{else}\ foldl\ unif\ \theta\ (zip\ ss\ ts)
\end{array}
$$

$$
\begin{array}{lcl}
zip\ (x :: xs)\ (y :: ys) & = & (x,y) :: (zip\ xs\ ys) \\
zip\ []\ [] & = & []
\end{array}
$$

$$
\begin{array}{lcl}
proj\ V\ \theta\ s & = & \texttt{case}\ s\theta{\downarrow_\beta}\ \texttt{of} \\
 & & \quad \lambda x.t \Rightarrow proj\ (V \cup \{x\})\ \theta\ t \\
 & & \quad c(\overline{s_m}) \Rightarrow foldl\ (proj\ V)\ \theta\ \overline{s_m} \\
 & & \quad x(\overline{s_m}) \Rightarrow \texttt{if}\ x \in V\ \texttt{then}\ foldl\ (proj\ V)\ \theta\ \overline{s_m}\ \texttt{else}\ \texttt{fail} \\
 & & \quad F(\overline{y_m}) \Rightarrow \texttt{let}\ \{\overline{z_n}\} = \{\overline{y_m}\} \cap V\ \texttt{in}\ \{F \mapsto \lambda\overline{y_m}.H(\overline{z_n})\} \circ \theta
\end{array}
$$

The free variable H is assumed to be a *new* variable in each instance. The conditions $m \neq n$ and $|ss| \neq |ts|$ prepare the ground for untyped terms; for typed terms, $m = n$ and $|ss| = |ts|$ are implied by $F = G$ and $a = b$, respectively.

3 On-the-fly α-conversion

Let us now drop assumption α and take α-conversion (almost) seriously. We merely assume that in a term $\lambda x.s$, s does not contain a subterm $\lambda x.t$, i.e. nested abstractions bind distinct variables. If necessary, this assumption can be enforced by preprocessing.

$$
\begin{array}{lcl}
unif\ \theta\ (s,t) & = & \texttt{case}\ (s\theta{\downarrow_\beta}, t\theta{\downarrow_\beta})\ \texttt{of} \\
 & & \quad (\lambda x.s, \lambda y.t) \Rightarrow unif\ \theta\ (s, t\{y \mapsto x\}) \\
 & & \quad (s,t) \Rightarrow cases\ \theta\ (s,t)
\end{array}
$$

$$
\begin{array}{lcl}
cases\ \theta\ (F(\overline{y_m}), G(\overline{z_n})) & = & flexflex(F, \overline{y_m}, G, \overline{z_n}, \theta) \\
cases\ \theta\ (F(\overline{y_m}), t) & = & flexrigid(F, \overline{y_m}, t, \theta) \\
cases\ \theta\ (t, F(\overline{y_m})) & = & flexrigid(F, \overline{y_m}, t, \theta) \\
cases\ \theta\ (a(\overline{s_m}), b(\overline{t_n})) & = & rigidrigid(a, \overline{s_m}, b, \overline{t_n}, \theta)
\end{array}
$$

4 Implementing substitutions

Now we implement substitutions as association lists of variables and terms. Consequently we replace substitutions of the form $\{F \mapsto t\} \circ \theta$ by $(F,t) :: \theta$. The expression $s\theta{\downarrow_\beta}$ now becomes $devar\ \theta\ s$, where $devar$ is a "lazy" form of substitution application: only in $F(\ldots)$ is F replaced by θF; all other terms are left unchanged. Because $devar$ is lazy, $F \in \mathcal{FV}(t)$ becomes $occ\ F\ \theta\ t$:

$$
\begin{array}{lcl}
devar\ \theta\ (F(\overline{y_n})) & = & \texttt{case}\ assoc\ F\ \theta\ \texttt{of} \\
 & & \quad Some(\lambda\overline{x_n}.t) \Rightarrow devar\ \theta\ (t\{\overline{x_n \mapsto y_n}\}) \\
 & & \quad None \Rightarrow F(\overline{y_n}) \\
devar\ \theta\ s & = & s
\end{array}
$$

$$
\begin{array}{lcl}
assoc\ F\ ((G,t) :: \theta) & = & \texttt{if}\ F = G\ \texttt{then}\ Some(t)\ \texttt{else}\ assoc\ F\ \theta \\
assoc\ F\ [] & = & None
\end{array}
$$

$$
\begin{array}{lcl}
occ\ F\ \theta\ G & = & (F = G)\ \texttt{orelse}\ \texttt{case}\ assoc\ G\ \theta\ \texttt{of} \\
 & & \qquad Some(s) \Rightarrow occ\ F\ \theta\ s \\
 & & \qquad None \Rightarrow false \\
occ\ F\ \theta\ (s.t) & = & occ\ F\ \theta\ s\ \texttt{orelse}\ occ\ F\ \theta\ t \\
occ\ F\ \theta\ (\lambda x.s) & = & occ\ F\ \theta\ s \\
occ\ F\ \theta\ \_ & = & false
\end{array}
$$

5 On-the-fly η-expansion

The following modifications remove the need to work with η-expanded simply typed terms. This is not just relevant for applications to untyped terms. It also paves the way for terms containing type variables which may get instantiated during unification because such instantiations may require further η-expansions.

However, we still retain the assumption that all arguments to free variables must be bound variables. This merely simplifies the notation and can be relaxed, for example, by a preprocessing phase that η-normalizes the arguments of free variables.

$$
\begin{array}{lcl}
\mathit{unif}\ \theta\ (s,t) & = & \texttt{case}\ (\mathit{devar}\ \theta\ s, \mathit{devar}\ \theta\ t)\ \texttt{of} \\
 & & \quad (\lambda x.s, \lambda y.t) \Rightarrow \mathit{unif}\ \theta\ (s, t[x/y]) \\
 & & \quad (\lambda x.s, t) \Rightarrow \mathit{unif}\ \theta\ (s, t.x) \\
 & & \quad (s, \lambda x.t) \Rightarrow \mathit{unif}\ \theta\ (s.x, t) \\
 & & \quad (s,t) \Rightarrow \mathit{cases}\ \theta\ (s,t)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{devar}\ \theta\ s & = & \texttt{case}\ \mathit{strip}\ s\ \texttt{of} \\
 & & \quad (F, ys) \ \Rightarrow\ \texttt{case}\ \mathit{assoc}\ F\ \theta\ \texttt{of} \\
 & & \qquad\qquad \mathit{Some}(t) \ \Rightarrow\ \mathit{devar}\ \theta\ (\mathit{red}\ t\ ys) \\
 & & \qquad\qquad \mathit{None} \ \Rightarrow\ s \\
 & & \quad \_ \ \Rightarrow\ s
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{red}\ (\lambda x.s)\ (y :: ys) & = & \mathit{red}\ (s[y/x])\ ys \\
\mathit{red}\ s\ (y :: ys) & = & \mathit{red}\ (s.y)\ ys \\
\mathit{red}\ s\ [] & = & s
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{strip}\ t & = & \texttt{let}\ \mathit{strip}\ (f.t)\ ts\ =\ \mathit{strip}\ f\ (t :: ts) \\
 & & \qquad \mathit{strip}\ t\ ts\ =\ (t, ts) \\
 & & \texttt{in}\ \mathit{strip}\ t\ []
\end{array}
$$

6 Pattern-matching

As a final step towards an executable algorithm we remove all occurrences of (programming language) patterns of the form $a(\overline{s_m})$ in favour of the destructor *strip*:

$$
\begin{array}{lcl}
\mathit{cases}\ \theta\ (s,t) & = & \texttt{case}\ (\mathit{strip}\ s, \mathit{strip}\ t)\ \texttt{of} \\
 & & \quad ((F,\overline{y_m}),(G,\overline{z_n})) \ \Rightarrow\ \mathit{flexflex}(F,\overline{y_m},G,\overline{z_n},\theta) \\
 & & \quad ((F,\overline{y_m}),\_) \ \Rightarrow\ \mathit{flexrigid}(F,\overline{y_m},t,\theta) \\
 & & \quad (\_,(F,\overline{y_m})) \ \Rightarrow\ \mathit{flexrigid}(F,\overline{y_m},s,\theta) \\
 & & \quad ((a,\overline{s_m}),(b,\overline{t_n})) \ \Rightarrow\ \mathit{rigidrigid}(a,\overline{s_m},b,\overline{t_n},\theta)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{proj}\ V\ \theta\ s & = & \texttt{case}\ \mathit{strip}(\mathit{devar}\ \theta\ s)\ \texttt{of} \\
 & & \quad (\lambda x.t, \_) \Rightarrow \mathit{proj}\ (V \cup \{x\})\ \theta\ t \\
 & & \quad (c, ss) \Rightarrow \mathit{foldl}\ (\mathit{proj}\ V)\ \theta\ ss \\
 & & \quad (x, ss) \Rightarrow \texttt{if}\ x \in V\ \texttt{then}\ \mathit{foldl}\ (\mathit{proj}\ V)\ \theta\ ss\ \texttt{else}\ \texttt{fail} \\
 & & \quad (F, \overline{y_m}) \Rightarrow \texttt{let}\ \{\overline{z_n}\} = \{\overline{y_m}\} \cap V\ \texttt{in}\ (F, \lambda\overline{y_m}.H(\overline{z_n})) :: \theta
\end{array}
$$

References

[1] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, pages 253–281. LNCS 475, 1991.

[2] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349, 1991.

[3] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 74–85, 1991.

Undecidability of the Horn-Clause Implication Problem

Jerzy Marcinkowski[1] and Leszek Pacholski[2]

1 Introduction

In this paper we prove that the problem "given two Horn clauses $\mathcal{H}_1 = (\alpha_1 \wedge \alpha_2 \rightarrow \beta)$ and $\mathcal{H}_2 = (\gamma_1 \wedge ... \wedge \gamma_k \rightarrow \delta)$, where $\alpha_i, \beta, \gamma_i, \delta$ are atomic formulas, decide if $\mathcal{H}_2$ is a consequence of $\mathcal{H}_1$" is not recursive, thus solving one of the last open decidability problems concerning formulas in pure predicate logic (i.e. without equality symbol). It follows from a result of M. Schmidt-Schauß [6], that the problem, if a Horn clause $(\alpha \rightarrow \beta)$, with α, β atomic, implies another Horn clause, is decidable.

Problems concerning decidability of restricted classes of quantificational formulas have been studied since the thirties by W. Ackermann, P. Bernays, J. Büchi, K. Gödel, W. Goldfarb, Y. Gurevich, L. Kalmár, H.R. Lewis, M. Schönfinkel, H. Wang and many others (see [2]). Recently several papers on the clause implication problem have been written by researchers being motivated by problems in artificial intelligence. (see e.g. [6] [3]).

We use standard notation and terminology. An *atomic formula* is an expression of the form $Q(t_1, ..., t_k)$, where Q is a relation symbol and $t_1, ..., t_k$ are terms. A *literal* is an atomic formula or a negation of an atomic formula. A *clause* is a disjunction of literals. A *ground* clause (term) is a clause (term) without variables. A clause with n literals is called *a n-clause,* a *unit* clause is a 1-clause. A *Horn clause* is a clause with at most one non-negated literal, or equivalently a formula of the form $\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_k \rightarrow \beta$, where $\alpha_1, \alpha_2, ..., \alpha_k, \beta$ are atomic formulas. The Horn clause above a *k-Horn clause.* Let $\mathcal{A}$ and $\mathcal{B}$ be two clauses. We say that $\mathcal{A}$ implies $\mathcal{B}$, if $\mathcal{A} \vdash \mathcal{B}$, or equivalently, if $(\forall x_1)...(\forall x_m)\mathcal{A} \rightarrow (\forall y_1)...(\forall y_n)\mathcal{B}$ holds, where $x_1, ..., x_m, y_1, ..., y_n$ are all variables in $\mathcal{A}$ and $\mathcal{B}$ (we can and do assume, that $\mathcal{A}$ and $\mathcal{B}$ have disjoint sets of variables). The clause implication problem is equivalent to the satisfiability problem for the set formulas consisting of the clause $\mathcal{A}$ and ground unit clauses obtained by the negation of $\mathcal{B}$.

A tree $\mathcal{T}$ is a *k-tree* if each non-leaf node of $\mathcal{T}$ has exactly k sons. We identify k-trees with sets of words over the alphabet $\{1, 2, ..., k\}$, which are closed under prefixes, i.e. if

[1]Institute of Computer Science, University of Wrocław, Przesmyckiego 20, 51-151 Wrocław, Poland, Email: `jmarcink@plwruw11.bitnet`.

[2]Institute of Mathematics, Polish Academy of Sciences, Kopernika 18, 51-617 Wrocław, Poland, Email: `jmarcink@plwruw11.bitnet`.

$w \in \mathcal{T}$ and v is a prefix of w, then $v \in \mathcal{T}$. Given a tree $\mathcal{T}$, we put $(\mathcal{T})_w = \{v : \exists u \in \mathcal{T}, u = wv\}$, i.e. $(\mathcal{T})_w$ is the subtree of $\mathcal{T}$ rooted at w.

