Higher-Order Equational Logic Programming

Zhenyu Qian*
Universität Bremen†

Abstract

Higher-order equational logic programming is a paradigm which combines first-order equational and higher-order logic programming, where higher-order logic programming is based on a subclass of simply typed λ-terms, called higher-order patterns. Central to the notion of higher-order equational logic programming is the so-called higher-order equational unification. This paper extends several important classes of first-order equational unification algorithms to the higher-order setting: only problems of the extensions are discussed and first-order equational unifications are viewed as black boxes whenever possible.

We first extend narrowing and show that the completeness of many higher-order narrowing strategies reduces to that of their underlying first-order counterparts. Then we propose an algorithm for higher-order equational unification of free higher-order patterns in an arbitrary equational theory. Finally a general approach to extend first-order unification combination algorithms is sketched informally. The termination property of the above higher-order extensions is considered in a uniform way.

1 Introduction

Higher-order logic programming paradigm, like e.g. L_λ [20], Elf [27] or the recent implementation [23] of Isabelle [25], provides a powerful and efficient computational mechanism for succintly representing and manipulating syntactical structures involving notions of abstractions, scope, bound and free variables, and is therefore very suitable for syntactically handling formulas, types, proofs and programs. Equational logic programming paradigm (as in [7, 10, 12, 13, 21, 31, 32, 35, 39] and in the collection [6]) includes first-order extensions of Prolog, where symbols can be specified by equations. Although the higher- and first-order paradigms are successful on their own, no techniques of one paradigm, to our knowledge, have been systematically used in the other. In this paper we investigate the integration of them.

The integration is important from the perspective of higher-order logic programming, since equations exist naturally in many mathematical systems and facilities to handle them naturally ease the manipulation of these mathematical systems in the higher-order setting. Examples are the mathematical systems that include the number theory, where arithmetic operations on numbers may be naturally defined by equations. Changing the perspective, extending equational logic programming to a higher-order setting is also important, since equational theories are often subject to some syntactical structures, which can be naturally formulated in a higher-order setting. Examples are parameterized many-sorted algebraic specifications.

The higher-order logic programming paradigm mentioned above is based on a subclass of λ-terms, discovered by Miller, where the unification is decidable, unifiable terms always have a most general unifier [20] and a most general unifier can be computed in linear time and space [28]. This subclass consists of all β-normal forms where free variables F may only occur in the form $F(x_1, \cdots, x_k), k \geq 0$, with $x_1, \cdots, x_k$ being distinct bound variables. We follow Nipkow [22] to call these terms *higher-order patterns* (short: *patterns*). Their unification is called *higher-order unification* in this paper. Roughly speaking,

*Research partially supported by ESPRIT Basic Research WG *COMPASS* 6112.

†Address: FB 3 Informatik, Universität Bremen, D-28334 Bremen, Germany. E-mail: qian@informatik.uni-bremen.de

POPL 94- 1/94, Portland Oregon,USA

the higher-order logic programming paradigm can be viewed as an extension of Prolog where first-order terms are replaced by patterns, and first-order unification by higher-order one. Just as in the first-order case, where unification modulo equations, usually called *equational unification*, is central to the notion of equational logic programming, so is higher-order unification modulo equations, called *higher-order equational unification* hereafter, to the notion of higher-order equational logic programming. Unification modulo no equations is said to be *syntactic*.

In the first-order setting, a unification algorithm in an equational theory can either be designed directly or composed of existing unification algorithms for disjoint sub-theories (cf. e.g. [1, 17]). In the higher-order case, the situation seems to be similar. However, we should make use of first-order algorithms whenever possible. Nipkow, Qian and Wang proposed some equational unification algorithms, which accept all simply typed λ-terms and are parameterized by arbitrary first-order equational unification algorithms ([24, 29]). But the restrictions of these algorithms to higher-order patterns inherit some unpleasant operational properties from syntactic unification of simply typed λ-terms. The aim of this paper is to extend three important kinds of first-order equational unification algorithms to the higher-order setting, where the resulting algorithms have operational properties close to those in the first-order case.

Firstly, we consider functional logic programming systems (in Section 4), which are logic programming systems, where function symbols may be specified by directed equations, called *rewrite rules*. The equational unification in this case is also called *narrowing*. For example, let the usual addition $+$ be defined by $0 + X \rightarrow X$ and $suc(X) + Y \rightarrow suc(X + Y)$ where 0 is zero and suc the successor function as usual. Then narrowing $X + Y =^? suc(Y)$ may yield a solution $\{X \mapsto suc(0)\}$.

We introduce a notion of *higher-order narrowing* such that e.g. the unification problem $\lambda y.F(y) + y =^? \lambda y.suc(y)$ with F being a free (function) variable may be solved modulo the above rewrite rules. Although different first-order narrowing strategies have been considered in the literature, which have led to very different first-order functional logic programming systems (cf. e.g. [7, 10, 12, 13, 21, 31, 32, 35, 39] and the collection [6]), it will be shown here that if a strategy of higher-order narrowing is a *total* higher-order extension of its first-order restriction in a certain sense, then the completeness of the strategy reduces to that of the first-order restriction.

As the second contribution of this paper, the notion of unifiers is extended and an algorithm is presented (in Section 6) for computing the most general unifiers of free patterns in the presence of an arbitrary equational theory. Note that in the first-order case, syntactic unification is conservative in equational unification in the sense that if the terms to be unified are free, (i.e. contain no function symbols occurring in the equations,) then their equational unification is just syntactic unification. This property is lost in the higher-order case.

Consider the unification problem $\lambda xy.F(x, y) =^? \lambda xy.F(y, x)$ in the presence of a usual commutative function symbol $+$, where F is a free variable. Let θ_n denote the substitution

$$\{F \mapsto \lambda xy.G_n(H_1(x, y) + H_1(y, x), \cdots, H_n(x, y) + H_n(y, x))\},$$

where n is a natural number, G_n, $H_i, i = 1, \cdots, n$, are distinct free variables. It may be easily checked that θ_n is a unifier of the above unification problem. But the syntactical unification of $\lambda xy.F(x, y) =^? \lambda xy.F(y, x)$ yields a most general unifier $\{F \mapsto \lambda xy.H\}$ with a new free variable H, which is obviously not a most general one in the presence of $+$. Thus, the algorithms in [20], [22] and [27] are not complete for equational unification of free patterns. In general, θ_n is not a most general unifier either, since θ_{n+1} is a unifier and strictly more general than θ_n. Note that $\lambda xy.F(x, y)$ and $\lambda xy.F(y, x)$ are both free patterns and θ_n contains λ-terms that are not patterns. It may be proved that in the simply typed λ-calculus every unifier of the above unification problem is an instance of θ_n for some n. Therefore, the higher-order unification is *not* finitary, even if all simply typed λ-terms are allowed in the solutions.

In the first-order case, a unification algorithm for an equational theory may be built by combining unification algorithms for the disjoint sub-theories using some combination algorithm (cf. e.g. [2, 9, 33, 38]). Higher-order equational unification can also be pursued in an analogous way. We informally sketch (in Section 7) a uniform way of extending first-order combination algorithms to the higher-order setting. Higher-order combination algorithms combine the higher-order equational algorithms proposed in this paper, and also those developed elsewhere (cf. [30] for a higher-order AC unification algorithm).

Termination of a unification algorithm is an important property, since the decidability property may depend on it. A general method is proposed (in Section 5) which reduces the termination of a higher-order equational unification to its first-order counter-

part. The method is applied several times in this paper.

2 Other related work

This section discusses other related work not mentioned in Section 1.

Breazu-Tannen and Meyer were the first who showed that the integration of a first-order equational theory in a simply typed λ-calculus is conservative w.r.t. the first-order equational theory [5]. The point was then made more clearly by Breazu-Tannen and Gallier [3, 4] where the computational reductions in the integrations are studied.

In the spirit of "universal unification" [11], Snyder studied equational unification of simply typed λ-terms [34]. Wolfram considered the same problem in terms of a general form of term rewriting [37], Dougherty and Johann for restricted equational theories in a combinatory logic framework [8]. As already mentioned, Nipkow, Qian and Wang proposed a unification algorithm parameterized by first-order equational unification algorithms ([24, 29]). Weber and Möller implemented the Nipkow, Qian and Wang's algorithm in their software development system (cf. [36]). All the above algorithms inherit some serious problems from the syntactic unification of simply typed λ-terms: the unification is undecidable and unifiable terms may have infinite independent unifiers. With a higher-order logic programming paradigm based on patterns we have a new starting point in this paper.

3 Preliminaries

This section briefly introduces our notations. More technical details are given in Appendix A.

We follow the standard notations for the simply typed λ-calculus and use the following conventions: s, t, u and v stand for the simply typed λ-terms (short: terms), x, y and z for bound variables, X, Y, Z, F, G and H for free variables, c, d, f, g and h for function symbols, a and b for atoms (i.e. function symbols, bound or free vaiables). Bound and free variables are always kept disjoint. The set of all bound (or free) variables in a syntactic object O is denoted by $\mathcal{BV}(O)$ (or $\mathcal{FV}(O)$). The letters α, β, τ range over types, and $\alpha_1 \to \cdots \to \alpha_n \to \beta$ or $\overline{\alpha_n} \to \beta$ stands for $\alpha_1 \to (\cdots \to (\alpha_n \to \beta))$.

