TOBIAS NIPKOW AND CHRISTIAN PREHOFER

HIGHER-ORDER REWRITING AND EQUATIONAL REASONING

1. Introduction

In 1972, Knuth and Bendix published their seminal paper (Knuth and Bendix, 1970) which introduced the notion of a critical pair and showed that confluence of terminating term-rewriting systems is decidable. This was the birth of what is now known as *term rewriting*. Much effort has since gone into the study of term rewriting systems and as a result there is a large body of knowledge about their properties. The study of term rewriting proceeded in parallel and mostly independently of research in λ-calculus. A first attempt to bridge this gap is due to Klop (Klop, 1980), but it was not until the early nineties that a mixture of typed λ-calculus and term rewriting was proposed (Nipkow, 1991; Wolfram, 1993), called *higher-order rewrite systems*. They allow higher-order functions and their rules are pairs of simply typed λ-terms (as opposed to first-order terms).

As an example of higher-order rewriting consider symbolic differentiation. The function $\mathit{diff}(F, X)$, defined by

$$\begin{array}{rcl} \mathit{diff}(\lambda y.F, X) & \to & 0 \\ \mathit{diff}(\lambda y.y, X) & \to & 1 \\ \mathit{diff}(\lambda y.\mathit{sin}(F(y)), X) & \to & \mathit{cos}(F(X)) * \mathit{diff}(\lambda y.F(y), X) \\ \mathit{diff}(\lambda y.\mathit{ln}(F(y)), X) & \to & \mathit{diff}(\lambda y.F(y), X)/F(X), \end{array}$$

computes the differential of a function F at a point X. This set of rewrite rules is shown to be confluent and terminating in (Prehofer, 1995c). With these rules, we can for instance reduce a term as follows:

$$\mathit{diff}(\lambda y.\mathit{sin}(\mathit{sin}(y)), X) \xrightarrow{*} \mathit{cos}(\mathit{sin}(X)) * \mathit{cos}(X)$$

It is well-known that λ-abstraction is general enough to represent quantification in formulae, abstraction in functional programs, and many other variable-binding constructs (Pfenning and Elliott, 1988). Using this representation, many operations on formulae and programs can be expressed naturally as higher-order rewrite systems.

The objective of this chapter is to survey the results which have been generalized from first-order to higher-order rewrite systems. The aim of this generalization is to lift the rich theory developed around first-order rewrite systems and apply it to systems manipulating higher-order terms, such as program transformers, theorem provers and the like. We will consider the basic properties and techniques and focus on confluence results and equational reasoning.

The contents of the chapter is as follows. In Section 2 we review the terminology and notation of the typed λ-calculus which is used to define object-level rewrite systems, and define some basic properties. In Section 3 we define Higher-Order Rewrite Systems (HRSs), Pattern Rewrite Systems (PRSs) and the induced reduction relation on terms, and show how the latter interacts with substitutions. Then we show how HRSs induce an equality on terms and how it relates to reduction. In Section 4 the Critical Pair Lemma due to Knuth and Bendix is generalized to PRSs. This is extended to confluence modulo equality in Section 5. In Section 6 we deal with Orthogonal Pattern Rewrite Systems (OPRSs). These are a special kind of PRSs that have no critical pairs and whose rewrite rules are all left-linear. Section 7 briefly surveys the literature on termination of HRSs, which is still in its infancy. In Section 8 we introduce a unification algorithm for a restricted class of λ-terms, called higher-order patterns, followed by an algorithm for the full higher-order case in Section 9. Then we focus in Section 10 on calculi for solving equational goals modulo a higher-order rewrite system and show some refinements in Section 11. This chapter combines and summarizes several earlier papers of the authors (Nipkow, 1991; Nipkow, 1993b; Nipkow, 1993a; Mayr and Nipkow, 1998; Prehofer, 1994b; Prehofer, 1995b), where all proofs can be found.

2. Preliminaries

What follows is a description of the meta-language of simply typed λ-calculus which is used to define object-level rewrite systems. The notation is roughly consistent with the standard literature (Barendregt, 1992; Hindley and Seldin, 1986).

Starting with some fixed set of **base types** $\mathcal{B}$ the set of all **types** $\mathcal{T}$ is the closure of $\mathcal{B}$ under the function space constructor $\rightarrow$. The letter τ is used to denote types. Function types associate to the right: $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ means $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. Instead of $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ we also write $\overline{\tau_n} \rightarrow \tau$, if τ is a base type.