It is easy to notice that to study the problem, if one Horn clause implies another, it suffices to assume that all Horn clauses under consideration are of the form

$$\mathcal{H} = (Q(\Gamma_1) \wedge Q(\Gamma_2) \wedge \ldots \wedge Q(\Gamma_k) \to Q(\Gamma)) \tag{1}$$

where $\Gamma, \Gamma_1, \Gamma_2, ..., \Gamma_k$ are n-tuples of terms, for n being the arity of Q. Again, it is easy to notice that we can furthermore assume, that the arity of Q is 1, so $\Gamma, \Gamma_1, \Gamma_2, ..., \Gamma_k$ are either variables or terms with the same main symbol.

For a given k-Horn clause $\mathcal{H}$, an *$\mathcal{H}$-derivation* is a finite labelled k-tree $\mathcal{T}$, whose each node w is labeled by a n tuple $\Delta(w)$ of constant terms (or just a constant term), and moreover, for each inner node w of $\mathcal{T}$, there exists a substitution σ, such that $\sigma\Gamma = \Delta(w)$, and $\sigma\Gamma_i = \Delta(wi)$ for each $i \leq k$.

For a set $\mathcal{G}$ of n-tuples of constant terms, an $\mathcal{H}$-derivation $\mathcal{T}$ will be called a $\mathcal{G}$-$\mathcal{H}$-derivation, if $\Delta(w) \in \mathcal{G}$, or $\Delta(w)$ is a constant, for each leaf w of $\mathcal{T}$. In other words a $\mathcal{G}$-$\mathcal{H}$-derivation for a Horn clause $\mathcal{H}$ given by (1) is an $\mathcal{H}$-derivation from the set $\{Q(t) : t \in \mathcal{G}\} \cup \{Q(c) : c \text{ is a constant }\}$.

2 Regular Thue trees

In this chapter, for a given Thue system $\mathcal{S}$, we introduce notions of a $\mathcal{S}$-regular and of a $\mathcal{S}$-semiregular tree. For the notion of a Thue system see [5]. To fix the terminology, by a *Thue system* $\mathcal{S}$ over an alphabet Σ we understand a finite set of *symmetric* productions, i.e. a finite set of *unordered* pairs $< u, v >$ of words in Σ^*. We moreover require, that $\mathcal{S}$ is *closed*, i.e. if $< u, v > \in \mathcal{S}$, and $< u, w > \in \mathcal{S}$ then $< v, w > \in \mathcal{S}$. We write $\overset{\mathcal{S}}{\Longleftrightarrow}$ to denote that u can be obtained from v by one application of a production in $\mathcal{S}$, and $\overset{*\mathcal{S}}{\Longleftrightarrow}$ to denote that u, v are equivalent in $\mathcal{S}$.

Definition 2.1 *Let $\mathcal{S} = \{< u_i, v_i >: i \in I\}$ be a finite set of productions of a Thue system $\mathcal{S}$ in the alphabet $\{1, 2, ..., p\}$. We put $\mathcal{T}_\mathcal{S} = \{u : u \text{ is a proper prefix of } v, \text{ for some } < v, w > \in \mathcal{S}\}$.*

*For a set $\mathcal{W}$ of words over the alphabet $\{1, 2, ..., p\}$, we put $Thue_\mathcal{S}(\mathcal{W}) = \{u : (\exists w \in \mathcal{W})(w \overset{*\mathcal{S}}{\Longleftrightarrow} u)\}$. Finally, we put $THUE_\mathcal{S}(\mathcal{W}) = pr(Thue_\mathcal{S}(W))$, where, for a set $\mathcal{X}$, $pr(\mathcal{X})$ denotes the set of all prefixes of $\mathcal{X}$. We call $THUE_\mathcal{S}(\mathcal{W})$, a $\mathcal{S}$-*Thue tree *of $\mathcal{W}$.*

Definition 2.2 *(i) Let $\mathcal{T}$ be a p-tree. We call a node $w \in \mathcal{T}$, $\mathcal{S}$-*semiregular*, if $(\mathcal{T})_{wu_i} = (\mathcal{T})_{wv_i}$, for each $i \in I$. We call a node w $\mathcal{S}$-*regular *if it is $\mathcal{S}$-semiregular and if $\mathcal{T}_\mathcal{S} \subseteq (\mathcal{T})_w$.*

*(ii) A p-tree $\mathcal{T}$ is $\mathcal{S}$-*semiregular*, if every node of $\mathcal{T}$ is semiregular. Given a finite set $\mathcal{G}$ of semiregular trees, a $\mathcal{S}$-semiregular p-tree $\mathcal{T}$ is called $\mathcal{G}$-$\mathcal{S}$-*regular *if for each node w of $\mathcal{T}$, either w is $\mathcal{S}$-regular, or $(\mathcal{T})_v$ is in $\mathcal{G}$ for some, non-necessarily proper, prefix v of w.*

Lemma 2.3 *For every Thue system $\mathcal{S}$ and each set $\mathcal{W}$ of words over the alphabet $\{1, 2, ..., p\}$, $THUE(\mathcal{W})$ is a $\mathcal{S}$-semiregular p-tree.* ■

Lemma 2.4 *Every semiregular tree containing a node w contains all elements of the set $\{v : w \overset{*\mathcal{S}}{\Longleftrightarrow} v\}$. So, if there exists a finite semiregular tree containing w, then the set $\{v : w \overset{*\mathcal{S}}{\Longleftrightarrow} v\}$ is finite.* ■

Definition 2.5 *We say that a closed Thue system $\mathcal{S}$ is good, if $(\mathcal{T}_{\mathcal{S}})_v = \emptyset$, for every $< u, v > \in \mathcal{S}$.*

Definition 2.6 *We denote by $\mathcal{H}_{\mathcal{S}}$ the Horn clause*

$$Q(\Gamma_1) \wedge Q(\Gamma_2) \wedge \ldots \wedge Q(\Gamma_p) \rightarrow Q(\Gamma),$$

where Q is an unary relation symbol, and

(i) $\Gamma = g(t_1, t_2, \ldots t_p)$ is a term built over the language containing exactly one p-ary functional symbol g and no constant symbols. For a word w, Γ has g in the position w iff $w \in \mathcal{T}_{\mathcal{S}}$. The variables of Γ in positions u and v are equal iff $< u, v > \in \mathcal{S}$. (ii) For $i \in \{1, 2, ..., p\}$, $\Gamma_i = t_i$.

From now on, we consider terms built from variables, a function symbol g and from a constant c. So, a constant term is uniquely determined by the tree of its function symbols g, so we can, and will identify constant terms with the corresponding trees. The next lemma describes the structure of $\mathcal{G}$-$\mathcal{H}_{\mathcal{S}}$-derivation and establishes a duality between terms and $\mathcal{H}_{\mathcal{S}}$-derivations.

Lemma 2.7 *(i) If the root of an $\mathcal{H}_{\mathcal{S}}$-derivation $\mathcal{D}$ is labeled by a constant term Δ, then for each node w of the derivation, we have $\Delta(w) = (\Delta)_w$.*

(ii) If the root of a $\mathcal{G}$-$\mathcal{H}_{\mathcal{S}}$-derivation $\mathcal{D}$ is labeled by a constant term Δ, then Δ is a semiregular tree. ■

Lemma 2.8 *(i) If the root of a constant term Δ is a regular node, then there exists a substitution σ, such that $\Delta_i = \sigma\Gamma_i$, for $i \in \{1, 2, ..., p\}$. and*

$$\Delta = \sigma\Gamma. \tag{2}$$

(ii) If a constant term Δ is a $\mathcal{G}$-$\mathcal{S}$-regular tree, then Δ is a label of the root of a $\mathcal{G}$-$\mathcal{H}_{\mathcal{S}}$-derivation $\mathcal{D}$. ■

3 Existence of paths in p-clause derivations

Definition 3.1 *We say that a good Thue system $\mathcal{S}$ over $\{1, 2, ..., p\}$ is* very good *if it is closed, and the following conditions are satisfied:*

(i) if $< u, v >$ is a production of $\mathcal{S}$, then both u and v are non-empty and have length at most 2. Moreover, at least one of them has length 2.

(ii) Symbols of the alphabet are divided into two disjoint categories, called dynamic *and* static, *in such a way that:*

(a) if a symbol $i \in \{1, 2, ..., p\}$ occurs in a production $< i, w >$, for $w \in \{1, 2, ..., p\}^*$, then it is dynamic,

(b) if a symbol i occurs in a production of a form $< ji, vk >$, where $v \in \{1, 2, ..., p\}^*$, and $j, k \in \{1, 2, ..., p\}$, and k is dynamic, *then i is* dynamic,

(c) if a symbol i occurs in any production of the form $< ij, v >$ with v, j as above, then i is static.

(iii) The set of words w containing exactly one dynamic symbol as the last symbol of w such that $\{v : w \overset{*\mathcal{S}}{\Longleftrightarrow} v\}$ finite is not recursive.

Lemma 3.2 *Assume that $w \overset{*\mathcal{S}}{\Longleftrightarrow} v$. Then w and v have the same number of dynamic symbols. If w contains exactly one dynamic symbol as its last symbol, then also v contains exactly one dynamic symbol as its last symbol. If w contains no dynamic symbols then the length of w is equal to the length of v.* ■

Lemma 3.3 *There exists a very good Thue process $\mathcal{S}$.*

For the rest of this section we fix $\mathcal{G} = \{g(c, c, \ldots, c)\}$.

Lemma 3.4 *Let t be a word which contains at most one dynamic symbol which, moreover, can appear only as the last symbol of t, and let $\mathcal{T} = THUE(\{t\})$. If $\mathcal{T}$ is finite, then there exists a $\mathcal{G}$-$\mathcal{S}$-regular tree $\mathcal{U}$ containing $\mathcal{T}$ and having the same depth as $\mathcal{T}$.*

As an immediate consequence of Lemma 2.4, Lemma 2.7, Lemma 2.8 and Lemma 3.4 we get the following corollary.

Lemma 3.5 *Let w be a word containing exactly one dynamic symbol as its last symbol. Then the set $\{v : w \overset{*\mathcal{S}}{\Longleftrightarrow} v\}$ is finite iff there exists a $\mathcal{G}$-$\mathcal{H}_{\mathcal{S}}$-derivation containing w as an inner node or as a leaf labeled by the term $g(c, c, \ldots c)$.* ■

As Lemma 3.5 and Lemma 3.3 give the main result of this section.

Theorem 3.6 *There exists an integer p and a p-Horn clause $\mathcal{H}_{\mathcal{S}}$ in a language containing one unary predicate symbol Q, a constant c and a p-ary function symbol g, such that, for $\mathcal{G} = \{c, g(c, c, ..., c)\}$, the problem whether, for a given w, there exists a $\mathcal{G}$-$\mathcal{H}_{\mathcal{S}}$-derivation containing w as an inner node or as a leaf labeled with the term $g(c, c, \ldots c)$ is undecidable.*
■

To derive the final result from Theorem 3.6 several technical lemmas are needed. First, to replace an unknown p by 2, then, to hide a large uncontrollable term that appear in the root of the derivations considered and finally to force the derivation tree to contain the given branch. More details can be found in [4]

References

[1] P.-J. Müller-Mayer. Order-Sorted Higher-Order ACID-Disjunification in the Combination of Shallow Theories with a Sequential Signature. *J. Association for Unification Research*, 1:1–392, 1999.

[2] Y. Gurevich, On the Classical Decision Problem, *Bulletin of European Association for Theoretical Computer Science* 42 (1990), pp. 140-150.

[3] A. Leitsch, G.Gottlob, Deciding Horn clause implication problems by ordered semantic resolution, *Computational Intelligence II*, F. Gardin and G. Mauri, ed., 1990, pp.19-26.

[4] J. Marcinkowski, L. Pacholski, Undecidability of the Horn-Clause Implication Problem, *Proc. IEEE Annual Symposium on Foundations of Computer Science*, Los Alamitos 1992, pp. 354-362.

[5] E. Post, Recursive unsolvability of a problem of Thue, *Journal of Symbolic Logic* 12 (1947), pp. 1-11.

[6] M. Schmidt-Schauß, Implication of Clauses is Undecidable, *Theoretical Computer Science* 59 (1988), pp. 287-296.

Higher-Order E-Unification for Arbitrary Theories

Zhenyu Qian and Kang Wang[1]

We present an algorithm consisting of three transformation rules for pre-unification of simply typed λ-terms w.r.t. α, β and η conversions and an arbitrary first-order equational theory E. The algorithm is parameterized by E-unification algorithms that admit free function symbols. It is proved that the algorithm is complete if the given E-unification algorithm is complete. The result is relevant to implementations of higher-order logic programming languages and higher-order proof systems.

This work has appeared in: Proc. Joint Int. Conf. and Symp. on Logic Programming, Nov. 1992, Washington, D.C., MIT Press.

[1]FB Mathematik/Informatik, Universität Bremen, 2800 Bremen 33, Germany, email: `qian@informatik.uni-bremen.de`,`wang@pc-labor.uni-bremen.de`.

Unification in a Combination of Equational Theories with Shared Constants and its Application to Primal Algebras

Christophe Ringeissen[1]

1 Introduction

The general idea of the unification in a combination of theories consists in breaking an equational problem into sub-problems that are pure in the sense that they can be solved in one component of the combination. This problem was initiated in [5, 9, 11] where syntactic conditions on the axioms of the disjoint theories to be combined were assumed. Then, this problem has been solved by M. Schmidt-Schauß [8] and A. Boudet [2] in the general case of arbitrary disjoint equational theories: a unification algorithm with free constants and a free constant elimination algorithm should be provided for each equational theory. Recently, F. Baader and K. Schulz [1] have shown an improved method for solving the combined unification problem: only one algorithm for solving unification with linear free constant restriction, which is a slight generalization of unification with free constants, is necessary for each equational theory.