We may write $\lambda\overline{x_n}.s$, or simply $\lambda\overline{x}.s$ for $\lambda x_1 \ldots \lambda x_n.s$, where s is not an abstraction, and $a(u_1, \ldots, u_n)$, $a(\overline{u_n})$ or $a(\overline{u})$ for $(((a\ u_1)\ u_2) \ldots u_n)$. In the same context, occurrences of $\overline{x}$ (or $\overline{u}$) always denote the occurrences of the same $\overline{x_n}$ (or $\overline{u_n}$, resp.). If $\overline{u}$ stands for $\overline{u_n}$ and $\overline{v}$ for $\overline{v_n}$ then $\{\overline{u \mapsto v}\}$ stands for $\{\overline{u_k \mapsto v_k}\}$.

Let $\mathcal{X} \in \{\beta, \eta, \beta\eta\}$. We use $\to_{\mathcal{X}}$ to denote one step $\mathcal{X}$-reduction, $\to^*_{\mathcal{X}}$ the reflexive and transitive closure and $=_{\mathcal{X}}$ the equivalence of $\to_{\mathcal{X}}$. Define $\mathcal{H}(\lambda\overline{x}.a(\overline{t_n})) = a$. Let $s{\downarrow_\beta}$ denote the β-normal form of s. An *η-long* form is a β-normal form $\lambda\overline{x}.a(\overline{t_n})$ with $a(\overline{t_n})$ being of a base type and each t_i being η-long. Use $s{\uparrow_\eta}$ to denote the unique η-long form such that $s{\uparrow_\eta} \to^*_\eta s{\downarrow_\beta}$. The η-long form of a single bound variable x may still be written as x.

Patterns are the η-long forms, in which arguments of a free variable may only be distinct bound variables. If $F(y, x)$, $G(z)$, $f(G(x))$ and $y(\lambda z.G(z), f(G(x)))$ are of base types, then terms $\lambda xyz.F(y, x)$ and $\lambda xy.y(\lambda z.G(z), f(G(x)))$ are patterns. The terms $\lambda x.F(c, x)$, $\lambda xy.F(x, x)$ and $\lambda x.F(G(x))$ are not patterns. In the sequel, β-reductions are always assumed to be performed automatically, and all terms are patterns unless stated otherwise.

We use $\mathcal{O}(s)$ to denote the set of the positions of all subterms in a pattern s, $\overline{\mathcal{O}}(s)$ that of all rigid subterms. For $p \in \mathcal{O}(s)$, $s_{|p}$ denotes the subterm of s at p, $s[u]_p$ the result of replacing $s_{|p}$ in s by u.

A substitution may be written as $\{X_1 \mapsto t_1, \cdots, X_n \mapsto t_n\}$ or $\{\overline{X_n \mapsto t_n}\}$. The letters σ, θ, ρ range over substitutions. Let $\theta = \{\overline{X_n \mapsto t_n}\}$. Then $\mathcal{D}(\theta) = \{X_1, \cdots, X_n\}$ and $\mathcal{I}(\theta) = \mathcal{FV}(t_1, \cdots, t_n)$. The *restriction* of θ to a variable set $\mathcal{W}$ is denoted by $\theta_{|\mathcal{W}}$, the *composition* of σ and θ by $\sigma\theta$ satisfying $\sigma\theta(X) = \sigma(\theta(X))$ for every X. We call θ a *variable renaming* if $t_1, \cdots, t_n$ are distinct free variables.

A rewrite rule $l \to r$ is an ordered pair, an equation $l \simeq r$ an unordered pair of first-order terms. A term rewrite system (short: TRS) R is a finite set of rewrite rules, an equational theory E a finite set of equations.

In this paper, many notions and notations defined for rewrite rules may also be used for equations in an obvious way. For example, function symbols *free* w.r.t. R are those not occurring in R, and function symbols free w.r.t. E means similar.

In the sequel, all TRS's are assumed to be consistent and confluent. In Section 4, every rewrite rule $l \to r$ is required to additionally satisfy that $\mathcal{FV}(r) \subseteq \mathcal{FV}(l)$ and l is not a free variable.

We use $\to_R$ to denote one step R-rewriting, $\to^*_R$ the reflexive transitive closure and $=_R$ the equivalence of $\to_R$. Let $=_{\beta\eta R}$ denote $(=_R \cup =_{\beta\eta})^*$. Since for all patterns u and v, $u =_{\beta\eta R} v$ if and only if $u{\uparrow_\eta} =_R v{\uparrow_\eta}$

[3], we need only to consider $=_R$ instead of $=_{\beta\eta R}$.

Let $\mathcal{W}$ be a free variable set. We use $\sigma =_R \theta\ [\mathcal{W}]$ to denote $\sigma(X) =_R \theta(X)$ for each $X \in \mathcal{W}$, where $[\mathcal{W}]$ may be omitted if clear from the context. We may write $\sigma \supseteq_R \theta\ [\mathcal{W}]$ if $\rho\sigma =_R \theta\ [\mathcal{W}]$ for some ρ.

A *unification pair* $s =^? t$ is an unordered pair of patterns s and t of the same type. It may be assumed that both sides of a unification pair are always automatically α-converted to have the same sequence of outer λ-binders. *Unification problems* are multisets of unification pairs. Let P, Q range over them. Then we use $\mathcal{U}_R(P)$ to denote the set of all higher-order R-unifiers of P. If $\mathcal{U}_R(P) = \mathcal{U}_R(Q)$, P and Q are said to be *equivalent* w.r.t. R.

4 Higher-order narrowing

In the first-order case, an R-narrowing step on a first-order term s is the combination of guessing a first-order substitution σ_1 for s and applying a rewrite rule $l \to r$ to $\sigma_1(s)$ at a position $p \in \overline{\mathcal{O}}(s)$. Let σ_2 be such that $\sigma_2(l) = \sigma_1(s_{|p})$. Then the result of the R-narrowing step is $\sigma_1(s)[\sigma_2(r)]_p$. It may be assumed that $\mathcal{FV}(l \to r) \cap \mathcal{FV}(s) = \{\}$. Thus $\sigma_1 \cup \sigma_2$ may be chosen as a most general syntactic unifier of $s_{|p}$ and l.

For a rewrite rule $l_1 \to r_1$ and a variable renaming ρ such that $\mathcal{D}(\rho) \supseteq \mathcal{FV}(l_1 \to r_1)$ and $\mathcal{I}(\rho)$ contains only new variables, we call $\rho(l_1) \to \rho(r_1)$ a *variant of* $l_1 \to r_1$. Let s be a first-order term and $R = \{l_1 \to r_1, \cdots, l_m \to r_m\}$ a TRS. If $l \to r$ is a variant of some $l_i \to r_i$, $1 \le i \le m$, and σ a most general syntactic unifier of $s_{|p}$ and l, an R-narrowing step may be written as $s \leadsto_{p,i,\sigma} \sigma(s[r]_p)$. We may write $s \leadsto_\sigma t$ for $s \leadsto_{p,i,\sigma} t$. A narrowing derivation

$$s = s_1 \leadsto_{\sigma_1} s_2 \leadsto_{\sigma_2} \cdots \leadsto_{\sigma_{n-1}} s_n = t$$

may be denoted as $s \leadsto^*_\sigma t$, where $\sigma = \sigma_{n-1} \cdots \sigma_1$. If $n = 1$ then $\sigma = \{\}$.

Regard $=^?$ as a new binary function symbol. Then a unification pair $s =^? t$ is a term with $(s =^? t)_{|1\cdot p} = s_{|p}$ and $(s =^? t)_{|2\cdot p} = t_{|p}$. A narrowing derivation $s =^? t \leadsto^*_\sigma u =^? v$ is called *successful* if there is σ' with $\sigma'(u) = \sigma'(v)$. In this case, $\sigma'\sigma$ is an R-unifier of $s =^? t$.

Let $R = \{0 + X \to X, suc(X) + Y \to suc(X + Y)\}$. Then the unification pair $X + Y =^? suc(Y)$ has only one successful narrowing derivation

$$\begin{array}{ll} & X + Y =^? suc(Y) \\ \leadsto_{1,2,\sigma_1} & suc(X_1 + Y) =^? suc(Y) \\ \leadsto_{1\cdot 1,1,\sigma_2} & suc(Y) =^? suc(Y), \end{array}$$

where $suc(X_1)+Y_1 \to suc(X_1+Y_1)$ is the variant used in $\leadsto_{1,2,\sigma_1}$ and $0 + X_2 \to X_2$ in $\leadsto_{1\cdot 1,1,\sigma_2}$ with $\sigma_1 = \{X \mapsto suc(X_1), Y_1 \mapsto Y\}$ and $\sigma_2 = \{X_1 \mapsto 0, X_2 \mapsto Y\}$. Restricting $\sigma_2\sigma_1$ to $\{X, Y\}$ yields an R-unifier $\{X \mapsto suc(0)\}$.