Terms are generated from a set of typed **variables** $V = \bigcup_{\tau\in\mathcal{T}} V_\tau$ and a set of typed **constants** $C = \bigcup_{\tau\in\mathcal{T}} C_\tau$, where $V_\tau \cap V_{\tau'} = C_\tau \cap C_{\tau'} = \emptyset$ if $\tau \neq \tau'$, by λ-abstraction and application. Arbitrary variables are denoted by x, y and z, free variables by upper case letters F, G and H, and constants by c, d, f and g. **Atoms** are constants or variables and are denoted by a and b. Terms are denoted by l, r, s, t and u. We write $t : \tau$ to indicate that a term t is of type τ. The inductive definition of **simply typed** λ-terms is as follows:

$$\frac{x \in V_\tau}{x : \tau} \quad \frac{c \in C_\tau}{c : \tau} \quad \frac{s : \tau \to \tau' \quad t : \tau}{(s\ t) : \tau'} \quad \frac{x : \tau \quad s : \tau'}{(\lambda x.s) : \tau \to \tau'}$$

In the sequel, all our λ-terms are assumed to be simply typed.

Instead of $\lambda x_1 \ldots \lambda x_n.s$ we also write $\lambda x_1, \ldots, x_n.s$ or just $\lambda\overline{x_n}.s$, where the x_i are assumed to be distinct. Instead of $(\ldots(t\ u_1)\ldots)u_n$ we write $t(u_1, \ldots, u_n)$ or just $t(\overline{u_n})$. The notation $t(\overline{u_n})$ includes the possibility $n = 0$ if t is of base type. The **free** and **bound** variables occurring in a term s are denoted by $fv(s)$ and $bv(s)$, respectively. A term is called **linear** iff no free variable occurs in it more than once.

We assume the usual definition of α, β and η conversion between λ-terms. We write $s =_\gamma t$, where $\gamma \in \{\alpha, \beta, \eta\}$ if s and t are equivalent modulo γ-conversion. This means, for example, $\lambda x.(x\ y) =_\alpha \lambda z.(z\ y)$ but $\lambda x.(x\ y) \neq_\alpha \lambda y.(y\ y)$, $(\lambda x.x)t =_\beta t$ and $\lambda x.(y\ x) =_\eta y$. In the sequel, α-equivalent terms are identified.

As the simply typed λ-calculus is confluent and terminating w.r.t. β-reduction (η-reduction) every term t has a β-normal form (η-normal form) which is denoted $t{\downarrow_\beta}$ ($t{\downarrow_\eta}$). Let t be in β-normal form. Then t is of the form $\lambda\overline{x_k}.a(\overline{u_m})$, where a is called the **head** of t. The **η-expanded** form of t is defined by

$$t{\uparrow^\eta} = \lambda\overline{x_{n+k}}.a(\overline{u_m}{\uparrow^\eta}, x_{n+1}{\uparrow^\eta}, \ldots, x_{n+k}{\uparrow^\eta})$$

where $t : \overline{\tau_{n+k}} \to \tau$ and $x_{n+1}, \ldots, x_{n+k} \notin fv(\overline{u_m})$. Instead of $t{\downarrow_\beta}{\uparrow^\eta}$ we write $t{\updownarrow^\eta_\beta}$. A λ-term t is in **long $\beta\eta$-normal form** iff $t = t{\updownarrow^\eta_\beta}$.

Convention: Unless stated otherwise, the variables r, s, t, etc., range over λ-terms in long $\beta\eta$-normal form.

Terms can also be viewed as trees. Subterms can be numbered by so-called **positions** which are the paths from the root to the subterm in Dewey decimal notation. Details can be found in (Huet, 1980; Dershowitz and Jouannaud, 1990). We just briefly review the notation. The **positions** in a term t are denoted by $\mathcal{P}os(t) \subseteq \mathbb{N}^*$. The letters p and

q stand for positions. The root position is ε, the empty sequence. Two positions p and q are appended by juxtaposing them: pq. The letter i is reserved for natural numbers and $i.p$ is the position obtained by appending i to the front of p. Given $p \in \mathcal{P}os(t)$, t/p is the subterm of t at position p; $t[u]_p$ is t with t/p replaced by u. Note that there are slightly different approaches to positions in λ-terms, as discussed in (Prehofer, 1995c). As we do not give proofs, this issue is not elaborated here.

Substitutions are finite mappings from variables to terms of the same type. Substitutions are denoted by θ, σ and δ. For $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ we define $\mathcal{D}om(\theta) = \{x_1, \ldots, x_n\}$ and $\mathcal{C}od(\theta) = \{t_1, \ldots, t_n\}$. The application of a substitution θ to a term is defined by $\theta(t) = (\lambda\overline{x_n}.t)(\overline{t_n})\updownarrow_\beta^\eta$. We often drop the parentheses and simply write θt.