The problem considered in this paper is the unification in the union $E_1 \cup E_2$ of equational theories E_1 and E_2 where E_1, E_2 are built over non disjoint signatures: they shared only constants. In this context, we are faced to several problems:

- Solving pure equations in the related component of the combination is sound with disjoint equational theories, but what about non disjoint ones? We show that this essential feature is still available for equational theories with only shared constants.
- The theory of the top symbol may change during an $E_1 \cup E_2$-equational proof, thanks to collapse axioms, for example $f(x, x) = x$, but also now thanks to axioms $s =_{E_1} c$ and $c =_{E_2} t$, where c is a shared constant. The change of theory is achieved through a variable or a shared constant. Therefore, treating variables as free constants is no more sufficient, we must also take into account shared constants. By adding a transition rule for this last case, we derive a complete unification algorithm from the one designed in [1].

[1]CRIN-CNRS & INRIA-Lorraine, BP 239 54506 Vandoeuvre-lès Nancy Cedex France, Email: Christophe.Ringeissen@loria.fr.

The unification algorithm is illustrated on a particularly interesting class of equational theories generated by some finite algebras called Primal Algebras for which unification is of greatest interest since it is unitary [3, 6]. We show how a convenient strategy improves some a priori nondeterministic steps of the algorithm.

Unification in Primal Algebras has attracted considerable interest for its applications to hardware descriptions [4, 10].

A full paper is available [7].

2 Combined Algorithm with Shared Constants

Let E_1, E_2 be equational theories over signatures F_1, F_2 and let $E = E_1 \cup E_2$ denotes their union over $F = F_1 \cup F_2$. The set $SC = (F_1)_0 \cap (F_2)_0$ of shared constants may be non empty.

The first step of the combination algorithm transforms an equational problem Γ into a pair (Γ_1, Γ_2) of pure (respectively in E_1, E_2) equational problems such that Γ and $\Gamma_1 \wedge \Gamma_2$ are equivalent. Purification is achieved by applying repeatedly the following rules

Variable Abstraction

$$\frac{\Gamma \wedge s =^? t}{\Gamma \wedge s[\omega \hookleftarrow x] =^? t \wedge x =^? s_{|\omega}} \quad \text{if } s_{|\omega} \text{ is an alien subterm of } s,\ x \text{ is fresh.}$$

Impure Equation

$$\frac{\Gamma \wedge s =^? t}{\Gamma \wedge x =^? s \wedge x =^? t} \quad \text{if } s \in T(F_1, X)\backslash X,\ t \in T(F_2, X)\backslash X,\ x \text{ is fresh.}$$

In the context of disjoint equational theories, the main difficulty of the algorithm is to combine unifiers computed for each equational theory E_i. The reason is that a same variable may be instantiated in both theories. In order to introduce and justify the new transformation rules, it is shown first that a pair (Γ_1, Γ_2) of pure equational problems may be solved thanks to the computation of unifiers in the combination of *disjoint* equational theories. When some constants are shared, we need to consider each possible instantiation of variables with shared constants (in addition to identify variables in all possible ways) before choosing the theory, otherwise we forget some solutions as shown in the following example.

Example: *Let us consider $E_1 = \{x \star \top = \top\}$ and $E_2 = \{x + \top = \top\}$ two consistent equational theories sharing $\top$. The equation $(x \star \top =^? x + \top)$ is valid in E and consequently unifiable. A unification algorithm in the union of disjoint equational theories works as follows with this equation. A new variable is added in order to obtain two pure equations $(x \star \top =^? y)$ and $(x + \top =^? y)$ where y must be treated as a free constant in one theory. There is no solution to these equations. Otherwise we get $y =_{E_i} \top$, that is E_i is inconsistent which yields a contradiction. Therefore this algorithm yields no solution and does not compute a complete set of E-unifiers. Instead, the proposed method forces to consider both equations $(x \star \top =^? \top)$ and $(x + \top =^? \top)$.*

In the following, we establish how to reuse the combination techniques developed in [1].

Definition: *Let* Γ_i *be a i-pure equational problem,* ind *a mapping from variables* $V \supseteq V(\Gamma_i)$ *to the set of theory indices* $\{1, 2\}$ *and* $<$ *a linear ordering on* V. $CSU_{E_i}^{(ind,<)}(\Gamma_i)$ *denotes a complete set of* E_i*-unifiers* σ_i *of* Γ_i *w.r.t.* linear constant restriction $<$ *such that*

- $x\sigma_i = x$ *if* $ind(x) \neq i$.
- $y \notin x\sigma_i$ *if* $x < y$ *and* $ind(y) \neq i$.

Given a substitution $\sigma = \{x_1 \mapsto t_1, \ldots x_n \mapsto t_n\}$, $\hat{\sigma}$ denotes the equational problem $x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$ in *tree solved form.*

Theorem: *A complete set of* $E_1 \cup E_2$*-unifiers of* $\Gamma_1 \wedge \Gamma_2$ *are given by all the dag solved forms* $\hat{\phi} \wedge \hat{\rho} \wedge \hat{\sigma_1} \wedge \hat{\sigma_2}$ *such that*

- ϕ *is an identification on variables:*

$$\mathcal{D}om(\phi) \cup \mathcal{R}an(\phi) \subseteq V(\Gamma_1 \wedge \Gamma_2).$$

- ρ *is a mapping onto shared constants:*

$$\mathcal{D}om(\rho) \subseteq V(\Gamma_1 \wedge \Gamma_2) \text{ and } \mathcal{R}an(\rho) \subseteq SC,$$

  where SC *denotes the set of shared constants.*

- $\sigma_1 \in CSU_{E_1}^{(ind,<)}(\Gamma_1\phi\rho)$.
- $\sigma_2 \in CSU_{E_2}^{(ind,<)}(\Gamma_2\phi\rho)$.

When a symbol of arity strictly greater than 1 is assumed shared, many problems arise. For example, there are infinitely many proper shared terms that make a connection between both equational theories: skolemizing shared variables in one theory does not capture this kind of potential solution. For this reason, the straightforward generalization of this work to other shared symbols appears unfortunately to be jeopardized.

References

[1] F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *Proceedings 11th International Conference on Automated Deduction, Albany (NY, USA)*, 1992.

[2] A. Boudet. *Unification dans les mélanges de théories équationelles. Application aux axiomes d'associativité, commutativité, identité et idempotence, aux anneaux Booléens, et aux groupes Abéliens.* Thèse de Doctorat d'Université, Université de Paris-Sud, Orsay (France), February 1990.

[3] W. Büttner. Unification in finite algebras is unitary. In *Proceedings 9th International Conference on Automated Deduction, Argonne (Illinois, USA)*, pages 368–205. Springer-Verlag, 1988.

[4] W. Büttner, K. Estenfeld, R. Schmid, H.-A. Schneider, and E. Tiden. Symbolic constraint handling through unification in finite algebras. *Applicable Algebra in Engineering, Communication and Computation*, 1(2):97–118, 1990.

[5] A. Herold. Combination of unification algorithms. In J. Siekmann, editor, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, volume 230 of *Lecture Notes in Computer Science*, pages 450–469. Springer-Verlag, 1986.

[6] T. Nipkow. Unification in primal algebras, their powers and their varieties. *Journal of the Association for Computing Machinery*, 37(1):742–776, October 1990.

[7] C. Ringeissen. Unification in a combination of equational theories with shared constants and its application to primal algebras. In *Proceedings of LPAR'92*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 261–272. Springer-Verlag, July 1992.

[8] M. Schmidt-Schauß. Combination of unification algorithms. *Journal of Symbolic Computation*, 8(1 & 2):51–100, 1989. Special issue on unification. Part two.

[9] E. Tidén. Unification in combinations of collapse-free theories with disjoint sets of functions symbols. In J. Siekmann, editor, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, volume 230 of *Lecture Notes in Computer Science*, pages 431–449. Springer-Verlag, 1986.

[10] E. Tidén. Symbolic verification of switch-level circuits using a prolog enhanced with unification in finite algebras. In G. Milne, editor, *The fusion of hardware design and verification*, pages 465–485. IFIP WG 10.2, North-Holland, 1988.

[11] K. Yelick. Unification in combinations of collapse-free regular theories. *Journal of Symbolic Computation*, 3(1 & 2):153–182, April 1987.

Retrieving Library Functions by Unifying Types Modulo Linear Isomorphism

Mikael Rittri[1]

This is a updated summary of a report [12] that was distributed at UNIF'92. The updates are that CCC-unifiability has been shown to be undecidable [9], and that two related software retrieval systems have appeared [6, 13].

1 Introduction

Types can be used as search keys in software libraries, especially in functional languages. This idea was independently proposed by me [10] and by Runciman and Toyn [14], but in different ways. I suggested retrieving a library function if its type was CCC-isomorphic to the query, while Runciman and Toyn suggested retrieving it if its type was unifiable with the query (they also allowed the library function to have extra arguments).

Both CCC-isomorphism and unification are useful, so they should be combined. I have developed an algorithm for matching modulo CCC-isomorphism [11], which makes it possible to retrieve a function if its type has an instance that is CCC-isomorphic to the query. But in some cases, unification is necessary for retrieval, and Paliath Narendran has shown that unifiability modulo CCC-isomorphism is not decidable [9].

2 Linear isomorphism suffices for library search

What is CCC-isomorphism, then? It is the isomorphism between types that holds in any Cartesian closed category (CCC). An equivalent definition is that two types A and B are CCC-isomorphic iff there are λ-expressions $f : A \to B$ and $g : B \to A$ such that the equalities $g \circ f = id_A$ and $f \circ g = id_B$ hold in simply typed $\lambda\beta\eta$-calculus with surjective pairing [4]. This isomorphism is more familiar as equality in the algebra $(\mathbf{N}, 1, \times, \uparrow)$ of natural numbers with 1, multiplication and exponentiation, which is a CCC. Table 0.1 gives an equational axiomatization [1, 7, 15].

Why should CCC-isomorphism be used in software retrieval? Because when two types A and B are CCC-isomorphic, it is easy to convert back and forth using the bijections

[1]Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden, Email: rittri@cs.chalmers.se.

$$
\begin{array}{rrcll}
 & A \times B & \cong & B \times A & \text{(Com-2)} \\
 & (A \times B) \times C & \cong & A \times (B \times C) & \text{(Ass-2)} \\
 & \mathbf{1} \times A & \cong & A & \text{(Ass-0)} \\
 & (A \times B) \to C & \cong & A \to (B \to C) & \text{(Cur-2)} \\
 & \mathbf{1} \to C & \cong & C & \text{(Cur-0)} \\
\text{non-linear} \left\{ \begin{array}{r} \\ \\ \end{array} \right. \!\!\!\!\!\!\!\!\! & \begin{array}{r} A \to (B \times C) \\ A \to \mathbf{1} \end{array} & \begin{array}{c} \cong \\ \cong \end{array} & \begin{array}{l} (A \to B) \times (A \to C) \\ \mathbf{1} \end{array} & \begin{array}{l} \text{(Dist-2)} \\ \text{(Dist-0)} \end{array}
\end{array}
$$

Table 0.1: Equational axioms for CCC-isomorphism.

f and g that must exist. This means that a programmer often must make an arbitrary choice between several isomorphic types when he implements a library function, and a library user cannot guess which one was chosen. Instead, the user's query type should retrieve any function of an isomorphic type.

But some useful library functions can be retrieved only if we allow instantiation of library types and query types before we check CCC-isomorphism. This is unifiability modulo CCC-isomorphism, which unfortunately is not decidable [9], although matchability is [9, 11]. If we want to unify, we are thus forced to change the equational theory. I have chosen to remove the (Dist-2) and (Dist-0) axioms of Table 0.1, not only to make unification possible, but because I think they are not really needed. Their associated bijections are not linear in the sense of linear logic, as are those of the other axioms. To be linear in this sense means that each variable should be bound once and used once [3, section 7], and the bijections for (Dist-2) must use some variables twice, while the bijections for (Dist-0) must bind variables they do not use. A non-linear bijection can change the amount of shared computation, and since most library functions have a natural amount of sharing that a user can guess, only the linear bijections are needed for library search. For instance, if a function returns a B-value and a C-value in a single computation for every A-value, its most natural type is $A \to (B \times C)$, but if it computes only B-values for some A-values and only C-values for others, it is more natural to split it into a function-pair of the distributed type $(A \to B) \times (A \to C)$. Therefore, the choice between isomorphic types is arbitrary only when the isomorphism is linear. When it is not linear, the user has an excellent chance of guessing the choice of the library programmer.

Sergei Soloviev has shown that the five linear axioms in Table 0.1 are equationally sound and complete for linear CCC-isomorphism [16]. In the terminology of category theory, the five axioms describe the types that are isomorphic in all symmetric monoidal closed categories, sometimes called SMC categories or just closed categories.

A unification algorithm modulo these axioms has been given by Narendran, Pfenning and Statman [9].