In the higher-order setting, unification pairs may contain patterns, thus higher-order unification is needed. Roughly speaking, a higher-order narrowing step is a first-order narrowing step where first-order syntactic unification is replaced by higher-order one.

As a preparation, we first explain how to make a rewrite rule applicable in the higher-order setting. The idea is inspired by the notion of "lifting over parameters" in [26].

Definition 4.1 Let $l \to r$ be a rewrite rule with $\mathcal{FV}(l \to r) = \{\overline{X_n}\}$. Let $H_1, \cdots, H_n$ be distinct new free variables, and $y_1, \cdots, y_k$ distinct bound variables. Let u and v be the results of replacing all occurrences of X_i in l and r, resp., by $H_i(\overline{y_k})$. Then $\lambda\overline{y_k}.u \to \lambda\overline{y_k}.v$ is a *variant of* $l \to r$ *over* $\overline{y_k}$. □

Lemma 4.2 *For $l \to r \in R$, if $\lambda\overline{y}.u \to \lambda\overline{y}.v$ is a variant of $l \to r$ over $\overline{y}$, then $\lambda\overline{y}.u \to_R \lambda\overline{y}.v$ holds.*

Let $R = \{0 + X \to X, suc(X) + Y \to suc(X + Y)\}$. Then the unification pair $\lambda y.F(y) + y =^? \lambda y.suc(y)$ has a higher-order narrowing derivation

$$\begin{array}{ll} & \lambda y.F(y) + y =^? \lambda y.suc(y) \\ \leadsto_{1\cdot 1,2,\theta_1} & \lambda y.suc(X_1(y) + y) =^? suc(y) \\ \leadsto_{1\cdot 1\cdot 1,1,\theta_2} & \lambda y.suc(y) =^? \lambda y.suc(y) \end{array}$$

where $\lambda y.suc(X_1(y))+Y_1(y) \to \lambda y.suc(X_1(y)+Y_1(y))$ is the variant used in $\leadsto_{1\cdot 1,2,\theta_1}$ and $\lambda y.0 + X_2(y) \to X_2(y)$ in $\leadsto_{1\cdot 1\cdot 1,1,\theta_2}$ with

$$\begin{array}{lll} \theta_1 & = & \{F \mapsto \lambda z.suc(X_1(z)), Y_1 \mapsto \lambda z.z\} \\ \theta_2 & = & \{X_1 \mapsto \lambda z.0, X_2 \mapsto \lambda z.z\}. \end{array}$$

The resulting R-unifier is $(\theta_2\theta_1)_{|\{F\}} = \{F \mapsto \lambda z.suc(0)\}$. If no lifting over y were made in e.g. the variant in $\leadsto_{1\cdot 1,2,\theta_1}$, then the syntactical unification of the subterm $F(y) + y$ and the left-hand side $suc(X_1)+Y_1$ of the variant would fail, since y cannot occur free in the substitution of Y_1.

Definition 4.3
Suppose $R = \{l_1 \to r_1, \cdots, l_m \to r_m\}$. Let s be a pattern, $p \in \overline{\mathcal{O}}(s)$ and $\lambda x_1, \cdots, \lambda x_k$ all λ-binders in s covering p. Let $\lambda\overline{x_k}.u \to \lambda\overline{x_k}.v$ be a variant of $l_i \to r_i$ over $\overline{x_k}$. A *higher-order R-narrowing step* (short: *R-narrowing step*) is defined as $s \leadsto_{p,i,\theta} \theta(s[v]_p)$, where θ is a most general syntactic unifier of $\lambda\overline{x_k}.s_{|p} =^? \lambda\overline{x_k}.u$ such that $\mathcal{D}(\theta) \subseteq \mathcal{FV}(s, \lambda\overline{x_k}.u)$ and $\mathcal{D}(\theta) \cap \mathcal{I}(\theta) = \{\}$. We may write $s \leadsto_\sigma t$ or $s \leadsto_{p,i} t$ for $s \leadsto_{p,i,\sigma} t$. The notation $s \leadsto^*_\sigma t$ is defined in the same way as in the

first-order case. If $\mathcal{D}(\theta) \cap \mathcal{FV}(s) = \{\}$, we may write $s \rightarrow_{p,i} s[\theta(v)]_p$. □

Lemma 4.4 *Let R be a TRS. Then $s \rightarrow_{p,i} t$ if and only if $s \rightarrow_R t$.*

Lemma 4.5 *If $s \rightsquigarrow_{p,i,\sigma} t$ then $\sigma(s) \rightarrow_{p,i} t$.*

Theorem 4.6 *(Soundness) Let R be a TRS. If a unification pair $s =^? t$ has an R-narrowing derivation $s =^? t \rightsquigarrow^*_\sigma u =^? v$ with σ' satisfying $\sigma'(u) = \sigma'(v)$, then $\sigma'\sigma$ is an R-unifier of $s =^? t$.*

Definition 4.7 Let θ be a substitution. An R-narrowing derivation $s =^? t \rightsquigarrow^*_\sigma u =^? v$ is called *successful (for θ)* if there is σ' satisfying $\sigma'(u) = \sigma'(v)$ and $\sigma'\sigma \supseteq_R \theta[\mathcal{FV}(s,t)]$. □

4.1 Completeness

Usually the completeness of narrowing means that if $s =^? t$ has an R-unifier θ, then there exists an R-narrowing derivation issuing from $s =^? t$ and successful for θ. However, in order to find a successful one, too many R-narrowing derivations may need to be looked at in general. Therefore, restricted TRS's and restricted unification pairs have been considered where special and efficient strategies can be developed for finding successful narrowing derivations (cf. the references of [19]). Since we are only interested in the problems of higher-order extensions, the details of the existing first-order approaches will be avoided whenever possible. A notion of narrowing strategy is then defined in an abstract way.

Definition 4.8 A *higher-order narrowing strategy* (short: *narrowing strategy*) is a function $\mathcal{S}$ which for each TRS R and pattern t yields a set $\mathcal{S}(R,t)$ of R-narrowing derivations issuing from t. □

Many existing first-order narrowing strategies can be naturally extended to the higher-order case. Take basic narrowing as an example. Then higher-order basic narrowing should have the same idea as the firsr-order one ([15]): for a confluent and terminating TRS, a narrowing step need not happen at a subterm introduced by a substitution in a previous narrowing step. The difference is just that higher-order substitutions are used in higher-order basic narrowing. It should not be surprising that the definition of higher-order basic narrowing is almost identical to that in [15].

Definition 4.9 Let $R = \{l_1 \rightarrow r_1, \cdots, l_m \rightarrow r_m\}$ be a TRS. Then *higher-order basic R-narrowing* consists of all R-narrowing derivations of the form

$$t_1 \rightsquigarrow_{p_1,i_1} \cdots \rightsquigarrow_{p_{n-1},i_{n-1}} t_n$$

where $p_j \in B_j$ for $1 \leq j \leq n-1$, and the sets $B_1, \cdots, B_{n-1}$ are inductively defined as

$$B_1 = \overline{\mathcal{O}}(t_1)$$
$$B_{j+1} = (B_j - \{q \mid p_j \leq q\}) \cup \{p_j \cdot q \mid q \in \overline{\mathcal{O}}(r_{i_j})\}, \quad j = 1, \ldots, n-2$$

□

Let us now introduce an abstract notion of completeness. Note that we will only consider normalized R-unifiers, as in most existing work, although TRS's R are not required to be even weakly normalizing in general.

Definition 4.10 Let R be a TRS and $\mathcal{P}$ a class of unification pairs. A narrowing stragety $\mathcal{S}$ is said to be *complete* for R and $\mathcal{P}$ if for every $s =^? t \in \mathcal{P}$ and every normalized R-unifier θ of $s =^? t$, there is an R-narrowing derivation in $\mathcal{S}(R, s =^? t)$ which is successful for θ. □

We will present a result which reduces the completeness of a higher-order narrowing strategy to that of its first-order underlying counterpart. The key is to relate a pattern with a first-order term by viewing λ-binders as new unary free function symbols and flexible subterms as new first-order free variables. For example, $\lambda x.f(F(x))$ should be viewed as $\lambda x(f(X))$ where λx denotes a free function symbol.

When fixing λ-binders as free function symbols, no explicit α-conversion is possible. Therefore patterns should be α-converted beforehand so that no explicit α-conversion is needed any more. For doing this, assume an infinite list of new bound variables for every type. A pattern is said to be *α-converted* if each λ-binder in the pattern always uses in the corresponding list the first bound variable that has not been used by other covering λ-binders. For example, $\lambda y_1.f(\lambda y_2 y_3.y_2, \lambda y_2.y_2)$ is an α-converted form, provided that f is of the type $(\beta \rightarrow \beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta) \rightarrow \gamma$ and $\langle y_1, y_2, y_3, \cdots \rangle$ the list of bound variables for β. Note that a λ-binder may occur more than once in an α-converted form, but only at independent positions. Occurrences of the same λ-binder will be viewed as occurrences of the same function symbol. No explicit α-conversions are needed in proving R-equivalence of α-converted forms (cf. [29] for a proof). It is assumed in the rest of this section that all patterns are α-converted.