Let θ_1 and θ_2 be substitutions. Then $\theta_1 + \theta_2$ is a substitution with

$$\mathcal{D}om(\theta_1 + \theta_2) = \mathcal{D}om(\theta_1) \cup \mathcal{D}om(\theta_2)$$

defined by

$$(\theta_1 + \theta_2)(F) = \begin{cases} \theta_2(F) & \text{if } F \in \mathcal{D}om(\theta_2) \\ \theta_1(F) & \text{otherwise} \end{cases}$$

A **renaming** ρ is an injective substitution with $\mathcal{C}od(\rho) \subset V$ and $\mathcal{D}om(\rho) \cap \mathcal{C}od(\rho) = \emptyset$. Renamings are always denoted by ρ.

Two **substitutions are equal on a set of variables** W, written as $\theta =_W \theta'$, if $\theta F = \theta' F$ for all $F \in W$. A substitution θ is **idempotent** iff $\theta = \theta\theta$. We will in general assume that substitutions are idempotent. A substitution θ' is more general than θ, written as $\theta' \leq \theta$, if $\theta = \sigma\theta'$ for some substitution σ.

Note that we will always assume that the domain of a substitution does not contain any variable bound in a term the substitution is applied to. If necessary, the bound variables are renamed automatically.

Two terms s and t are called **unifiable** iff there is a substitution θ, such that $\theta(s) = \theta(t)$. The term s **matches** the term t iff there is a substitution θ, such that $\theta(s) = t$. The problem to decide if a term s matches a term t and to compute the substitution θ is called the **matching problem**, which is very important for rewriting (see the remarks after Definition 3.2).

Given $p \in \mathcal{P}os(t)$, $bv(t, p)$ is the list of all λ-abstracted variables on the path from the root of t to p:

$$\begin{aligned} bv(t, \varepsilon) &= [] \\ bv((t_1\ t_2), i.p) &= bv(t_i, p) \\ bv(\lambda x.t, 1.p) &= x.bv(t, p) \end{aligned}$$

It is frequently necessary to "lift" a term into a context of certain bound variables. An $\overline{x_k}$**-lifter** of a term t **away from a set of variables** W is a substitution $\sigma = \{F \mapsto (\rho F)(\overline{x_k}) \mid F \in fv(t)\}$ where ρ is a renaming such that $\mathcal{D}om(\rho) = fv(t)$, $\mathcal{C}od(\rho) \cap W = \emptyset$ and $\rho F : \tau_1 \to \cdots \to \tau_k \to \tau$ if $x_1 : \tau_1, \ldots, x_k : \tau_k$ and $F : \tau$. For example $\sigma = \{F \mapsto G(x), S \mapsto T(x)\}$ is an x-lifter of $f(\lambda y.g(F(y)), S)$ away from any W not containing G or T; the corresponding renaming is $\rho = \{F \mapsto G, S \mapsto T\}$.

DEFINITION 2.1. We define the **order** of a type in the traditional way:

$$\begin{array}{rcl} ord(\tau) & = & 1 \qquad \text{if } \tau \text{ is a base type} \\ ord(\tau_0 \to \tau_1) & = & max\{ord(\tau_0) + 1, ord(\tau_1)\} \end{array}$$

The **order** of a substitution is the maximal order of the types of the variables in its domain:

$$ord(\theta) = max\{ord(\tau) \mid V_\tau \cap \mathcal{D}om(\theta) \neq \emptyset\}$$

Note that $ord(\emptyset)$, i.e. $max(\emptyset)$ is defined to be 0.

Given a relation $\to$, $\stackrel{=}{\to}$ denotes the reflexive closure, $\leftrightarrow$ the symmetric closure and $\stackrel{*}{\to}$ the transitive reflexive closure of $\to$. We write $s \downarrow t$ iff there is a u, such that $s \stackrel{*}{\to} u$ and $t \stackrel{*}{\to} u$. The relation $\to$ is **(locally) confluent** if $r \stackrel{*}{\to} s$ $(r \to s)$ and $r \stackrel{*}{\to} t$ $(r \to t)$ imply $s \downarrow t$. The relation $\to$ is **terminating** if there is no infinite sequence $s_i \to s_{i+1}$ for all $i \in \mathbb{N}$. It is well known that terminating relations are confluent iff they are locally confluent (Huet, 1980).