3 Experiments with equational unification

I have implemented the unification algorithm of Narendran *et al.* [9] on top of a Standard ML program for associative-commutative unification [5]. But unrestricted unifiability would be too liberal for function retrieval. When query variables express polymorphism, they should not be instantiated. For instance, a library user who seeks a function that reverses lists will know that its type $\forall\alpha.\,[\alpha] \to [\alpha]$ is polymorphic, so it makes no sense to retrieve functions of type $[\mathit{Float}] \to [\mathit{Float}]$ for such a query. Therefore, bound variables in a query are not instantiated. But in other cases, a user needs query variables to stand for unknown types; these variables are free and can be instantiated.

I have added a further restriction that variables in library types must not be instantiated to $\mathbf{1}$, as this seems to retrieve only rubbish.

The retrieval system is still often too liberal; for instance, if the user seeks a function of type Q, and allows library functions to have extra arguments by submitting the query $\varepsilon \to Q$, then any function of a type $\forall\alpha.\,A \to \alpha$ will be retrieved via the substitution $\{\alpha := Q,\ \varepsilon := A[Q/\alpha]\}$. Although the query and the answer are unifiable in this case, they need not be similar in any other way. To handle this problem, I rank the retrieved functions by the sizes of the necessary substitutions, with the effect that library functions whose types need only be instantiated a little (or not at all) are placed first. The motivation is that the more general type a function has, the less it can do, since it cannot examine the internal structure of its polymorphic arguments; therefore, the more instantiation needed to fit a library type to a query, the less likely it is that the associated function is useful. My definition of substitution size is empirical but works surprisingly well.

Example: Let us look for a function to check membership in a list. To try Runciman and Toyn's strategy [14] to allow extra arguments to library functions, we can query with $\forall\alpha.\,\varepsilon \times \alpha \times [\alpha] \to \mathit{Bool}$. Since ε is a free variable, unlike α, it can be instantiated to the unknown type of the extra argument(s). From the Lazy ML library of 294 identifiers, this query retrieves

$$
\begin{array}{lcll}
\mathit{mem} & : & \forall\beta.\,\beta \to [\beta] \to \mathit{Bool} & (0,\,2) \\
\mathit{member} & : & \forall\beta\gamma.\,(\beta \to \gamma \to \mathit{Bool}) \to \beta \to [\gamma] \to \mathit{Bool} & (1,\,7) \\
(=) & : & \forall\beta.\,\beta \to \beta \to \mathit{Bool} & (5,\,5) \\
\multicolumn{3}{l}{\ldots\text{thirty-four functions omitted}\ldots} & \vdots \\
\mathit{while} & : & \forall\beta.\,(\beta \to \mathit{Bool}) \to (\beta \to \beta) \to \beta \to \beta & (9,\,47)
\end{array}
$$

Thirty-eight functions are retrieved, but the two relevant ones have the best ranks. The *mem* function is retrieved via the substitution $\{\beta := \alpha,\ \varepsilon := \mathbf{1}\}$, and *member*, which takes an equivalence test as argument, is retrieved via $\{\beta := \alpha,\ \gamma := \alpha,\ \varepsilon := (\alpha \to \alpha \to \mathit{Bool})\}$. Note that the type of *member* is more general than the query.

An older version of the retrieval system has been used for more than a year by the Lazy ML programmers at Chalmers. Most of them find it a useful tool, and Thomas Hallgren and Staffan Truvé have made a window-based interface that allows users to click at a retrieved function to get a look at its source code. The retrieval time varies between a few seconds and a minute on a SUN-4. It should be said, though, that a plain isomorphism test is usually enough for retrieval; matching and unification are seldom needed.

4 Related work

Brian Matthews [6] has extended my retrieval method to the Haskell language, which uses a system of type classes to deal with overloaded operators. The unification must then be order-sorted.

Roberto Di Cosmo [2] has shown that the seven axioms in Table 0.1, although complete for simply typed λ-calculus, are not complete for Hindley/Milner types. Some extra axioms like $\forall\alpha.\, A \times B \cong \forall\alpha\beta.\, A \times (B[\beta/\alpha])$ would make them complete, but the extra ones can be used directly when the compiler derives types, in which case a retrieval system does not need them.

I know of two attempts to use formal specifications as queries; both use type queries as an initial filter. Richard Morgan [8] has built a theorem prover on top of Boyer/Moore's, while Rollins and Wing [13] use λ-Prolog.

Acknowledgements

I thank Paliath Narendran, Frank Pfenning and Richard Statman for inventing the unification algorithm, Erik Lindström *et al.* for implementing associative-commutative unification, and Sergei Soloviev for proving equational completeness. This work would not be possible without assistance from them. My retrieval system is much nicer to use since Thomas Hallgren and Staffan Truvé made a window-based user interface. I am also grateful to Roberto Di Cosmo, Peter Dybjer, Yves Lafont, Pierre Lescanne, Giuseppe Longo, Brian Matthews, G. E. Mints, François Rouaix, and Antti Valmari for advice and information. And my local user group, especially Annika Aasa, has provided valuable feedback.

References

[1] K. B. Bruce, R. Di Cosmo, and G. Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2), 1992.

[2] R. Di Cosmo. Type isomorphisms in a type-assignment framework: From library searches using types to the completion of the ML type checker. In *19th Ann. ACM Symp. on Principles of Programming Languages*, pages 200–210, ACM Press, 1992.

[3] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, with corrigenda, 62:327–328, 1988.

[4] J. Lambek. From lambda calculus to Cartesian closed categories. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 376–402. Academic Press, 1980.

[5] E. Lindström. Unification in μUTRL. Tech. report UMINF-92.03, Dept. Comput. Sci., U. of Umeå, Sweden (term@cs.umu.se), 1992.

[6] B. Matthews. Reusing functional code using type classes for library search. Presented at the *ERCIM Workshop on Software Reuse*, Heraklion, Crete, Oct. 1992. Author's address: Dept. Comput. Sci., The University, Glasgow, U.K. (brian@dcs.glasgow.ac.uk).

[7] L. Meertens and A. Siebes. Universal type isomorphisms in Cartesian closed categories — preliminary version. Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands ({lambert,arno}@cwi.nl), 1990.

[8] R. Morgan. *Component Library Retrieval using Property Models.* PhD thesis, SEAS, U. of Durham, England, (rick@easby.dur.ac.uk), 1991.

[9] P. Narendran, F. Pfenning, and R. Statman. On the unification problem for Cartesian closed categories. Addresses: P. Narendran, State U. of NY at Albany, USA (dran@cs.albany.edu). F. Pfenning and R. Statman, Carnegie Mellon U., Pittsburgh, USA ({fp,statman}@cs.cmu.edu), 1992.

[10] M. Rittri. Using types as search keys in function libraries. *J. Functional Programming*, 1(1):71–89, 1991. Preliminary version in *Functional Program. Lang. and Comput. Arch.*, pages 174–183, ACM Press 1989.

[11] M. Rittri. Retrieving library identifiers via equational matching of types. In M. E. Stickel, editor, *10th Int. Conf. on Automated Deduction*, Kaiserslautern, Germany. Vol. 449 of *Lecture Notes in Artificial Intelligence*, pages 603–617, Springer-Verlag, 1990.

[12] M. Rittri. Retrieving library functions by unifying types modulo linear isomorphism. PMG Report 66, Dept. Comput. Sci., Chalmers U. of Tech., Göteborg, Sweden (rittri@cs.chalmers.se), May 1992.

[13] E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Proc. Eighth Int. Conf. on Logic Programming,* pages 173–187, MIT Press, 1991.

[14] C. Runciman and I. Toyn. Retrieving reusable software components by polymorphic type. *J. Functional Programming*, 1(2):191–211, 1991. Preliminary version in *Functional Program. Lang. and Comput. Arch.*, pages 166–173, ACM Press 1989.

[15] S. V. Soloviev. The category of finite sets and Cartesian closed categories. *J. Soviet Math.*, 22(3):1387–1400, 1983. First published in Russian in *Zap. Nauch. Semin. Leningr. Otd. Mat. Inst.*, 105:174–194, 1981.

[16] S. V. Soloviev. The ordinary identities form a complete axiom system for isomorphism of types in closed categories. The Institute for Informatics and Automation of the Academy of Sciences, 199178, St. Petersburg, Russia, fax: +7(812)217 5105, (e-mail via S. Baranoff, sergei@iias.spb.su), 1991.

Feature Algebras as Coalgebras: A Category Perspective on Unification

William Rounds[1]

A *feature algebra* is a set of objects, together with a collection of partial unary functions called *features*. One use of feature algebras is to regard the elements as real-world objects, and the features as functions, which when evaluated, return attributes of those objects. Thus the notion of *extensionality* is important: an algebra is extensional if (intuitively) whenever two objects have the same features, then they are the same. We characterize the notion in terms of subsumption relationships induced by algebra homomorphisms. We prove representation theorems for extensional algebras, using two technical definitions of extensionality. One such class is the class of F*-extensional* algebras in which isomorphic elements are identified, and the other is the class of *strongly extensional* algebras, in which "bisimilar" elements are identified. Both of these definitions are slightly stronger than the "true" intuitive defintions, modulo a minor technicality for F-extensional algebras.

Now the extensional algebras turn out, by these theorems, to be "structurally determinate," to borrow a phrase from Barwise. That is, the elements of these algebras are data structures akin to graphs or terms, in which, for example, unification can be successfully defined. The reason for this is that the representation theorems can be stated in category-theoretic terms, which gives us the focus for this abstract, and also relates the topic to that of general unification theory. It turns out that feature algebras are the coalgebras of certain obvious functors on the class of sets. The extensional algebras of both types form a natural class of algebras in which unification, as a partial binary operation on pairs of elements, is a join operation with respect to the subsumption ordering. This follows from the explicit representation of the extensional algebras, and then from the characterization of two particular algebras as final coalgebras of the appropriate functors.

The analogy to ordinary unification of terms helps us to understand these results. Term unification is with respect to the term algebra, which is initial in the category of algebras of a certain standard functor essentially given by the signature. Feature unification, on the other hand, works on the final coalgebras of roughly the same functor.

[1]Artificial Intelligence Laboratory, EECS Department, University of Michigan, Ann Arbor, Michigan 48109, email: `rounds@caen.engin.umich.edu`.

Constraint Programming Based on Relative Simplification

Gert Smolka[1]

Constraint logic programming, concurrent logic programming and negation as failure are three well-established subareas of logic programming that developed more or less independently. More recently it turned out that at least constraint and concurrent logic programming can be profitably merged into one more general paradigm, now becomming known under the name concurrent constraint programming. We argue that negation also fits well into this new paradigm, and that in fact the operational semantics for deep guards in concurrent logic programming has much in common with constructive negation, a powerful operational semantics for negation in logic programming.

We present a rewrite calculus that gives a unified and abstract operational account of concurrent constraint languages with disjunction and negation. The calculus is parameterized with respect to a constraint theory and a program extending the constraint system with new predicates. Computation amounts to rewriting expressions according to the rules of the calculus. The major inovation of the calculus is the principle of relative simplification, which simplifies a constraint at a position P in an expression E modulo its context, which is a constraint uniquely determined by P and E. The principle of relative simplification at the same time provides for constraint simplification, incremental entailment checking, deep guards, and constructive negation.

References

[1] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka, "A Feature-based Constraint System for Logic Programming with Entailment," in *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo (1992) 1012–1021.

[2] Gert Smolka and Ralf Treinen, "Records for Logic Programming," *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, K. Apt (Ed.) (1992) 240-254. Full version has appeared as Research Report RR-92-23, DFKI, Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany.

[1]DFKI, Stuhlsatzenhausweg 3, D-W-6600 Saarbr"ucken, Germany, email: `smolka@dfki.uni-sb.de`.

[3] Gert Smolka, Martin Henz, and Jörg Würtz, "Object-Oriented Concurrent Constraint Programming in Oz," Research Report RR-93-16, DFKI, Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany.

Relative Simplification for and Independence of CFT

Gert Smolka and Ralf Treinen[1]

1 Introduction

Records are an important data structure in programming languages. They appeared first with imperative languages such as ALGOL 68 and Pascal, but are now also present in modern functional languages such as SML. A major reason for providing records is the fact that they serve as the canonical data structure for expressing object-oriented programming techniques.

In this paper we will show that records can be incorporated into logic programming in a straightforward and natural manner. We will model records with a constraint system CFT, which can serve as the basis of future constraint (logic) programming languages. Since CFT is a conservative extension of Prolog II's rational tree system [9, 10], the familiar term notation can still be used. We haven chosen to admit infinite trees so that cyclic data structures can be represented directly. However, a set-up admitting only finite trees as in the original Horn clause model is also possible.

2 Records are Feature Trees

We model records as feature trees [6, 7]. A feature tree is a tree whose edges are labeled with symbols called features, and whose nodes are labeled with symbols called sorts. The features labeling the edges correspond to the field names of records. As one would expect, the labeling with features must be deterministic, that is, every direct subtree of a feature tree is uniquely identified by the feature of the edge leading to it. Feature trees without subtrees model atomic values (e.g., numbers). Feature trees may be finite or infinite. Infinite feature trees provide for the convenient representation of cyclic data structures.

A ground term, say $\mathsf{f}(\mathsf{g}(\mathsf{a},\mathsf{b}),\mathsf{h}(\mathsf{c}))$, can be seen as a feature tree whose nodes are labeled with function symbols and whose arcs are labeled with numbers. Thus the trees corresponding to first-order terms are in fact feature trees observing certain restrictions (e.g., the features departing from a node must be consecutive positive integers).