For viewing flexible subterms as first-order free variables, we introduce a special mapping.

Definition 4.11 A *weak abstraction* is a large enough injective mapping ϕ which maps a flexible subterm $F(\overline{x})$ into a first-order variable X. If u is

a pattern then $\phi(u)$ denotes the result of replacing each flexible subterm $F(\overline{x})$ in u by $\phi(F(\overline{x}))$. For a substitution θ, define $\phi(\theta)$ to be a substitution such that if $\theta(F) = \lambda\overline{x_k}.t$ then $\phi(\theta)(\phi(F(\overline{y_k}))) = \phi(\{\overline{x_k \mapsto y_k}\}(t))$ for all bound variable sequences $\overline{y_k}$ with each y_i being of the same type as that of x_i. □

Note that if $F \neq G$ or $\overline{y} \neq \overline{z}$ then $\phi(F(\overline{y})) \neq \phi(G(\overline{z}))$. For a pattern u, $\phi(u)$ can be regarded as a first-order term when the types of the symbols in u are forgotten and all λ-binders in u are regarded as free function symbols. For a substitution θ, $\phi(\theta)$ can be regarded as a first-order substitution in a similar way, where the domain $\mathcal{D}(\phi(\theta))$ is restricted to a finite set including only $\phi(F(\overline{y}))$ of those $F(\overline{y})$ which are in consideration. By definition we may directly prove that the weak abstraction mapping can always be moved into a substitution in the following way:

Lemma 4.12 *For a pattern u and substitution θ, $\phi(\theta(u)) = \phi(\theta)(\phi(u))$ always holds.*

Furthermore, the weak abstraction mapping preserves rewriting relations.

Lemma 4.13 *Let R be a TRS and s, t two patterns. Then $s \to_{p,i} t$ if and only if $\phi(s) \to_{p,i} \phi(t)$.*

Proof See Appendix B. □

A rewrite step always transforms a weak abstraction into a weak abstraction.

Lemma 4.14 *Let s be a pattern. If $\phi(s) \to_{p,i} u$ then there exists a pattern t such that $u = \phi(t)$.*

Finally, we formulate how ϕ relates narrowing derivations with their first-order counterparts. Note that derivations (1) and (2) below employ the same rewrite rules at the same positions.

Lemma 4.15 *Let R be a TRS, $s =^? t$ a unification pair and θ a normalized R-unifier of $s =^? t$. Then $\phi(\theta)$ is a normalized R-unifier of $\phi(s) =^? \phi(t)$. Furthermore, if*

$$\phi(s) \stackrel{?}{=} \phi(t) \quad \leadsto_{p_1,i_1} \cdots \leadsto_{p_n,i_n} \quad s' \stackrel{?}{=} t' \qquad (1)$$

is successful for $\phi(\theta)$, then we have

$$s \stackrel{?}{=} t \quad \leadsto_{p_1,i_1,\sigma_1} \cdots \leadsto_{p_n,i_n,\sigma_n} \quad u \stackrel{?}{=} v \qquad (2)$$

for some $\sigma_1, \cdots, \sigma_n$ such that there is σ' with $\sigma'(u) = \sigma'(v)$ and $\sigma'\sigma_n \cdots \sigma_1 \supseteq_R \theta$ $[\mathcal{FV}(s,t)]$.

Proof If θ is a normalized R-unifier of $s =^? t$, then $\theta(s) =_R \theta(t)$, thus $\phi(\theta)$ is a normalized R-unifier of $\phi(s) =^? \phi(t)$ by Lemmas 4.4, 4.12 and 4.13.

The proof of the second claim is based on Lemmas 4.5, 4.14, 4.13 and an additional lemma, which is rather technical. For details see Appendix B. □

Now we may introduce a way to characterize narrowing strategies.

Definition 4.16 A higher-order narrowing stragety $\mathcal{S}$ is said to be a *total higher-order extension* for a TRS R and a class $\mathcal{P}$ of unification pairs if for every $s =^? t \in \mathcal{P}$ and every normalized R-unifier θ of $s =^? t$, whenever $\mathcal{S}(R, \phi(s) =^? \phi(t))$ contains a first-order R-narrowing derivation as derivation (1) successful for $\phi(\theta)$, $\mathcal{S}(R, s =^? t)$ contains a higher-order R-narrowing derivation as derivation (2). □

By Lemma 4.15, the existence of derivation (1) always implies that of derivation (2). Therefore, to see whether a narrowing strategy $\mathcal{S}$ is a total higher-order extension, we need only to check that whenever derivation (1) is in $\mathcal{S}(R, \phi(s) =^? \phi(t))$, derivation (2) is in $\mathcal{S}(R, s =^? t)$.

Higher-order basic narrowing is a total higher-order extension. For, every position of narrowing in derivations (1) and (2) corresponds to a rigid subterm in either the initial $s =^? t$ or the right-hand side of a rewrite rule, and $s =^? t$ and $\phi(s) =^? \phi(t)$ have the same set of positions of rigid subterms.

A narrowing strategy consisting of all those higher-order basic narrowing derivations where the positions of narrowing are always outermost is a total higher-order extension. To see this, we can check that if a position p_i of narrowing in derivation (2) is not outermost, assuming that it is the first one which is not outermost, then the position p_i in derivation (1) cannot be outermost.

Theorem 4.17 *Let R be a TRS and $\mathcal{P}$ a set of unification pairs. A higher-order narrowing strategy $\mathcal{S}$ is complete for R and $\mathcal{P}$ if $\mathcal{S}$ is a total higher-order extension and complete for R and $\phi(\mathcal{P})$.*

Proof Follows from Lemma 4.15 and Definition 4.16. □

As an instance of the above theorem, by the fact that basic narrowing in the first-order setting is complete for all confluent and strongly normalizing TRS's and all first-order unification pairs [15], we know that higher-order basic narrowing is complete for all confluent and strongly normalizing TRS's and all (higher-order) unification pairs.

5 Termination of equational unification

A very important property of an equational unification algorithm is the termination. This section introduces an approach to reduce the termination of a higher-order equational unification algorithm to that

of its first-order counterpart.

Let us first formulate very abstractly the notions of unification process and unification algorithm.

Definition 5.1 An *abstract unification process* is a (finite or infinite) sequence of pairs of unification problems P_i and substitution sets S_i in the form

$$\langle P_1, S_1\rangle \Longrightarrow \langle P_2, S_2\rangle \Longrightarrow \cdots.$$

An *abstract unification algorithm* $\mathcal{UA}$ is a function which for every unification problem P yields a set of abstract unification processes issuing from $\langle P, \{\}\rangle$ such that if P contains only first-order terms then $\mathcal{UA}(P)$ contains only first-order terms and substitutions. An algorithm $\mathcal{UA}$ is called *terminating* if for every P, $\mathcal{UA}(P)$ contains no infinite abstract unification processes. □

To relate a higher-order equational unification algorithm to its first-order counterpart, we view all bound variables of the same type as identical. This implies that all λ-binders with their bound variables being of the same type are viewed as identical and so are all flexible subterms with the same head free variable.

Definition 5.2 Let us associate each type τ with two new function symbols l_τ and o_τ such that all these new function symbols are pairwise distinct. A *strong abstraction* ψ is a mapping which transforms every pattern in a topdown way as follows:

$$\begin{array}{rcl}
\psi(\lambda x.s) & = & l_\tau(\psi(s)) \\
\psi(a(\overline{s_n})) & = & \psi(a)(\overline{\psi(s_n)}) \\
\psi(F(\overline{x_k})) & = & F \\
\psi(x) & = & o_\tau \\
\psi(f) & = & f
\end{array}$$

where x is a bound variable of type τ, a a function symbol or a bound variable, F a free variable and f a function symbol. For a substitution σ, define $\psi(\sigma) = \{F \mapsto \psi(t) \mid F \mapsto \lambda\overline{y}.t \in \sigma\}$. Strong abstractions of sets of patterns or of substitutions are defined componentwise. □

For a pattern t, $\psi(t)$ may be viewed as a first-order term. For example, if x and y are bound variables of type α, and z of type β, then $\psi(\lambda xyz.z(x, F(x,y), F(y,x), y)) = l_\alpha(l_\alpha(l_\beta(o_\beta(o_\alpha, F, F, o_\alpha))))$.