3. Higher-Order Rewrite Systems

Higher-Order Rewrite Systems are generalizations of first-order rewrite systems (Baader and Nipkow, 1998; Dershowitz and Jouannaud, 1990) to terms with higher-order functions and bound variables. Since unifiability of λ-terms is undecidable in general (Goldfarb, 1981), we often restrict to a certain subclass of λ-terms which behave very much like first-order terms w.r.t. unification.

DEFINITION 3.1. A λ-term t in β-normal form is called 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.

Some examples of higher-order patterns are $\lambda x.c(x)$, F, $\lambda x.F(\lambda z.x(z))$ and $\lambda x, y.F(y, x)$. Non-patterns are $F(c)$, $\lambda x.F(x, x)$, $\lambda x, y.F(y, c)$ and $\lambda x.G(H(x))$.

The following crucial result about unification of patterns is due to Miller (Miller, 1991a): it is decidable whether two patterns are unifiable; if they are unifiable, a most general unifier can be computed. This result is discussed in more detail in Section 8. Qian (Qian, 1996) shows that patterns can be unified in linear time.

DEFINITION 3.2. A **rewrite rule** is a pair $l \to r$ such that l is not a free variable, l and r are of the same *base* type, and $fv(l) \supseteq fv(r)$. A **pattern rewrite rule** is a rewrite rule whose left-hand side is a pattern. A **Higher-Order Rewrite System** (**HRS**) is a set of rewrite rules. A **Pattern Rewrite System** (**PRS**) is a set of pattern rewrite rules. The letter R always denotes an HRS (which will often be a PRS). An HRS R induces a relation $\to_R$ on terms:

$$s \underset{R}{\to} t \quad \Leftrightarrow \quad \exists (l \to r) \in R, p \in \mathcal{P}os(s), \theta.\ s/p = \theta l \ \wedge\ t = s[\theta r]_p.$$

A rewrite rule is called **left-linear** iff its left-hand side is linear. A HRS is called **left-linear** iff all its rewrite rules are left-linear.

Our HRSs are Wolfram's "higher-order term rewriting systems" (Wolfram, 1993).

A few remarks are in order:

- Recall that by convention l, r, s and t are in long $\beta\eta$-normal form.
- Due to the restrictions placed on left-hand sides, they must always be of the form $f(\overline{s_n})$.
- For PRSs, the restriction $fv(l) \supseteq fv(r)$ is preserved under substitution. Thus it has the same effect as for TRSs: rewriting cannot introduce new variables. This fails for general HRSs: although $f(F(X)) \to g(X)$ meets the restriction, replacing F by $\lambda x.Y$ yields $f(Y) \to g(X)$ which violates the restriction. This problem is investigated in more detail by Kahrs (Kahrs, 1995b).

  Since we derive only very basic results about HRSs, we do not need to impose anything like $fv(l) \supseteq fv(r)$, whereas the latter is essential for the confluence results for PRSs.

- The restriction to rules of base type is necessary because of the simple matching procedure inherent in the definition of $\rightarrow_R$. Otherwise $\overset{*}{\leftrightarrow}_R$ and $=_R$ do not coincide (see Section 3.1). Pulling rules down to base type by applying them to new variables is fine for HRSs but may fail for PRSs: $\lambda x.c(F(x))$ is a pattern but ceases to be one when pulled down to base type: $c(F(X))$.
- $\rightarrow_R$ is defined only between terms in long $\beta\eta$-normal form. This simplifies technicalities. Alternatively we could work with a relation $\rightsquigarrow_R$ defined by $s \rightsquigarrow_R t :\Leftrightarrow s\updownarrow_\beta^\eta \rightarrow_R t\updownarrow_\beta^\eta$.
- The relation $\rightarrow_R$ is decidable if the matching problem is decidable for the left-hand sides of the rules in R. As unifiability is decidable for patterns, the relation $\rightarrow_R$ is decidable for any PRS R; full unification will come in handy in connection with critical pairs.

  For full HRSs the situation is different. Although unifiability is known to be undecidable even for second-order λ-terms (Goldfarb, 1981), it is not known whether higher-order matching is decidable. Padovani (Padovani, 1995) showed that 4th order matching is decidable, but the general case is still open.

EXAMPLE 3.3. The standard example of a PRS is pure lambda-calculus itself. The syntax involves just the type *term* of terms and two constants for abstraction and application:

$$\begin{array}{rl} abs: & (term \rightarrow term) \rightarrow term \\ app: & term \rightarrow term \rightarrow term \end{array}$$

The rewrite rules are:

$$\begin{array}{rrcl} beta: & app(abs(\lambda x.F(x)), S) & \rightarrow & F(S) \\ eta: & abs(\lambda x.app(S, x)) & \rightarrow & S \end{array}$$

Note how the use of meta-level application and abstraction removes the need for a substitution operator (in the *beta*-rule) and side conditions (in the *eta*-rule).