[1]Deutsches Forschungszentrum für künstliche Intelligenz (DFKI), Stuhlsatzenhausweg 3, D6600 Saarbrücken, Germany. Email: {`smolka,treinen`}`@dfki.uni-sb.de`.

3 Record Descriptions

In CFT, records (i.e., feature trees) are described by first-order formulae. To this purpose, we set up a first-order structure $\mathcal{T}$ (CFT's standard model) whose universe is the set of all feature trees (over given alphabets of features and sorts), and whose descriptive primitives are defined as follows:

- Every sort symbol A is taken as a unary predicate, where a *sort constraint* $x\colon A$ holds if and only if the root of the tree x is labeled with A.
- Every feature symbol f is taken as a binary predicate, where a *feature constraint* $x[f]y$ holds if and only if the tree x has the direct subtree y at feature f.
- Every finite set F of features is taken as a unary predicate, where an *arity constraint* xF holds if and only if the tree x has direct subtrees exactly at the features appearing in F.

The descriptions or constraints of CFT are now exactly the first-order formulae obtained from the primitive forms specified above, where we include equations "$x = y$" between variables.

A feature constraint $x[f]y$ corresponds to field selection for records. A more familiar notation for $x[f]y$ might be $y = x.f$. Note that the field selection function "$x.f$" is partial since not every record has a field f.

Next we note that the familiar term notation can still be used in CFT if a little syntactic sugar is provided. For instance, the equational constraint

$$\mathsf{X} \;=\; \mathsf{point}(\mathsf{Y}, \mathsf{Z})$$

employing the binary constructor point translates into the conjunction

$$\mathsf{X}\colon \mathsf{point} \wedge \mathsf{X}\{1, 2\} \wedge \mathsf{X}[1]\mathsf{Y} \wedge \mathsf{X}[2]\mathsf{Z}.$$

Note that constructors and features are dual in the sense that features are argument selectors for constructors.

CFT can also express constructors that identify their arguments by keywords rather than by position. For instance, the equation

$$\mathsf{P} \;=\; \mathsf{point}(\mathsf{xval}\colon \mathsf{X},\ \mathsf{yval}\colon \mathsf{Y},\ \mathsf{color}\colon \mathsf{Z})$$

can be taken as an abbreviation for

$$\mathsf{P}\colon \mathsf{point} \wedge \mathsf{P}\{\mathsf{xval}, \mathsf{yval}, \mathsf{color}\} \wedge \mathsf{P}[\mathsf{xval}]\mathsf{X} \wedge \mathsf{P}[\mathsf{yval}]\mathsf{Y} \wedge \mathsf{P}[\mathsf{color}]\mathsf{Z}.$$

Compared to the standard tree constraint systems, the major expressive flexibility provided by CFT is the possibility to access a feature without saying anything about the existence of other features. The constraint

$$\mathsf{X}[\mathsf{color}]\mathsf{Y}$$

says that X must have a color field whose value is Y, but nothing else. Hence we can express properties of the color of X without knowing whether X is a circle, triangle, car or something else. Using constructor constraints, we would have to write a disjunction

$$\mathsf{X} = \mathsf{circle}(\ldots, \mathsf{Y}, \ldots) \ \vee \ \mathsf{X} = \mathsf{triangle}(\ldots, \mathsf{Y}, \ldots) \ \vee \ \ldots$$

which means that we have to know statically which alternatives are possible dynamically. Moreover, disjunctions are expensive computationally. In contrast, feature constraints like $\mathsf{X}[\mathsf{color}]\mathsf{Y}$ allow for efficient constraint simplification, as we will see in this paper.

Descriptions leaving the arity of a record open are also essential for knowledge representation, where a description like

$$\mathsf{X}\colon \mathsf{person}[\mathsf{father}\colon \mathsf{Y},\ \mathsf{employer}\colon \mathsf{Y}]$$

should not disallow other features. In CFT this description can be expressed by simply *not* imposing an arity constraint:

$$\mathsf{X}\colon \mathsf{person} \wedge \mathsf{X}[\mathsf{father}]\mathsf{Y} \wedge \mathsf{X}[\mathsf{employer}]\mathsf{Y}.$$

4 Constraint Simplification

The major technical contribution of this paper is the presentation and verification of a constraint simplification method for CFT. This method provides for incremental entailment / disentailment checking as it is needed for more advanced constraint programming frameworks [14, 15].

To state our technical results precisely, let a simple constraint be a formula in the fragment

$$[x\colon A,\ x[f]y,\ xF,\ x = y,\ \bot,\ \top]_{\wedge,\exists}$$

obtained by closing the atomic formulae under conjunction and existential quantification. Let γ and ϕ be simple constraints. We give a method that decides simultaneously entailment $\gamma \models_{\mathrm{CFT}} \phi$ and disentailment $\gamma \models_{\mathrm{CFT}} \neg\phi$. This method can be implemented by an incremental algorithm having quasi-linear complexity, provided the features possibly occurring in γ and ϕ are restricted a priori to some finite set. We also prove that CFT satisfies the independence property, that is,

$$\gamma \models_{\mathrm{CFT}} \phi_1 \vee \ldots \vee \phi_n \quad \Longleftrightarrow \quad \exists i\colon\ \gamma \models_{\mathrm{CFT}} \phi_i.$$

Hence, our decision method can decide the satisfiability of conjunctions of positive and negative simple constraints since

$$\gamma \wedge \neg\phi_1 \wedge \ldots \wedge \neg\phi_n \models_{\mathrm{CFT}} \bot$$

is equivalent to

$$\gamma \models_{\mathrm{CFT}} \phi_1 \vee \ldots \vee \phi_n.$$

All results are obtained under the assumption that the alphabets of sorts and features are infinite.

5 Related Work

CFT can be viewed as the minimal combination of Colmerauer's rational tree system [9, 10] with the feature constraint system FT [6]. In fact, CFT is obtained from FT by simply adding arity constraints as new descriptive primitive. However, the addition of arity constraints requires a nontrivial extension of FT's relative simplification method [6], which can be seen from the fact that the entailment

$$x = f(x, y) \land y = f(y, y) \models_{\mathrm{CFT}} x = y$$

holds in CFT. (It of course also holds in Colmerauer's rational tree system.)

Our operational investigations are based on congruences and normalizers of constraints, two new notions providing for an elegant presentation of our results. Huet [11] uses the related notion of "équivalence simplifiable" in his study of rational tree unification. We improve on Colmerauer's [10] results for rational trees since our constraints are closed under existential quantification. For instance, our algorithm is complete for quantified negative constraints such as $\neg\exists y \exists z(z = f(y, z))$.

Feature descriptions have a long and winded history. One root are the unification grammar formalisms FUG [13] and LFG [12] developed for applications in computational linguistics (see [8] for a more recent paper in this area). Another, independent root is Aït-Kaci's ψ-term calculus [1, 2], which is the basis of several constraint programming languages [3, 4, 5]. Smolka [16] gives a unified logical view of most earlier feature formalisms and studies an expressive feature constraint logic.

Feature trees appeared only recently with the work on FT [7, 6]. To our knowledge the notion of an arity constraint is new. Carpenter's [8] extensional types are somewhat related in that they fix an arity for all elements of a type.

A short version of the paper containing the technical results has been accepted for publication [17]. The full paper including the proofs and an abstract machine implementing our relative simplification algorithm is published as [18].

References

[1] H. Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures.* PhD thesis, University of Pennsylvenia, Philadelphia, PA, 1984.

[2] H. Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Comput. Sci.*, 45:293–351, 1986.

[3] H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[4] H. Aït-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[5] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming*

Language Implementation and Logic Programming, Springer LNCS vol. 528, pages 255–274. Springer-Verlag, 1991.

[6] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. In *Fifth Generation Computer Systems 1992*, pages 1012–1021, Tokyo, Japan, June 1992. Institute for New Generation Computer Technology.

[7] R. Backofen and G. Smolka. A complete and recursive feature theory. Research Report RR-92-30, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany, July 1992.

[8] B. Carpenter. Typed feature structures: A generalization of first-order terms. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 187–201, San Diego, USA, 1991. The MIT Press.

[9] A. Colmerauer. Prolog and infinite trees. In K. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[10] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[11] G. Huet. Résolution d'equations dans des langages d'ordre $1, 2, \cdots, \omega$. Thèse de Doctorat d'Etat, l'Université Paris VII, Sept. 1976.

[12] R. M. Kaplan and J. Bresnan. Lexical-Functional Grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–381. MIT Press, Cambridge, MA, 1982.

[13] M. Kay. Functional grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, Berkeley, CA, 1979. Berkeley Linguistics Society.

[14] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Cambridge, MA, 1987. MIT Press.

[15] V. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, January 1990.

[16] G. Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[17] G. Smolka and R. Treinen. Records for logic programming. Research Report RR-92-23, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-W-6600 Saarbrücken, Germany, 1992.

[18] G. Smolka and R. Treinen. Records for logic programming. Research Report RR-92-23, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany, 1992.

A Combinatory Logic Rewriting Relation which Supports Narrowing

Marian Vittek[1]

1 Introduction

We are concerned in solving higher order unification problem in simply typed lambda calculus. We present an procedure which finds a complete set of unifiers for a pair of terms t_1, t_2. Our approach is close to Dougherty's one [Dou90] which uses the ideas taken from a combinatory logic theory. Combinatory logic theory is an alternative, algebraic, formalization of higher-order logic. There are well known transformations between CL and λ-calculus. Dougherty shows a nice correspondence between solutions of unifications problems in both theory. Moreover weak reduction in CL seems to be a good candidate to a rewriting relation on which we can base the narrowing. Unfortunately weak reduction is too weak to provide an equivalent of $\beta\eta$-equivalence in λ-calculus. Finding such a relation is still an open problem. To enrich weak reduction, Dougherty introduced a new relation defined on systems of terms which covers weak reduction and which permits to define a unification procedure.

In our approach we define new rewriting relation defined on terms, that needs to enrich the set of combinatory logic terms by λ-abstraction. Since our relation is defined between the terms, it permits a unification procedure very similar to first order narrowing.

2 Deciding $\beta\eta$ equality on λ-terms

In this part we introduce rewriting relation which decides $\beta\eta$-equality between λ-terms. The set of terms, which we used is essentially a set of simply typed combinatory logic terms extended by a λ-abstraction constructor (or a set of simply typed λ-terms extended by combinators S,K,I). This notion of term covers the notion of term in both λ-calculus and combinatory logic theory. We have introduced moreover new equality relation (called $=_{\beta\eta SKI}$) which is essentially $\beta\eta$ equality extended by equations:
$S = \lambda x.\lambda y.\lambda z.xz(yz),\ \ K = \lambda x.\lambda y.y,\ \ I = \lambda x.x.$

[1]INRIA-Lorraine & CRIN, 615, rue du Jardin Botanique, BP 101, 54602 Villers-les-Nancy CEDEX FRANCE, E-mail: `vittek@loria.fr`

In the rest of paper we will say that a term t is η_0-*expandable*, iff $t \equiv \lambda x_0 \cdots \lambda x_k a e_0 \cdots e_n$, a is an atom, x_i is of type α_i and t is of type $\alpha_0 \to \cdots \to \alpha_k \to \alpha_{k+1} \to \beta$. Then

$$\eta_0[t] \equiv \lambda x_0 \cdots \lambda x_k \lambda x_{k+1}.ae_0 \cdots e_n x_{k+1}$$

where x_{k+1} is a variable of type α_{k+1} not occurring in t.

We will say also that a term t is *passive* iff $t = \lambda x.t_1$ or $t = at_1 \cdots t_n$, where a is an atom different from S, K, I.

Definition 1 *Let* $\to^{DR}$ *be the relation on terms defined by the following set of* DR-rules. $t \to^{DR} t'$, *iff*

- **SKI-reduction***: t' is obtained by applying one of the rewriting rules* $Sxyz \to^{DR} xz(yz), Kxy \to^{DR} x, Ix \to^{DR} x$ *on some subterm of* t, *or*
- **head** η_0**-expansion***: if* t *is* η_0*-expandable then* $t' \equiv \eta_0[t]$, *or*
- **internal** η_0**-expansion***: if* t *contains a subterm* $t_1 t_2$, *whe re* t_1 *is passive,* t_2 *is* η_0*-expandable, then* t' *is obtained by replacing* t_2 *in* t *by* $\eta_0[t_2]$.

Theorem 1 *Let* t *be a term, then every sequence of DR-reductions out of* t *terminates.*

Theorem 2 *Let* t *is a* λ*-term and let* $C(t) \to^{DR^*} t'$. t' *is DR-irreducible. Then* $t' =_\alpha \eta[t \downarrow_\beta]$.

From the theorems we can obtain a simple decision procedure for deciding $\beta\eta$-equality on λ-terms. Let t, t' are λ-terms, then $t =_{\beta\eta} t'$ iff $(C(t) \downarrow_{DR}) =_\alpha (C(t') \downarrow_{DR})$.