Definition 5.3 An abstract unification algorithm $\mathcal{UA}$ is said to be *first-order embedded* in another abstract unification algorithm $\mathcal{UA}_1$ if for every unification problem P_1,

$$\langle P_1, \{\}\rangle \Longrightarrow \langle P_2, S_2\rangle \Longrightarrow \cdots \qquad \in \mathcal{UA}(P_1)$$

always implies

$$\langle \psi(P_1), \{\}\rangle \Longrightarrow \langle \psi(P_2), \psi(S_2)\rangle \Longrightarrow \cdots \in \mathcal{UA}_1(\psi(P_1)).$$

□

We may now easily prove the following theorem.

Theorem 5.4 *Let $\mathcal{UA}$ be first-order embedded in $\mathcal{UA}_1$. For a unification problem P, if all abstract unification processes in $\mathcal{UA}_1(\psi(P))$ are terminating then so are all those in $\mathcal{UA}(P)$.*

5.1 Termination of higher-order narrowing

The method developed in the above will be used several times in this paper. First of all, let us consider narrowing strategies as equational unification algorithms and see how to reduce the termination of higher-order narrowing strategies to that of their first-order counterparts.

We show first that the most general unifiers are preserved by strong abstractions.

Proposition 5.5 *If θ is a most general syntactic unifier of two patterns s and t then $\psi(\theta)$ is a most general syntactic unifier of $\psi(s)$ and $\psi(t)$.*

The preservation of the most general unifiers can be extended to narrowing derivations.

Proposition 5.6 *Let R be a TRS. If*

$$s_1 \leadsto_{p_1,i_1,\sigma_1} \cdots \leadsto_{p_{j-1},i_{j-1},\sigma_{j-1}} s_j \leadsto_{p_j,i_j,\sigma_j} \cdots$$

is an R-narrowing derivation, then so is

$$\begin{array}{lll}
\psi(s_1) & \leadsto_{p_1,i_1,\psi(\sigma_1)} & \cdots \\
& \leadsto_{p_{j-1},i_{j-1},\psi(\sigma_{j-1})} & \psi(s_j) \\
& \leadsto_{p_j,i_j,\psi(\sigma_j)} & \cdots
\end{array}$$

Let $s_1 =^? t_1$ be a unification pair. Then we may view an infinite R-narrowing derivation

$$\begin{array}{lll}
s_1 =^? t_1 & \leadsto_{p_1,i_1,\sigma_1} & \cdots \\
& \leadsto_{p_{n-1},i_{n-1},\sigma_{n-1}} & s_n =^? t_n \\
& \leadsto_{p_n,i_n,\sigma_n} & \cdots
\end{array}$$

as an abstract unification process

$$\langle s_1 \stackrel{?}{=} t_1, \{\}\rangle \Longrightarrow \cdots \Longrightarrow \langle s_n \stackrel{?}{=} t_n, \sigma_{n-1}\cdots\sigma_1\rangle \Longrightarrow \cdots$$

and a finite R-narrowing derivation

$$s_1 \stackrel{?}{=} t_1 \leadsto_{p_1,i_1,\sigma_1} \cdots \leadsto_{p_{n-1},i_{n-1},\sigma_{n-1}} s_n \stackrel{?}{=} t_n$$

with σ' being a most general syntactic unifier of $s_n =^? t_n$ as

$$\begin{array}{rcl}
\langle s_1 =^? t_1, \{\}\rangle & \Longrightarrow & \cdots \\
& \Longrightarrow & \langle s_n =^? t_n, \sigma_{n-1}\cdots\sigma_1\rangle \\
& \Longrightarrow & \langle \sigma'(s_n) =^? \sigma'(t_n), \sigma'\sigma_{n-1}\cdots\sigma_1\rangle.
\end{array}$$

Propositions 5.5 and 5.6 say that a strong abstraction of an R-narrowing derivation is still an R-narrowing derivation of the same length and may also be considered as an abstract unification process as defined above.

Let $\mathcal{S}$ be a narrowing strategy for some TRS R and some set $\mathcal{P}$ of unification pairs and $\mathcal{S}_1$ a narrowing strategy for R and $\psi(\mathcal{P})$ with $\mathcal{S}$ being first-order embedded in $\mathcal{S}_1$. By Theorem 5.4 we know that for a unification pair $s =^? t \in \mathcal{P}$, if $\mathcal{S}(R, \psi(s) =^? \psi(t))$ is terminating, then so is $\mathcal{S}(R, s =^? t)$.

6 Equational unification of free patterns

In this section we propose an algorithm for equational unification of free patterns. The algorithm is complete for every consistent equational theory. Let E be an arbitrary but fixed consistent equational theory.

Our algorithm is in fact a revision of the one in [22] (see also Appendix A.3). The key of the revision is to leave unification pairs of the form $\lambda\overline{x}.F(\overline{y}) =^? \lambda\overline{x}.F(\overline{z})$ as *constraints* in the unification process. For notational simplicity, the outer λ-binders $\lambda\overline{x}$ may be omitted, since any λ-binders $\lambda\overline{x}$ such that $\{\overline{x}\} \supseteq \{\overline{y}\} \cup \{\overline{z}\}$ can have the same effect. Let C and D range over sets of constraints. A substitution σ is said to *E-satisfy* C if $\sigma(F(\overline{y})) =_E \sigma(F(\overline{z}))$ for every $F(\overline{y}) =^? F(\overline{z}) \in C$.

Suppose that $\sigma = \{F \mapsto \lambda\overline{v}.a(\overline{H_m(\overline{v})})\}$ is a substitution with a being a function symbol or a bound variable and $H_1, \cdots, H_m$ distinct free variables. Let C be a set of constraints. Then we use $constr(\sigma(C))$ to denote

$$\{\overline{H_m(\overline{y}) =^? H_m(\overline{z})} \mid F(\overline{y}) =^? F(\overline{z}) \in C\}$$
$$\cup \{G(\overline{y'}) =^? G(\overline{z'}) \in C \mid G \neq F\}.$$

For example, if $C = \{F(x,z,y) =^? F(y,x,z),\ G(y,z) =^? G(x,z)\}$ and $\sigma = \{F \mapsto \lambda xyz.f(H_1(x,y,z), H_2(x,y,z))\}$, then $constr(\sigma(C)) = \{H_1(x,z,y) =^? H_1(y,x,z),\ H_2(x,z,y) =^? H_2(y,x,z),\ G(y,z) =^? G(x,z)\}$.

A *decorated substitution* is a pair (σ, C), which can be viewed as representing a set of instance substitutions $\mathcal{INS}_E(\sigma, C) = \{\rho\sigma \mid \rho\ E\text{-satisfies}\ C\}$. Let (θ, D) be another decorated substitution. We may write $(\sigma, C) \supseteq_E (\theta, D)$ if $\mathcal{INS}_E(\sigma, C) \supseteq \mathcal{INS}_E(\theta, D)$. Obviously, a single substitution σ can be written as $(\sigma, \{\})$: $(\sigma, \{\}) \supseteq_E (\theta, \{\})$ if and only if $\sigma \supseteq_E \theta$. A decorated substitution (σ, C) is also called an *E-unifier* of a unification problem if so is every $\sigma' \in \mathcal{INS}_E(\sigma, C)$.

Our algorithm is given by four transformation rules on triples of unification problems, substitutions and sets of constraints. The algorithm starts with the triple $\langle P_0, \{\}, \{\}\rangle$ for any unification problem P_0 and terminates with $\langle\{\}, \sigma, C\rangle$ if P_0 is E-unifiable, in which case the (σ, C) is the most general E-unifier of P_0. The four rules are extensions of rules (Bin), (Dec), (FF-1) and (FF-2) in Appendix A.3, resp. We only present two rules here in Figure 1. Other two rules can be obtained in a corresponding way.

Intuitively, rule (Bin') yields a partial solution for the head variable, and rule (FF-2') yields a partial solution with a constraint on the coomon head variable.

Compared with the algorithm in Appendix A.3, we have one more failure case here, i.e. where $a = y_i$ with $z_i \neq z'_i$ for some $F(\overline{z_n}) =^? F(\overline{z'_n}) \in C$ in rule (Bin').

Theorem 6.1 *There are no infinite sequences of transformations by the four rules of our algorithm here. For any equational theory E, a free unification problem P is E-unifiable if and only if $\langle P, \{\}, \{\}\rangle$ can always be transformed into $\langle\{\}, \sigma, C\rangle$, where $(\sigma_{|\mathcal{FV}(P)}, C)$ is the most general E-unifier of P.*

Proof Use the method in Section 5 to prove the termination. First, our algorithm here is first-order embedded the algorithm in Appendix A.3, since some strong abstraction mapping ψ may map each transformation sequence here into a transformation sequence via the rules in Appendix A.3. Since the algorithm in Appendix A.3 is terminating, so is our algorithm. The proof of soundness and completeness is similar to that in [22]. □

7 Combining higher-order equational unification algorithms

In this section we consider informally the higher-order extensions of first-order combination algorithms. Only the termination property of the higher-order extensions is discussed, since experiences in the first-order case show that it may be a hard problem (cf. e.g. [9, 38, 33, 2]). Again, our approach here is to reduce the termination of the higher-order extensions to that of their underlying first-order counterparts.