Example: Let $t = \lambda x.(\lambda y.\lambda z.z)x$, $t' = (\lambda x.\lambda y.\lambda z.xz(yz)).(\lambda x.\lambda y.x)$, then $C(t) = KI$, $C(t') = SK$ with appropriate typings. Now we can reduce both terms:

$$KI \to^{DR} \lambda x.KIx \to^{DR} \lambda x.I \to^{DR} \lambda x.\lambda y.Iy \to^{DR} \lambda x.\lambda y.y$$

$$SK \to^{DR} \lambda x.SKx \to^{DR} \lambda x.\lambda y.SKxy \to^{DR} \lambda x.\lambda y.Ky(xy) \to^{DR} \lambda x.\lambda y.y$$

and from this we can conclude that $t =_{\beta\eta} t'$.

3 Unification

Our unification method is based on the narrowing method using DR-reduction rules instead of classical rewriting system. In first order case the narrowing procedure transforms a pair of terms, say $< t_1, t_2 >$, by a narrowing step using a substitution σ_i until a pair of two syntacticaly unifiable terms is obtained. Then the composition of all used substitutions σ_i with most general unifier is the resulting E-unifier of terms t_1 and t_2. Single narrowing step using σ_i consist essentially of choosing a position p in term t_1 (or t_2) such that $t_{1|p}$ (or $t_{2|p}$) is syntacticaly unifiable with left side of some rewriting rule $l \to r$ via most general unifier σ_i. Then applying σ_i on pair $< t_1, t_2 >$ and providing a reduction step on $\sigma(t_1)$ (or $\sigma(t_2)$) at position p with rule $l \to r$. Narrowing gives a complete set of unifiers for theories having a presentation as canonical rewriting system.

Our case is very similar to first order case, what can matter is the presence of bound variables in terms. But thanks α conversion we can suppose (at each step) that set of free variables and set of bound variables of narrowed term are disjoint, moreover this sets are pairwise disjoint also with the sets $D(\sigma_i)$ and $I(\sigma_i)$ for each substitution σ_i. This permits us to apply the substitutions on terms in naive fashion and makes the proofs of soundness and completeness of our method analogous to first order case.

Definition 2 *Let t, t' are terms, we will say that t* narrows into *t' using substitution σ, written $t \rightsquigarrow_{\sigma}^{DR} t'$, iff*

- **SKI narrow** *there is some nonvariable position p in t SKI rewriting rule $l \rightarrow r$ and idempotent substitution σ such that $t_{|p} = s, \sigma = mgu(l, s), t' = \sigma(t[r]_p)$*
- **head η_0-expansion** *same as in the DR rules, σ is the identity substitution*
- **internal η_0-expansion** *same as in the DR rules, σ is the identity substitution*

Theorem 3 (Soundness) *Let $< t_1, t_2 > \rightsquigarrow_{\sigma}^{DR} < t_1', t_2' >$ and let θ is substitution such that $\theta t_1' =_{\beta\eta SKI} \theta t_2'$, then $(\sigma \circ \theta)t_1 =_{\beta\eta SKI} (\sigma \circ \theta)t_2$.*

Definition 3 *Set $CLNU(t_1, t_2)$ is set of all CL-substitutions θ, normalized w.r.t. SKI-reduction, such that $\theta(t_1) =_{\beta\eta SKI} \theta(t_2)$.*

Theorem 4 (Completeness) *Let t_1, t_2 are λ-terms and let $\theta \in CLNU(t_1, t_2)$. Then there is a pair of terms $< t_1', t_2' >$ and substitution δ, such that $< C(t_1), C(t_2) > \rightsquigarrow_{\sigma}^{DR*} < t_1', t_2' >, \delta(t_1') = \delta(t_2')$ and $\theta \leq \sigma \circ \delta$.*

Example: Let $t_1 = a$ and let $t_2 = Fa$, where a is a constant of type α and F is a variable. We will now show two runs of our procedure which yield to most important solutions.

- $< a, Fa > \rightarrow_{[F \rightarrow Ku]}^{DR} < a, Kua > \rightarrow^{DR} < a, u >$ where u is a variable. This terms, are unified via a substitution $\theta = [u \rightarrow a]$. So resulting unifier $\delta = [F \rightarrow Ka]$ or $\delta' = [F \rightarrow \lambda x.a]$.
- $< a, Fa > \rightarrow_{[F \rightarrow Suv]}^{DR} < a, Suva > \rightarrow^{DR} < a, ua(va) > \rightarrow_{[u \rightarrow K]}^{DR} < a, Ka(va) > \rightarrow^{DR} < a, a >$

  Here, the θ is identity substitution and resulting unifier $\delta = [F \rightarrow SKv]$ or $\delta' = [F \rightarrow \lambda x.x]$.

4 Discussion

We have presented here an procedure for HO-unification in simply typed lambda calculus. The main idea, inherited from [Dou90], is to transform a problem into combinatory logic and then looking here for the solutions. We introduce a new set of rules for deciding extensional equality in combinatory logic and we introduce unification procedure based on this set of rules. Unfortunately our procedure has infinitely branching searching space.

This is caused by the fact, that certains symbols introduced during computation have not completely determinated type and can be ameliorate by using a kind of polymorphism.

In comparison with procedure presented in [Dou90] we can point out some main differences. The transformation rule ADD ARGUMENT presented in [Dou90] changes the type of terms. So this rule can be applied only on the head of terms. For application inside the term one has to provide some DECOMPOSE steps. This is a source of problems in the case of higher order unification in combination with unification modulo some equational theory as it is shown in [Joh91][DJ92]. We have replaced the ADD ARGUMENT step by a η_0-expansion step. This rule doesn't change the type of term, so it can be applied anywhere inside the term. Our approach is than closer to standard narrowing as it is known from first order case. Moreover it allows greater area of choosing 'redex' subterm. This permits to investigate other reduction/narrowing strategies as they are known from first order case.

References

[Dou90] D. Dougherty. Higher-order unification via combinators. Technical report, Wesleyan University, May 1990. Presented at UNIF'90, Leeds (UK).

[Joh91] P. Johann. *Complete Sets of Transformations for Unification Problems.* PhD thesis, Wesleyan University (USA), 1991.

[DJ92] D. Dougherty and P. Johann. A combinatory logic approach to higher-order e-unification. In *Proceedings 11th International Conference on Automated Deduction, Saratoga Springs (NY, USA)*, 1992.

Minimal Modular Higher-Order E-Unification

Franz Weber[1]

1 Introduction

Recently Nipkow, Qian and Wang published extensions of the higher-order unification algorithm of Huet [2] in [3, 4, 5] which handle higher order E-unification for arbitrary pseudoalgebraic equational theories. Unfortunately, the extended algorithms are no longer minimal in contrast to Huet's algorithm. What are the sources of the superfluous solutions? How can we avoid them? This will be answered in this paper (see details in [7]). It will be shown that branches of the search tree may be pruned in many situations without additional runtime complexity. This may increase both the efficiency of unification and the understandability of the results in interactive usage.

2 Modular Higher-Order E-Unification

Higher order unification basically does a top down analysis of two λ-terms in β-normal form. Depending on the kind of the principal functor of the term, different transformation rules are applied. If both functors are function constants or bound variables *simplification* is applied. Simplification does just the same what first order unification algorithms do in this situation. If one of the functors is a function variable *imitation* and *projection* may be applied. Imitation instantiates the function variable with the functor of the other term while leaving the arguments given to that functor as general as possible. Afterwards, simplification is always applicable. Projection tries to instantiate the function variable with a projection to one of its arguments. Therefore, certain type conditions have to hold. Applying a breadth first search strategy to theses rules will enumerate all most general preunifiers of the two λ-terms.

Higher order E-unification requires the adaptation of simplification and imitation. The adaptation of simplification to *E-simplification* is quite similar to the combination of disjoint equational theories in first order unification [5][1]. A unification problem U is split up into two parts: a pseudoalgebraic equation P in which all *alien subterms* are

[1]Forschungszentrum Informatik (FZI), Haid-und-Neu-Straße 10-14, D-W-75 Karlsruhe 1, Germany, Email: `fweber@fzi.de`.

replaced by variables, and a set of equations M which express the mapping of variables to alien subterms. The solutions of P may be computed by a first order E-unification algorithm. Each solution is applied to M. The algorithm is applied recursively to the resulting unification problem. The application of imitation to a pair of terms is adapted by substituting all functors which might become equal to the constant functor of one term according to the underlying equational theory for the function variable of the other term.

In [4, 5] a very advanced notion of alien subterms is described. It is shown that even λ-abstractions, applications with a bound variable as functor, and 'trivial variable subterms' may be encoded as first order expressions. Only applications of a free variable as a functor to nontrivial arguments are alien subterms. Trivial arguments are a list of arguments which is exactly the list of bound variables at the occurrence of the application. The consequence of this definition of alien subterms is that all but the inherently higher order unification problems are solved by first order algorithms. 'Inherently higher order' means that imitation or projection is necessary to solve the unification problem.

Both E-simplification and E-imitation are the source for dependent solutions. This will be explained in the following chapters.

3 Minimality of E-Simplification

Even if the first-order algorithm used for E-simplification returns independent solutions the solutions might get dependent by resolving the remaining higher-order unification problems. Consider the following unification problem (variables in upper case, constants in lower case):

$$R(2) + S(2) =^? R(X) + S(X)$$

If we separate alien subterms we end up in

$$A + B =^? C + D,\ \ A =^? R(2),\ B =^? S(2),\ C =^? R(X),\ D =^? S(X)$$

Solving the first equation of this unification problem considering $+$ as a commutative operator we get as first alternative $[A \leftarrow C, B \leftarrow D]$ which leads to the unification problem

$$R(2) =^? R(X),\ S(2) =^? S(X)$$

with the solution $[X \leftarrow 2]$. The second solution of our first order problem is $[A \leftarrow D, B \leftarrow C]$ which leads to the unification problem

$$R(2) =^? S(X),\ S(2) =^? R(X)$$

with the solution $[R \leftarrow S, X \leftarrow 2]$ which is clearly dependent on the solution of our first alternative.

To avoid dependent solutions we could first compute the solutions and check for dependency afterwards. This does not work because higher order unification returns infinitely many preunifiers in general. We need a more intelligent method.

The process of higher order unification may be described by a search tree. Initially it consist only of the root which is the original unification problem. The application of a transformation rule to a node produces a child node. The tree is traversed by a breadth first search strategy. Some nodes are in *solved form* that means a solution may be easily extracted from them.

If we encounter such a solution during unification we move it along its parents back to the root. Each node on this path tries not to return dependent solutions assuming that none of its child returns dependent solutions. Therefore, it checks wether a solution for one child is also a solution of one of its siblings. If true, this sibling produces a more general solution and the solution found may be deleted. However, we have to store all deleted solutions for each child because if several children produce the same solution all of them will be deleted. We only may delete a solution if this solution solves a sibling for which it is not already deleted.

This method may only be applied for search trees with finite degree. Also, we may only delete substitutions. The deletion of preunifiers with flex-flex pairs would require to decide whether a preunifier solves a certain unification problem. For some higher order E-matching problems children which always produce dependent solutions may be pruned before actually computing the solutions for this child (cf. [7]).

4 Minimality of E-Imitation

Even if E-simplification does not lead to dependent solutions the combination of E-simplification and E-imitation leads to dependent solutions. Look at the following unification problem (+ is considered as a commutative operator):

$$P(a,b) =^? c + d \rightarrow \left\{ \begin{array}{c} P =^? \lambda x.\lambda y.G(x,y) + H(x,y) \\ G(a,b) + H(a,b) =^? c + d \end{array} \right\}$$

Application of imitation to the problem on the left hand sight leads to the unification problem at the right hand side. G and H are freshly introduced function variables of the appropriate type. Application of simplification to the second equation leads to the following two alternate solutions:

$$\left\{ \begin{array}{l} G(a,b) =^? c \\ H(a,b) =^? d \end{array} \right\}, \left\{ \begin{array}{l} G(a,b) =^? d \\ H(a,b) =^? c \end{array} \right\}$$

Applying imitation again leads to the following two alternate solutions:

$$\begin{array}{l} [P \leftarrow \lambda x.\lambda y.c + d, G \leftarrow \lambda x.\lambda y.c, H \leftarrow \lambda x.\lambda y.d] \\ [P \leftarrow \lambda x.\lambda y.d + c, G \leftarrow \lambda x.\lambda y.d, H \leftarrow \lambda x.\lambda y.c] \end{array}$$

These solutions are independent. However, G and H are auxiliary variables not occurring in the original problem. In the original unification problem P is the only free variable. In respect to P the two solutions are identical.

This happens for all imitations under commutativity. To understand this, we have to analyze the effect of one imitation followed by one simplification. We observe that the unification problems for each branch generated by simplification are equivalent up to renaming of auxiliary variables. Hence, all solutions of one branch are dependent on the solutions of the other branch. We therefore may prune one of the branches. This effect is similar for other 'permuting' equational theories (cf. [7]). For AC-unification 3 of 7 branches may be pruned due to such dependencies.

Combining one imitation with one simplification for pruning branches of the search tree prevents the incorporation of this simplification step into the first-order unification algorithm. Does pruning prevent the application of optimizations built into the first-order algorithm? Normally not. As may be seen from the example above, imitation leads to further imitations after a single simplification until a constant with no arguments is imitated. Hence, the first order algorithm would compute one simplification step in most cases. Therefore, the combination of imitation with simplification does not hinder the optimizations of the first order algorithm. Instead the combined step may be further optimized according to the idea introduced above.

5 Conclusions

Higher-Order E-unification consists of a first-order part and an intrinsically higher-order part. The first-order part is perfectly solved by the modular approach by applying first order E-unifcation algorithms. However, the technique leads to dependent solutions. Some of them may be omitted by the filter process described in this paper.

We just start to understand how the higher-order part of the problem - namely E-imitation and E-projection - is effectively solved. Transformation rules are known which solve this problem in principle. However, an optimization of these rules is possible, useful, and most likely necessary. Therefore, we have to analyze the subtle interplay between imitation, projection and an equational theory.

This paper is a first step into this direction. It shows optimizations for E-imitation for 'permutational' equational theories. Another possible clue for optimizations is to handle the series of pairs of imitation and simplification steps triggered by one imitation as described above by special algorithms.

References

[1] Franz Baader and Klaus Schulz. *Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures.* Technical Report RR-91-33. DFKI, Postfach 2080, D-W-6750 Kaiserslautern, Germany, 1991.

[2] Gerard Huet. A Unification Algorithm for Typed Lambda Calculus *Theoretical Computer Science*, 1(1):27-57, 1975.

[3] Tobias Nipkow and Zhenyu Qian Modular Higher-Order E-Unification. LNCS 488, pp 200 - 214, Springer 1991.

[4] Zhenyu Qian and Kang Wang. Higher-Order E-Unification for Arbitrary Theories. *Proceedings of 1992 Int. Joint Conference and Symposium on Logic Programming*, MIT Press, Dec. 1992.

[5] Zhenyu Qian and Kang Wang. Higher-Order E-Unification for Arbitrary Theories. *Proceedings of UNIF92*, this report.

[6] M. Schmidt-Schauß. Combination of Unification Algorithms. *J. of Symbolic Computation*, 8, 1989.

[7] Franz Weber *Minimal Modular Higher-Order E-Unification.* Technical Report ProSt/hl-rep-16. FZI, Haid-und-Neu-Str. 10-14, D-W-75 Karlsruhe, Germany, 1992.

On Approaches to Order-Sorted Rewriting

Andreas Werner [1]

Order-sorted rewriting builds a nice framework to handle partially defined functions and subtypes. Differing from many-sorted rewriting, the critical pair lemma and Birkhoff's completeness theorem do not hold in general. To retain a critical pair lemma, *Smolka & al* restrict rewriting to sort decreasing rules. In the last year efforts have been made in order to get a more general critical pair lemma. In this work we present a new approach to order-sorted rewriting and compare it with the other ones.

In order to retain Birkhoff's completeness theorem we generalize Σ-terms to (Σ, E)-terms (cf. [8], [5]) which denote well-defined elements in every algebra satisfying E. We also generalize Σ-substitutions in such a way that unification is decidable and finitary and a critical pair lemma can be proven under the additional restriction that every function symbol has only one range sort. Such signatures can be obtained by adding new rules. Starting from well-defined (Σ, E)-terms, e.g. Σ-terms, we can prove that all terms obtained by rewriting and unification are also (Σ, E)-terms and therefore well-defined in every Σ-algebra satisfying E.

In general it is not decidable whether an expression is a (Σ, E)-term or whether a (Σ, E)-term belongs to a certain sort of terms even if the given rewriting system is canonical and there is only one declaration for every function symbol. In the case of (weakly) sort decreasing and confluent term rewriting systems both properties are decidable by normalization of the given expression.

In [1], *Chen* & *Hsiang* add new function symbols and function declarations to the given (regular) signature in order to get weakly sort decreasing rules. This may yield a non-regular signature. In [9], *L. With* uses term declarations to transform a term rewriting system to a (weakly) sort decreasing one. Unfortunately, due to the (additional) term declarations unification becomes undecidable (cf. [5]). Furthermore there exist canonical (due to our critical pair lemma) rewriting systems which can't be transformed equivalently in a (weakly) sort decreasing one.

C. and H. Kirchner presented a different approach. To each term a type expression is associated expressing possible types of the term which are deduced during computation. Unfortunately, their unification is in general undecidable.

In [2], *H. Comon* uses sort constraints based on a fragment of second-order logic in order to get a critical pair lemma. Unfortunately, the corresponding completion procedure

[1] SFB 314, University of Karlsruhe, P.O. Box 6980, D-7500 Karlsruhe 1, Germany, E-mail: werner@ira.uka.de.

does not terminate even in cases of termination of the other approaches.

This work was supported by the Deutsche Forschungsgemeinschaft as part of the SFB 314.

References

[1] H. Chen and J. Hsiang: Order-Sorted Specification and Completion. *Technical Report. Department of Computer Science, SUNY at Stony Brook.* 1991.

[2] H. Comon: Completion of Rewrite Systems with Membership Constraints. *Research Report. CNRS & LRI, Paris.* 1991.

[3] G. Huet: Confluent Reductions: Abstract Properties and Application to Term Rewrite Systems. *Journal of ACM 27 (1980),* pages 797 - 821.

[4] C. Kirchner and H. Kirchner: Order-Sorted Rewriting Computation in G-algebra. *Technical Report. INRIA-Lorraine* & *CRIN, Nancy.* 1991.

[5] M. Schmidt-Schauß: Computational Aspects of an Order-Sorted Logic with Term Declarations. *LNAI 395, Springer-Verlag,* 1989.

[6] G. Smolka, W. Nutt, J. A. Goguen and J. Meseguer: Order-Sorted Equational Computation, *in: H. Ait-Kaci, M. Nivat: Resolution of Equations in Algebraic Structures, Academic Press,* Volume 2, pages 297 - 367, 1989.

[7] U. Waldmann: Unification in Order-Sorted Signatures. *Forschungsbericht Nr. 298. Universität Dortmund.* 1989.

[8] L. With: Completeness Without Confluence and Confluence Without Sort Decreasingness of Order-Sorted Rewriting. *Draft. TU Berlin.* 1990.

[9] L. With: Completeness and Confluence of Order-Sorted Term Rewriting. *3rd CTRS, to appear also as a technical report.* 1992.

The Decidability of Higher-Order Matching

D.A. Wolfram[1]

1 Introduction

We consider the decidability of a form of matching in the simply typed λ--calculus. It is similar to higher-order unification [2] except that one term at most in each disagreement pair has free variables.

Example 1.1 Here is an example of a higher-order matching problem:

$$\{\langle f(f(x)),\ A(A(B))\rangle\}$$

where $\tau(f) = \tau(A) = (\iota \to \iota)$ and $\tau(x) = \tau(B) = \iota$.

Variables are denoted by lower-case letters, and constant symbols by upper-case ones. Example 1.1 has the following solutions:

$$\begin{aligned}
\theta_0 &= \{\langle f,\ \lambda y.y\rangle,\ \langle x,\ A(A(B))\rangle\}\\
\theta_1 &= \{\langle f,\ \lambda u.A(u)\rangle,\ \langle x, B\rangle\}\\
\theta_2 &= \{\langle f,\ \lambda u.A(A(B))\rangle\}.
\end{aligned}$$

In 1976, Huet [3] stated the following about higher-order matching:

This problem is decidable, but the proof is probably hard.

More recently, third-order matching hs been shown to be decidable [1], but its decidability at higher orders appears to remain unknown.

We show that third-order matchability is NP-hard by a reduction from propositional satisfiability. The projection property is then introduced. If it is undecidable, then higher-order matching is undecidable; if not, then a type restriction on variables in terms gives a group of decidable higher-order matching problems of arbitrarily high order.

[1]Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, United Kingdom. Email: `David.Wolfram@prg`

2 Matchability is NP-Hard

As a simplification, we shall assume that all terms are in $\beta\overline{\eta}$-normal form: they have no β-redexes, and every subterm of the form t whose type is not a base type has been converted to $\lambda x.t(x)$ where x is not a free variable of t.

Another simplification is that no term will contain any constant symbol. This does not lead to any loss of generality because Statman's mapping [4], called the $*$-map, of higher-order unification problems to those in the pure simply-typed λ-calculus is a bijection.

Example 2.1 When the $*$-map is applied to Example 1.1, we obtain the following pure matching problem.

$$\{\langle f(f(x)), A(A(B))\rangle\} \leftrightarrow$$
$$\{\langle \lambda y_1 y_2.h_1(y_1, y_2, h_1(y_1, y_2, h_2(y_1, y_2))),\ \lambda y_1 y_2.y_1(y_1(y_2))\rangle\}$$

The solutions to this pure matching problem are in one-to-one correspondence with the solutions in Example 1.1.

$$\{\langle f, \lambda y.y\rangle, \langle x, A(A(B))\rangle\} \leftrightarrow \{\langle h_1, \lambda z_1 z_2 z_3.z_3\rangle, \langle h_2, \lambda z_1 z_2.z_1(z_1(z_2))\rangle\}$$
$$\{\langle f, \lambda u.A(u)\rangle, \langle x, B\rangle\} \leftrightarrow \{\langle h_1, \lambda z_1 z_2 z_3.z_1(z_3)\rangle, \langle h_2, \lambda z_1 z_2.z_2\rangle\}$$
$$\{\langle f, \lambda u.A(A(B))\rangle\} \leftrightarrow \{\langle h_1, \lambda z_1 z_2 z_3.z_1(z_1(z_2))\rangle\}$$

Theorem 2.2 *Third-order matchability is NP-hard.*

Proof: We give a polynomial reduction of propositional satisfiability to third--order matchability which uses the following encodings.

- A constant m is represented by the simply typed Church numeral $\overline{m} = \lambda xy.\underbrace{x(x(\cdots x(}_{m}y)\cdots))$ with type restricted to $N = ((\iota \rightarrow \iota), \iota \rightarrow \iota)$.

- Multiplication of terms of type N is represented by

  $$\overline{\times} = \lambda xyz.x(y(z))$$

  with type $(N, N \rightarrow N)$, where $\tau(z) = (\iota \rightarrow \iota)$.

- Addition of terms of type N is represented by

  $$\overline{+} = \lambda xyzt.x(z, y(z, t))$$

  of type $(N, N \rightarrow N)$, where $\tau(z) = (\iota \rightarrow \iota)$ and $\tau(t) = \iota$.

- The dyad combinator is represented by

  $$\mathsf{D} = \lambda uvxyz.x(\lambda t.v(y, z), u(y, z))$$

  with type $(N, N, N \rightarrow N)$.

Here are the encodings of propositional logic:

- Propositional variable: $enc\,x = \mathsf{D}\overline{0}\,\overline{1}\overline{x}$, where $\overline{x} = \lambda uv.x(\lambda t.u(t), v)$ with type N.
- Conjunction: $enc\,(x \wedge y) = \mathsf{D}\overline{0}\,\overline{1}(enc\,x \,\overline{\times}\, enc\,y)$.
- Disjunction: $enc\,(x \vee y) = \mathsf{D}\overline{0}\,\overline{1}(enc\,x \mp enc\,y)$.
- Negation: $enc\,(\neg x) = \mathsf{D}\overline{1}\,\overline{0}\,(enc\,x)$.

If X is a propositional formula, it is straightforward to verify that the third-order matchability problem $\{\langle encX, \overline{1}\rangle\}$ has an affirmative answer if and only if X is satisfiable, and that $enc\,X$ can be constructed in polynomial time. $\Box$

3 The Projection Property

The projection property is based on the effect of projection substitutions in Huet's higher-order unification procedure [2].

Definition 3.1 Given a term $t = \lambda x_1 \cdots x_n. \underbrace{@}_{head}(t_1, \ldots, t_m)$, the *projection property* holds if and only if there exists a substitution π such that $head(t\pi) = x_i$ where $1 \leq i \leq n$.

If this property does not hold for a disagreement pair in a matching problem where the head of the rigid term corresponds to x_i, then the matching problem has no solution. More strongly, if the projection property is undecidable then higher-order matching is undecidable. We make the following conjecture.

Conjecture 3.2 The projection property is decidable if and only if higher--order matching is decidable.

Sometimes it is possible to predict when the projection property holds, and this depends on the form of a sequence of subterms.

Definition 3.3 A sequence

$$\lambda x_1 \cdots x_n.@(t_1, \ldots, t_m) = E_1, E_2, \ldots, E_k$$

of terms is a *projection path* for x_i where $1 \leq i \leq n$ if and only if:

- E_j is an immediate subterm of E_{j-1} where $2 \leq j \leq k$.
- $head(E_k) = x_i$.
- There is no $l : 1 \leq l < k$ such that $head(E_l) \in \{x_1, \ldots x_n\}$.

Example 3.4 A projection path for v is:

$$\begin{aligned}&\lambda uv.f(f(x,\lambda g.h(v,u)),\lambda x.h(x,v)),\\&f(x,\lambda g.h(v,u)),\\&\lambda g.h(v,u),\\&v\end{aligned}$$

There are two kinds of projection path, and they are defined below.

Definition 3.5 A projection path $E_1,\ldots,E_k$ is *unmixed* if and only if whenever $head(E_i) = head(E_j)$ where $1 \leq i < j \leq k$, E_{i+1} and E_{j+1} are corresponding immediate subterms. A projection path is a *mixed* projection path if and only if it is not an unmixed projection path.

We can make correct predictions about the projection property for the former kind of projection path.

Lemma 3.6 The projection property holds for x in a term t if there is an unmixed projection path for x in t.