When a combination algorithm is available, an equational unification algorithm can always be obtained by combining an equational unification for free patterns (as in Section 6) and an equational unification algorithm for so-called *pure patterns*.

Definition 7.1 A pattern is called *pure* (in an equational theory E) if it contains no subterms of the form

$$\langle \{\lambda\overline{x}.F(\overline{y_n}) \stackrel{?}{=} \lambda\overline{x}.a(\overline{t_m})\}@P, \sigma, C\rangle \Longrightarrow \langle \{\lambda\overline{x}.H_m(\overline{y_n}) \stackrel{?}{=} \lambda\overline{x}.t_m\}@\sigma'(P), \sigma'\sigma, constr(\sigma'(C))\rangle \qquad \text{(Bin')}$$

if $F \notin \mathcal{FV}(\overline{t_m})$, $a \in \mathcal{C} \cup \{\overline{y_n}\}$, $a = y_i$ with $1 \leq i \leq n$ implies $z_i = z'_i$ for every $F(\overline{z_n}) =^? F(\overline{z'_n}) \in C$, where $\sigma' = \{F \mapsto \lambda\overline{y_n}.a(\overline{H_m(\overline{y_n})})\}$ with new distinct variables $H_1, \cdots, H_m$.

$$\langle \{F(\overline{y_n}) \stackrel{?}{=} F(\overline{z_n})\}@P, \sigma, C\rangle \Longrightarrow \langle \sigma'(P), \sigma'\sigma, \sigma'(C) \cup \{H(\overline{y_{p_q}}) \stackrel{?}{=} H(\overline{z_{p_q}})\}\rangle \qquad \text{(FF-2')}$$

where $p_i, 1 \leq i \leq q$, are all those from $\{1, \ldots, n\}$ such that $y_{p_i}, z_{p_i} \in \{\overline{y_n}\} \cap \{\overline{z_n}\}$, $\sigma' = \{F \mapsto \lambda\overline{y_n}.H(y_{p_1}, \ldots, y_{p_q})\}$ and H is a new variable.

Figure 1: Equational unification of free patterns

$a(\overline{t_n})$ with $n > 0$ and a being a free function symbol or a bound variable. A substitutions or unification pair is called *pure* if it contains only pure patterns. □

Assume that some equational theories are given, which are disjoint in the sense that no function symbols occur in more than one theory. If a term is pure in none of the equational theories, it can be decomposed into pure ones. For doing this, every atomic symbol in the term should be given an equational theory in the following way: First of all, every function symbol is either free or already belongs to an equational theory. Then bound variables and λ-binders are always treated as free function symbols. For notational simplicity, all free function symbols are assumed to belong to a *trivial equational theory* $E_0 = \{\}$. So free patterns are pure in E_0. Note that E_0 is disjoint to all other equational theories.

Those patterns, which contain function symbols belonging to different equational theories, may be transformed by rule (Abs) formulated as follows:

$$\{s[u] =^? t\} \cup P \Longrightarrow \{s[H(\overline{y_m})]_p =^? t, \lambda\overline{y_m}.H(\overline{y_m}) =^? \lambda\overline{y_m}.u\} \cup P$$

if u is a maximal rigid subterm of a base type in s such that $\mathcal{H}(u)$ and $\mathcal{H}(s)$ or $\mathcal{H}(u)$ and $\mathcal{H}(t)$ belong to different equational theories, where $\lambda y_1, \cdots, \lambda y_m$ are all λ-binders in $s[u]$ covering u.

In order to completely understand the compact form of the above rule, we should mention that, due to the well-typedness of a pattern, a subterm of the form $\lambda x.s$ cannot be directly covered by a function symbol belonging to some given (first-order) equational theory, since $\lambda x.s$ is of a function type.

Repeatedly applying the above rule will eventually yield some unification pairs that are pure in a certain equational theory E. The given equational unification algorithm for the theory E may be then used to unify these pure unification pairs.

For collapse-free regular equational theories such as AC theories, a cycle can be detected in applying rule (Abs) when $t = \lambda\overline{x}.F(\overline{y})$ with $F \in \mathcal{FV}(u)$. A cycle corresponds to a failure of unification. For example, the unification pair $\lambda xy.G(x) + f(F(y)) =^? \lambda xy.F(x)$ with $+$ being an AC and f free function symbol is not AC-unifiable, since F occurs both in the right-hand side at the outermost level and in the subterm $f(F(x))$ of the left-hand side.

A higher-order combination algorithm for collapse-free regular equational theories can in fact be obtained by repeating rule (Abs) with the detection of cycles and the application of given equational unification algorithms just as in the first-order case ([9, 38]). Note that the notion of decorated substitution may have to be used (cf. [30]). But it can be dealt with in a natural way as in Section 6. The higher-order combination algorithm as above is always terminating since the strong abstract mapping as defined in Section 6 maps all its unification processes into unification processes of some underlying first-order combination algorithm in [9, 38] and the first-order combination algorithm is terminating.

We may also build a higher-order combination algorithm for arbitrary equational theories by naturally extending the first-order combination algorithms in [33, 2]. At the moment we see no serious problems in the higher-order extensions. The strong abstraction mapping reduces the termination of the extensions to that of the first-order combination algorithms [33, 2].

8 Conclusion

We have shown in this paper that the notion of higher-order patterns is the key to connect first-order equational and higher-order logic programming. Three essential aspects in building higher-order equational unification have been considered: higher-order narrowing, unification of free patterns in the pres-

ence of equations and combination of higher-order equational unification algorithms for special equational theories. The results cover a large part of the whole spectrum of higher-order equational unification due to their formulations at an abstract level. Only problems of higher-order extensions are considered and details for the existing first-order approaches are avoided whenever possible.

What is missing in this paper is the development of higher-order equational unification algorithms for pure patterns in special equational theories. Combination algorithms may use these special algorithms in building algorithms for arbitrary patterns and for combined theories. Although general equational unification algorithms for arbitrary simply typed λ-terms and arbitrary equational theories have been proposed in [24, 29], where first-order equational unification algorithms are called as parameters, it seems difficult for us to build a similar algorithm just for patterns, which behaves closely to the called first-order unification algorithms. Therefore, equational unification algorithms for pure patterns in special equational theories might have to be developed individually. See [30] for such an algorithm in AC theories.

Another possible future direction would be to consider higher-order conditional narrowing by extending the results in the first-order case (cf. e.g. [14, 16, 18]). From the semantic point of view, model theoretic semantics still need to be studied. The work by Breazu-Tannen and Meyer [5, 3] might be a good starting point.

ACKNOWLEDGEMENT

We sincerely thank Claude Kirchner for helpful discussions and Dale Miller for useful comments on the paper.

References

[1] F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, Oxford, UK, 1993. To appear.

[2] A. Boudet. Combining unification algorithms. *J. Symbolic Computation*, 11, 1992.

[3] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. 3rd IEEE Symp. Logic in Computer Science*, pages 82–90, 1988.

[4] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *Proc. 16th Int. Coll. Automata, Languages and Programming*, pages 137–150. Springer-Verlag LNCS 372, 1988.

[5] V. Breazu-Tannen and A. Meyer. Computable values can be classical. In *Proc. 14th ACM Symp. Principles of Programming Languages*, pages 238–245. ACM, 1987.

[6] D. DeGroot and G. Lindstrom (eds.). *Logic Programming: Relations, Functions and Equations.* Prentice Hall, 1986.

[7] N. Dershowitz and D. Plaisted. Logic programming cum applicative programming. In *Proc. 1985 Symp. on Logic Programming*, pages 54–67. IEEE Comput. Soc. Press, 1985.

[8] D. Dougherty and P. Johann. A combinatory logic approach to higher-order E-unification. In *Proc. 11th Int. Conf. on Automated Deduction*, pages 79–93. Springer-Verlag LNCS 607, 1992.

[9] F. Fages. Associative-commutative unification. In R. Shostak, editor, *Proc. 7th Int. Conf. Automated Deduction.* Springer-Verlag LNCS 170, 1984.

[10] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proc. 1985 Symp. on Logic Programming*, pages 172–184. IEEE Comput. Soc. Press, 1985.

[11] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67:203–260, 1988.

[12] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *J. Computer and System Sciences*, pages 139–185, 1991.

[13] M. Hanus. Compiling logic programs with equality. In *Proc. Int. Workshop on Language Implementation and Logic Programming*, pages 387–401. Springer-Verlag LNCS 456, 1990.

[14] S. Hölldobler. *Foundations of Equational Logic Programming.* Springer-Verlag LNCS 353, 1989.

[15] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proc. 5th Int. Conf. Automated Deduction*, pages 318–334. Springer-Verlag LNCS 87, 1980.

[16] H. Hußmann. Unification in conditional-equational theories. In *Proc. European Conf.*

on Computer Algebras, pages 543–553. Springer-Verlag LNCS 204, 1985.

[17] 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.

[18] S. Kaplan. Simplifying conditional term rewriting systems: Unification, termination and confluence. *J. Symbolic Computation*, 4(3):295–334, 1987.