Example 3.7 The term $\lambda uv.f(\lambda x.f(\lambda g.h(v,u),x),y)$ has an unmixed projection path for v. The projection property holds for v with the substitution

$$\pi = \{\langle f,\ \lambda w_1 w_2.w_1(h_1(w_1,w_2))\rangle, \langle h,\ \lambda w_3 w_4.w_3\rangle\}.$$

Remark 3.8 The projection property can hold for mixed projection paths. It holds for v in $\lambda uv.f(f(x,\lambda g.h(v,u)),\lambda x.h(x,v))$ with the substitution

$$\pi = \{\langle f,\ \lambda w_1 w_2.w_2(w_1)\rangle, \langle h, \lambda w_3 w_4.w_3\rangle\}.$$

We now define a form of matching problem for which the projection property holds.

Lemma 3.9 *The projection property holds provided that for all variables f where $\tau(f) = (\alpha_1,\ldots,\alpha_n \to \beta)$ and β is a base type we have $\alpha_i = \alpha_j$ if and only if $i = j$ where $1 \leq i,j \leq n$.*

Proof: The condition on types in Lemma 3.9 ensures that all projection paths are unmixed. □

4 Discussion

We have shown that third-order matchability is NP-hard by a polynomial reduction from propositional satisfiability. We have also conjectured that the decidability of higher-order matching is equivalent to a property related to projection substitutions, and defined a group of decidable matching problems of arbitrarily high order. The techniques we introduce could be used in resolving the decidability of the general problem [5].

Acknowledgements

This research was supported by a Research Fellowship at Christ Church, Oxford, and Oxford University Computing Laboratory.

References

[1] G. Dowek. Third order matching is decidable. *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, D.C., 1992.

[2] G.P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science* **1** (1975) 27–57.

[3] G.P. Huet. *Résolution d'équations dans des Langages d'Ordre* $1, 2, \ldots, \omega$. Thèse de Doctorat d'Etat, Université Paris VII, Paris, (1976).

[4] R. Statman. On the existence of closed terms in the typed λ-calculus II: transformations of unification problems. *Theoretical Computer Science* **15** (1981) 329–338.

[5] D.A. Wolfram. *The Clausal Theory of Types*. Cambridge Tracts in Theoretical Computer Science **21**, Cambridge University Press, 1992. (To appear).

Unifying Cycles

Jörg Würtz [1]

Two-literal clauses of the form $\{L \leftarrow R\}$ occur quite frequently in logic programs, deductive databases, and—disguised as an equation—in term rewriting systems. These clauses define a cycle if the atoms L and R are weakly unifiable, i.e., if L unifies with a new variant of R. The obvious problem with cycles is to control the number of iterations through the cycle. Here we consider the cycle unification problem of unifying two literals G and F modulo a cycle $\{Pl_1 \dots l_n \leftarrow Pr_1 \dots r_n\}$, i.e., we consider additional clauses as follows. A goal clause – referred to as *(calling) goal* – of the form $\leftarrow Ps_1 \dots s_n$, which calls the cycle, and a fact – called *(terminating) fact* – of the form $Pt_1 \dots t_n$, which terminates the cycle. A *cycle unification problem* is then the following one:

> Is there a substitution σ such that $\sigma Ps_1 \dots s_n$ is a logical consequence of $\{Pl_1 \dots l_n \leftarrow Pr_1 \dots r_n\}$ and $Pt_1 \dots t_n$?

If such a substitution σ exists, then σ is said to be a *solution* for the cycle unification problem. For more general cases, cycle unification can be defined in an analogue way. To solve a cycle unification problem $\langle G \overset{\circ}{\longrightarrow} F\rangle_{\{L \leftarrow R\}}$, consisting of the goal G, the fact F and the cycle $\{L \leftarrow R\}$, we have to find a substitution which either unifies G and F or simultaneously unifies each equation in

$$\mathcal{C}^k = \{G \doteq L^1\} \cup \{R^i \doteq L^{i+1} \mid 1 \leq i \leq k\} \cup \{R^{k+1} \doteq F\}$$

where k denotes the number of iterations through the cycle.[2]

In order to be able to control a cycle we have to answer the following questions. Is cycle unification decidable? How many independent most general solutions has a cycle unification problem? Does there exist a unification algorithm which enumerates a minimal and complete set of solutions for a cycle unification problem? Answers to these questions may help to increase the power of automated theorem provers significantly. For example, if a cycle is embedded in a larger formula and it can be determined that the corresponding cycle unification problem is unsolvable, then the clauses defining the cycle can be eliminated from the formula. If a minimal and complete set Σ of solutions for a cycle unification problem exists and can be enumerated, then any other solution is subsumed

[1] DFKI, Stuhlsatzenhausweg 3, 6600 Saarbrücken, Germany, Email: `wuertz@dfki.uni-sb.de`.

[2] By X^k we denote the syntactic object where each variable occurring in X has the index k attached to it.

by a solution in Σ and need not to be considered. If Σ is finite, then this may prune a potentially infinite search space to a finite one.

Although cycle unification is of significant importance for the field of automated deduction and logic programming, it has received surprisingly little attention in the literature. Function-free cycle unification problems, i.e., cycle unification problems defined over variables and constants only, occur mainly in deductive databases and it can be shown that under certain conditions these problems do not give rise to infinite computations (cf. [MN83]). M. Schmidt–Schauß [SS88] has shown that cycle unification is decidable provided that the goal and the fact are ground, i.e., they do not contain variable occurrences. Independently, P. Devienne [Dev90] has given a more general result for cycle unification problems with linear goals and facts, i.e., each variable occurs at most once in the goal and the fact. He uses essentially the same ideas as Schmidt–Schauß, but a very special technique based on directed weighted graphs. Devienne's results were used by De Schreye et al. [DVB90] to decide whether cycles admit non-terminating queries to deductive systems. Another approach has been taken by H.J. Ohlbach who represented sets of terms by so-called abstraction trees which may compress the search space. Moreover, abstraction trees can be used to compile two-literal clauses and in certain cases a finite abstraction tree can represent infinitely many solutions of a cycle unification problem [Ohl90]. A further approach for unifying infinite sets of terms which are encoded in so called ρ-terms is described in [Sal92]. The incorporation of ρ-terms into logic programming allows on the one hand infinite queries and the finite representation of infinitely many answers. On the other hand, it avoids repeated computation and certain kinds of infinite loops, without changing the denotational semantics of the programs.

In [BHW92] we developed the theoretical foundations for cycle unification. For various classes of restricted cycle unification problems we showed their decidability, proved that they have at most finitely many most general solutions and constructed an algorithm to compute this set. The most general result concerned the class of non-recursive matching cycles $\mathcal{C}_{nrm}$, i.e., cycles $\{L \leftarrow R\}$ for which there exists a substitution σ such that $\sigma L = R$ or $L = \sigma R$ and $\nexists i : x \mapsto t \in \sigma^i$, $x \in \mathcal{V}ar(t)$, and $t \neq x$.[3] This work is an extension of the previous one since we consider unifying cycles, i.e., the left-hand and the right-hand side are unifiable [Wür92].

All results depend on certain kinds of paths in a dependency graph. Consider the cycle unification problem $C = \langle G \multimap\!\!\to F\rangle_{\{L \leftarrow R\}}$. A dependency graph relates variables occurring in the goal G and variables of the cycle $\{L \leftarrow R\}$. First, the variables in G are related to the variables occurring in the first instance of the left-hand side of the cycle, i.e., L^1. Second, the variables in the i-th instance of the right-hand side of the cycle, i.e., R^i, are related to the variables in the $i+1$-st instance of the left-hand side, i.e., L^{i+1}. Consider as an example the cycle unification problem

$$C = \langle Pu_1u_2u_3u_4u_5 \multimap\!\!\to Pababa\rangle_{\{Pyvywz \leftarrow Pxyvyw\}}.$$

We obtain as the most general unifier of $Pu_1u_2u_3u_4u_5$ and $Py^1v^1y^1w^1z^1$ the substitution

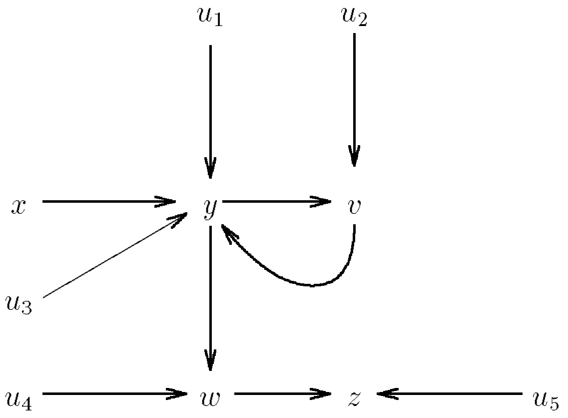
$$\sigma_0 = \{u_1 \mapsto y^1,\ u_2 \mapsto v^1,\ u_3 \mapsto y^1,\ u_4 \mapsto w^1,\ u_5 \mapsto z^1\}.$$

[3] By σ^i we denote the i–fold composition of σ with itself.

Furthermore, we obtain as the most general unifier of $Px^iy^iv^iy^iw^i$ and $Py^{i+1}v^{i+1}y^{i+1}w^{i+1}z^{i+1}$) the substitution

$$\sigma_i = \{x^i \mapsto y^{i+1},\ y^i \mapsto w^{i+1},\ v^{i+1} \mapsto w^{i+1},\ v^i \mapsto y^{i+1},\ w^i \mapsto z^{i+1}\}.$$

From these substitutions we derive a dependency graph representing the bindings in the most general unifiers computed above:



The dependency graph defines some paths which we call linear paths and permutations. In our example the graph defines the restricted linear paths $\langle w, z\rangle$ and $\langle z\rangle$ and the permutation $\langle y, v, y\rangle$ of lengths 2, 1 and 3, respectively.

The most general result concerns cycles $\{L \leftarrow R\}$ such that L and R are unifiable, i.e., there exists a substitution σ such that $\sigma L = \sigma R$; they are called unifying cycles. The class containing only cycle unification problems with unifying cycles is called $\mathcal{C}_u$. Let $\langle G \overset{\circ}{\longrightarrow} F\rangle_{\{L\leftarrow R\}}$ be a cycle unification problem in the class $\mathcal{C}_u$. The following steps define an algorithm for unifying cycles to compute a minimal and complete set of most general solutions.

1. If G and F are unifiable, then compute τ as the most general unifier for G and F restricted to the variables in G.

2. Compute the dependency graph for $\langle G \overset{\circ}{\longrightarrow} F\rangle_{\{L\leftarrow R\}}$.

3. If $\langle G \overset{\circ}{\longrightarrow} F\rangle_{\{L\leftarrow R\}} \in \mathcal{C}_u$, then compute the lengths $l_1, \ldots, l_i$ of all defined restricted linear paths and the lengths $m_1, \ldots, m_j$ of all defined permutations. Let $m = \max(1, l_1, \ldots, l_i)$ and $N = \mathrm{lcm}(1, m_1 - 1, \ldots, m_j - 1)$.

4. If $\mathcal{C}^k$ is solvable, then compute τ_k as the most general unifier for $\mathcal{C}^k$, restricted to the variables occurring in G, $0 \leq k \leq m + N - 2$.

5. Let Σ be the set of solutions obtained in steps (1) and (4). If $\Sigma = \emptyset$, the problem is unsolvable. Otherwise, iteratively eliminate a substitution α if the current set of solutions contains another substitution δ with $\delta \lessdot \alpha\ [\mathcal{V}ar(G)]$. The obtained set is a minimal and complete set of solutions for the cycle unification problem $\langle G \overset{\circ}{\longrightarrow} F\rangle_{\{L\leftarrow R\}}$.

As a result we obtain the following theorem for cycle unification problems in the class $\mathcal{C}_u$.

Theorem 1 *Let C be a unifying cycle.*

(i) $\langle G \xrightarrow[C]{\circ} F \rangle$ *is decidable.*

(ii) $\langle G \xrightarrow[C]{\circ} F \rangle$ *is finitary.*

(iii) *There exists an algorithm computing a minimal and complete set of solutions for* $\langle G \xrightarrow[C]{\circ} F \rangle$.

References

[BHW92] W. Bibel, S. Hölldobler, and J. Würtz. Cycle unification. In D. Kapur, editor, *Proceedings of the Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 94–108. Springer, June 1992.

[Dev90] P. Devienne. Weighted graphs: A tool for studying the halting problem and time complexity in term rewriting systems and logic programming. *Journal of Theoretical Computer Science*, 75:157–215, 1990.

[DVB90] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A practical technique for detecting non–terminating queries for a restricted class of horn clauses, using directed, weighted graphs. In *Proceedings of the International Conference on Logic Programming*, pages 649–663, 1990.

[MN83] J. Minker and J.M. Nicolas. On recursive axioms in deductive databases. *Information Systems*, 8(1):1–13, 1983.

[Ohl90] H.J. Ohlbach. Compilation of recursive two–literal clauses into unification algorithms. In *Proceedings of the AIMSA*, 1990.

[Sal92] G. Salzer. Solvable classes of cycle unification problems. In *IMYCS*, 1992. To appear.

[SS88] M. Schmidt-Schauß. Implication of clauses is undecidable. *Journal of Theoretical Computer Science*, 59:287–296, 1988.

[Wür92] J. Würtz. Unifying cycles. In B. Neumann, editor, *Proceedings of the European Conference on Artificial Intelligence*, pages 60–64. John Wiley & Sons, August 1992.