[19] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. In *Proc. 3rd Int. Conf. on Algebraic and Logic Programming*, pages 244–258. Springer-Verlag LNCS 632, 1992. A long version to appear in the journal of Applicable Algebra in Engineering, Communication and Computing.

[20] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497 – 536, 1991.

[21] J. Moreno-Navarro and M. Rodríguez-Artalejo. BABEL: A functional and logic programming language based on constructor discipline and narrowing. In *Proc. 2th Int. Conf. on Algebraic and Logic Programming*, pages 223–232. Springer-Verlag LNCS 343, 1989.

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

[23] T. Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.

[24] T. Nipkow and Z. Qian. Modular higher-order E-unification. In R. Book, editor, *Proc. 4th Int. Conf. Rewriting Techniques and Applications*, pages 200–214. Springer-Verlag LNCS 488, 1991.

[25] L. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[26] L. Paulson. Introduction to Isabelle. Technical report, University of Cambridge, Computer Laboratory, 1993.

[27] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*, pages 66–78. Cambridge University Press, 1991.

[28] Z. Qian. Linear unification of higher-order patterns. In J.-P. J. M.-C. Gaudel, editor, *Proc. TAPSOFT'93*, pages 391–405. Springer-Verlag LNCS 668, 1993.

[29] Z. Qian and K. Wang. Higher-order E-unification for arbitrary theories. In *Proc. 1992 Joint Int. Conf. and Symp. on Logic Programming*. MIT Press, 1992.

[30] Z. Qian and K. Wang. Modular equational unification of higher-order patterns: The AC case. Technical report, Draft, Universität Bremen, June 1993.

[31] U. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. 1985 Symp. on Logic Programming*, pages 138–151. IEEE Comput. Soc. Press, 1985.

[32] P. Réty, C. Kirchner, H. Kirchner, and P. Lescanne. NARROWER: A new algorithm and its application to logic programming. In *Proc. 1st Int. Conf. Rewriting Techniques and Applications*, pages 141–157. Springer-Verlag LNCS 256, 1985.

[33] M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *J. Symbolic Computation*, 8:51–99, 1989.

[34] W. Snyder. Higher-order E-unification. In *Proc. 10th Int. Conf. Automated Deduction*, pages 573–587. Springer-Verlag LNCS 449, 1990.

[35] M. van Emden and K. Yukawa. Logic programming with equality. *J. Logic Programming*, pages 265–288, 1987.

[36] F. Weber. *Softwareentwicklung mit Logik hoeherer Stufe*. PhD thesis, FZI, Universität Karlsruhe, 1993.

[37] D. Wolfram. Rewriting, and equational unification: the higher-order cases. In R. Book, editor, *Proc. 4th Int. Conf. Rewriting Techniques and Applications*, pages 25–36. Springer-Verlag LNCS 488, 1991.

[38] K. Yelick. Unification in combinations of collapse-free regular theories. *J. Symbolic Computation*, 3:153–181, 1987.

[39] J.-H. You. Unification modulo an equality theory for equational logic programming. *J. Computer and System Sciences*, pages 54–75, 1991.

A Appendix

A.1 The simply typed λ-calculus

A term of the form $(s\ t)$ is called an *application* and $\lambda x.s$ an *abstraction*. The topmost part λx in $\lambda x.t$ is called a λ-*binder* of x. It is assumed that no terms may contain λx more than once, unless stated otherwise. Every symbol in t of $\lambda x.t$ is said to be *covered* by or *in the scope* of λx. In $a(u_1, \cdots, u_n)$ the subterms $u_1, \cdots, u_n$ are called *arguments* of a, and symbols in $u_1, \cdots, u_n$ are said to be *covered* by or *in the scope* of a.

Terms are only compared modulo α-conversion.

The β-normal form s is called *flexible* if $\mathcal{H}(s)$ is a free variable, *rigid* if not.

If t is a pattern and σ a substitution containing only patterns, then $\sigma(t)\downarrow_\beta$ is a pattern. In the sequel, all terms are patterns unless stated otherwise.

Positions are strings of numbers, denoted by p, q, and concatenated by ".". The empty string is denoted by ε. *Subterms* of a pattern may be numbered by positions such that (i) $t_{|\varepsilon} = t$, (ii) $\lambda x.t_{|1\cdot p} = t_{|p}$ and (iii) $(a(\overline{t_k}))_{|i\cdot p} = (t_i)_{|p}$ for $1 \leq i \leq k$. We write $p \leq q$ if there is p' such that $p \cdot p' = q$. If neither $p \leq q$ nor $q \leq p$, we say that p and q are independent.

A.2 Equational theories

An *algebraic term* (or *first-order term*) is a pattern of a base type, which is either a free variable or of the form $f(\overline{s_n})$ with $n \geq 0$, f being a function symbol and each s_i an algebraic term.

The *R-rewriting* $\rightarrow_R$ is the smallest relation such that (i) $\sigma(l) \rightarrow_R \sigma(r)$ for all substitutions σ and all $l \rightarrow r \in R$, and (ii) $s \rightarrow_R t$ implies $(u\ s) \rightarrow_R (u\ t)$, $(s\ u) \rightarrow_R (t\ u)$ and $\lambda x.s \rightarrow_R \lambda x.t$. For example, $\lambda y.f(y) \rightarrow_R \lambda y.g(y)$ if $f(X) \rightarrow g(X) \in R$.

A TRS R is called *consistent* if $X \neq_R Y$ does not hold for distinct free variables X and Y. A TRS R is called *confluent* if for every $u \rightarrow_R^* s$ and $u \rightarrow_R^* t$ there is a term v with $s \rightarrow_R^* v$ and $t \rightarrow_R^* v$. For a confluent R, we may write $\rightarrow_R^* {}_R^* \leftarrow$ for $=_R$. A pattern s is R-normal if there is no pattern t with $s \rightarrow_R t$. A pattern s is R-normalizable if there is a R-normal pattern t with $s \rightarrow_R^* t$. A substitution σ is R-normal or R-normalizable if so is $\sigma(X)$ for all $X \in \mathcal{D}(\sigma)$. A TRS R is *weakly normalizing* if every pattern is R-normalizable. A TRS R is *strongly normalizing* if there are no infinite rewriting derivations $t_1 \rightarrow_R t_2 \rightarrow_R \cdots$.

An equational theory E can always be considered as a confluent TRS $\{l \rightarrow r, r \rightarrow l \mid l \simeq r \in E\}$. Furthermore, E is called *regular* if each $l \simeq r \in E$ satisfies $\mathcal{FV}(l) = \mathcal{FV}(r)$, E is called *collapse-free* if no equation $l \simeq r \in E$ satisfies $l \in \mathcal{V}$ and $r \notin \mathcal{V}$.

A substitution σ is called *idempotent* if $\sigma\sigma =_R \sigma$. A substitution is idempotent whenever $\mathcal{D}(\sigma) \cap \mathcal{I}(\sigma) = \emptyset$. Let $\mathcal{W}$ be a set of free variables such that there are still infinite many free variables not in $\mathcal{W}$. If $\mathcal{W} \supseteq \mathcal{D}(\sigma)$ then there always exists σ' such that $\mathcal{D}(\sigma') \cap \mathcal{I}(\sigma') = \emptyset$, $\mathcal{D}(\sigma) = \mathcal{D}(\sigma')$, $\sigma \supseteq_R \sigma'$ $[\mathcal{W}]$ and $\sigma' \supseteq_R \sigma$ $[\mathcal{W}]$. Note that σ' is idempotent. It suffices to consider the substitutions like σ'.

An R-unifier of a unification problem P is a substitution θ such that $\theta(s)\downarrow_\beta =_R \theta(t)\downarrow_\beta$ for each $s =^? t \in P$. Let $\mathcal{W}$ be a set of free variables such that there are still infinite many free variables not in $\mathcal{W}$. A set U of R-unifiers of P is said to be *complete* w.r.t. $\mathcal{W}$ if for every R-unifier θ there is $\sigma \in U$ such that $\sigma \supseteq_R \theta$ $[\mathcal{W}]$. We may write such U as $\mathcal{CSU}_R(P)[\mathcal{W}]$. It is always assumed that for each $\sigma \in \mathcal{CSU}_R(P)[\mathcal{W}]$, $\mathcal{D}(\sigma) \subseteq \mathcal{FV}(P)$ and $\mathcal{I}(\sigma) \cap (\mathcal{D}(\sigma) \cup \mathcal{W}) = \emptyset$ where $\mathcal{W}$ contains all free variables that have been used before, i.e. the free variables in $\mathcal{I}(\sigma)$ are always fresh variables.

A.3 Syntactic unification of patterns

Miller was the first to present an algorithm for syntactic unification of patterns [20]. Here we slightly revise the Nipkow's algorithm [22] and give an algorithm consisting of four transformation rules on pairs of substitutions and unification problems. The rules are shown in Figure 2, where unification problems are viewed as lists instead of multisets of unification pairs and @ denotes the concatenation operation of lists. The transformation starts with the pair $\langle P, \{\}\rangle$ for any unification problem P and terminates with $\langle\{\}, \sigma\rangle$ if P is unifiable, in which case σ is the most general unifier of P.

Intuitively, rule (Bin) finds a partial binding for a head variable, rule (Dec) breaks a unification pair into simpler ones, rule (FF-1) finds a unifier of two flexible patterns with distinct heads, and rule (FF-2) finds a unifier of two flexible patterns with the same head.

Inversing the preconditions to the rules in Figure 2 yields the following failure cases. The first case, called *clash*, is the case $a \neq b$ in rule (Dec). The second case, called *cycle*, is the case $F \in \mathcal{FV}(\overline{t_m})$ in rule

$$\langle \{\lambda\overline{x}.F(\overline{y}) \stackrel{?}{=} \lambda\overline{x}.a(\overline{t_m})\} @ P, \sigma\rangle \Longrightarrow \langle \overline{\{\lambda\overline{x}.H_m(\overline{y}) \stackrel{?}{=} \lambda\overline{x}.t_m\}} @ \sigma'(P), \sigma'\sigma\rangle \quad \text{(Bin)}$$

if $F \notin \mathcal{FV}(\overline{t_m})$ and $a \in \mathcal{C} \cup \{\overline{y}\}$ where $H_1, \cdots, H_m$ are new variables and $\sigma' = \{F \mapsto \lambda\overline{y}.a(\overline{H_m(\overline{y})})\}$.

$$\langle \{\lambda\overline{x}.a(\overline{s_n}) \stackrel{?}{=} \lambda\overline{x}.b(\overline{t_m})\} @ P, \sigma\rangle \Longrightarrow \langle \overline{\{\lambda\overline{x}.s_n \stackrel{?}{=} \lambda\overline{x}.t_n\}} @ P, \sigma\rangle \quad \text{(Dec)}$$

if $a, b \in \mathcal{C} \cup \{\overline{x}\}$, $a = b$ (and thus $n = m$).

$$\langle \{\lambda\overline{x}.F(\overline{y}) \stackrel{?}{=} \lambda\overline{x}.G(\overline{z})\} @ P, \sigma\rangle \Longrightarrow \langle \sigma'(P), \sigma'\sigma\rangle \quad \text{(FF-1)}$$

if F and G are distinct free variables, where $\sigma' = \{F \mapsto \lambda\overline{y}.H(\overline{v}), G \mapsto \lambda\overline{z}.H(\overline{v})\}$, $\{\overline{v}\} = \{\overline{y}\} \cap \{\overline{z}\}$ and H is a new free variable.

$$\langle \{\lambda\overline{x}.F(\overline{y_n}) \stackrel{?}{=} \lambda\overline{x}.F(\overline{z_n})\} @ P, \sigma\rangle \Longrightarrow \langle \sigma'(P), \sigma'\sigma\rangle \quad \text{(FF-2)}$$

where $\sigma' = \{F \mapsto \lambda\overline{y_n}.H(\overline{v})\}$, $\{\overline{v}\} = \{y_i \mid y_i = z_i, 1 \leq i \leq n\}$ and H is a new free variable.

Figure 2: Unification of patterns

(Bin). The third case, called *bound variable capture*, is the case $a \in \{\overline{x}\} - \{\overline{y}\}$ in rule (Bin).

Theorem A.1 *There are no infinite sequences of transformations by the rules in Figure 2. A unification problem P is syntactically unifiable if and only if every sequence of transformations starting with $\langle P, \{\}\rangle$ terminates with $\langle \{\}, \sigma\rangle$, in which case $\sigma_{|\mathcal{FV}(P)}$ is a most general syntactic unifier of P.*

B Appendix

Proof of Lemma 4.13.

$\Rightarrow$: Assume $s \rightarrow_{p,i,\theta} t$. Then there is a variant $\lambda\overline{x_k}.u \rightarrow \lambda\overline{x_k}.v$ such that $\theta(\lambda\overline{x_k}.u) = \lambda\overline{x_k}.s_{|p}$ and $t = s[\theta(v)]_p$. By Lemma 4.12, $\phi(\theta)(\phi(u)) = \phi(s_{|p})$ and $\phi(t) = \phi(s)[\phi(\theta)(\phi(v))]_p$. Since $\lambda\overline{x_k}.u = \lambda\overline{x_k}.v$ is a variant of a rewrite rule in R over $\overline{x_k}$, $\phi(u) \rightarrow \phi(v)$ is a variant of the same rule. Hence $\phi(s) \rightarrow_{p,i,\phi(\theta)} \phi(t)$.

$\Leftarrow$: Assume that $\phi(s) \rightarrow_{p,i,\sigma} \phi(t)$. Then there is a variant $l \rightarrow r$ in the first-order case such that $\sigma(l) = \phi(s)_{|p}$ and $\phi(t) = \phi(s)[\sigma(r)]_p$. Assume $\mathcal{D}(\sigma) \subseteq \mathcal{FV}(l)$ and $\mathcal{I}(\sigma) \subseteq \mathcal{FV}(\phi(s))$. Let $\lambda y_1, \cdots, \lambda y_k$ be all λ-binders in $\phi(s)$ covering the position p. Let $\mathcal{FV}(l \rightarrow r) = \{\overline{X_n}\}$. Without loss of generality, we may assume that $l \rightarrow r$ is so chosen that for each variable X_i, $\phi^{-1}(X_i)$ is of the form $H_i(\overline{y_k})$. Construct a variant $\lambda\overline{y_k}.u \rightarrow \lambda\overline{y_k}.v$ from $l \rightarrow r$ as required in Definition 4.1, where each X_i is replaced by $H_i(\overline{y_k})$. Construct a substitution $\theta = \{H_i \mapsto \lambda\overline{y_k}.\phi^{-1}(\sigma(X_i)) \mid 1 \leq i \leq n\}$. Then $\theta(u) = \phi^{-1}(\sigma(l)) = \phi^{-1}(\phi(s)_{|p}) = s_{|p}$ and $\theta(v) = \phi^{-1}(\sigma(r)) = \phi^{-1}(\phi(t)_p) = t_p$. Hence $s \rightarrow_{p,i,\theta} s[\theta(v)]_p = s[t_p]_p = t$. □

To prove the second claim of Lemma 4.15, we need the following lemma, which lifts the relationship between a rewrite and narrowing step in the first-order case to the higher-order setting. Indeed, the following lemma is a natural higher-order extension of the one in [19] except a subtle change in the requirement (v). The change is because the lemma in [19] is based on the fact that two unifiable first-order terms s and t that contain no common free variables always have a most general syntactic unifier σ with $\mathcal{D}(\sigma) \cap \mathcal{I}(\sigma) = \{\}$ and $\mathcal{D}(\sigma) \cup \mathcal{I}(\sigma) = \mathcal{FV}(s, t)$, while an analogous fact does not hold in the higher-order setting. A most general syntactic unifier in the higher-order setting may have to introduce new free variables, even in the case when the patterns to be unified contain no common free variables [20, 22].

Lemma B.1 *Let R be a TRS. Suppose we have a pattern s, a normalized substitution θ and a set $\mathcal{W}$ of free variables such that $\mathcal{FV}(s) \cup \mathcal{D}(\theta) \subseteq \mathcal{W}$. If $\theta(s) \rightarrow_{p,i} t'$ holds, then there exist a pattern s' and substitutions σ, θ' such that*
(i) $s \rightsquigarrow_{p,i,\sigma} s'$,
(ii) $\theta'(s') = t'$,
(iii) $\theta'\sigma = \theta[\mathcal{W}]$,
(iv) θ' is normalized and
(v) $\mathcal{FV}(s') \cup \mathcal{D}(\theta') \subseteq \mathcal{W} - \mathcal{D}(\sigma) \cup \mathcal{I}(\sigma_{|\mathcal{W}})$.

Proof Similar to that in [19]. □

Now we continue to prove Lemma 4.15.

Proof of 4.15 (continued): By using an induction on n in each line, we may prove as follows:

Derivation (1)
$\Rightarrow$ $\phi(\theta)(\phi(s)) =^? \phi(\theta)(\phi(t)) \rightarrow_{p_1,i_1} \cdots \rightarrow_{p_n,i_n} r =^? r$
by Lemma 4.5
$\Rightarrow$ $\phi(\theta)(\phi(s)) =^? \phi(\theta)(\phi(t))$
$\rightarrow_{p_1,i_1} \cdots \rightarrow_{p_n,i_n} \phi(s') =^? \phi(s')$
by Lemma 4.14
$\Rightarrow$ $\theta(s) =^? \theta(t) \rightarrow_{p_1,i_1} \cdots \rightarrow_{p_n,i_n} v' =^? v'$
by Lemma 4.13
$\Rightarrow$ Derivation (2) for some σ' with
$\sigma'(u) = \sigma'(v) \ \wedge \ \sigma'\sigma_n \cdots \sigma_1 \supseteq_R \theta \ [\mathcal{FV}(s,t)]$
by Lemma B.1

□