In the sequel, it will be convenient to have an inference-rule based formulation of rewriting.

DEFINITION 3.4. Given an HRS R, let $\Rightarrow_R$ be the least relation on terms in long $\beta\eta$-normal form which is closed under the following rules:

$$\frac{(l \rightarrow r) \in R}{\theta l \Rightarrow_R \theta r} \qquad \frac{s \Rightarrow_R t}{a(\overline{s_m}, s, \overline{u_n}) \Rightarrow_R a(\overline{s_m}, t, \overline{u_n})} \qquad \frac{s \Rightarrow_R t}{\lambda x.s \Rightarrow_R \lambda x.t}$$

where a is an atom of type $\overline{\tau_{m+1+n}} \to \tau$.

Of course the two definitions of rewriting are equivalent.

LEMMA 3.5. *If R is an HRS then $\to_R$ and $\Rightarrow_R$ coincide.*

In the sequel, we will not distinguish $\to_R$ and $\Rightarrow_R$ and use whichever definition is most appropriate. In addition we usually drop the subscript $_R$ and simply write $\to$.

A main theorem about $\stackrel{*}{\to}$ is stability under substitution. For TRSs this is simple and one obtains the stronger result that $s \to t$ implies $\theta s \to \theta t$. For HRSs we have to replace $\to$ by $\stackrel{*}{\to}$ because β-reductions during the application of a substitution can copy redexes. These copies need to be reduced sequentially; hence the use of $\stackrel{*}{\to}$ instead of $\to$. Later on we will see that a suitable notion of parallel reduction leads to a much nicer stability result (Lemma 6.3).

DEFINITION 3.6. Let $\to$ be an arbitrary relation on terms. We define $\theta \to \theta'$ to mean that $\theta(F) \to \theta'(F)$ holds for all $F \in \mathcal{D}om(\theta)$.

THEOREM 3.7. *Let R be an HRS. If $s \stackrel{*}{\to} s'$ and $\theta \stackrel{*}{\to} \theta'$ then $\theta s \stackrel{*}{\to} \theta' s'$.*

This theorem has two obvious corollaries:

- $s \stackrel{*}{\to} s'$ implies $\theta s \stackrel{*}{\to} \theta s'$ and
- $\theta \stackrel{*}{\to} \theta'$ implies $\theta s \stackrel{*}{\to} \theta' s$.

A different version of Theorem 3.7 is shown by Loría (Loría-Sáenz, 1993): he uses conditional rewrite rules whose left-hand sides are patterns.

3.1. *Rewriting versus Equality*

Originally, term rewriting was a means of analyzing equational theories, but it has long since taken on a life of its own. Returning to those roots we need to relate our notion of rewriting to a more "logical" notion of equality. In the sequel, E will always denote a set of **equations**, i.e. a set of pairs $s = t$, where s and t are terms of the same type. In particular any rewrite rule $l \to r$ can also be viewed as an equation $l = r$.

Throughout this subsection we do not assume that terms or substitutions are in any normal form and θt denotes the non-normalizing application of θ to t.

DEFINITION 3.8. A set of equations E induces a relation $=_E$ defined by the following inference rules, which come in three groups:

– Basic conversion rules:

$$\frac{(s = t) \in E}{\theta s =_E \theta t}\ (E) \qquad \frac{s =_\beta t}{s =_E t}\ (\beta) \qquad \frac{s =_\eta t}{s =_E t}\ (\eta)$$

– Equality rules:

$$\frac{}{s =_E s}\ (refl) \qquad \frac{s =_E t}{t =_E s}\ (sym) \qquad \frac{r =_E s \quad s =_E t}{r =_E t}\ (trans)$$

– Congruence rules:

$$\frac{s_1 =_E t_1 \quad s_2 =_E t_2}{(s_1\ s_2) =_E (t_1\ t_2)}\ (app) \qquad \frac{s =_E t}{\lambda x.s =_E \lambda x.t}\ (abs)$$

We call $=_E$ the **equational theory** generated by E.

Note that θ in (E) is present only for convenience: substitution can be simulated by (abs), (app) and (β).

Modulo $\updownarrow_\beta^\eta$ we have the same relationship between $\rightarrow_R$ and $=_R$ as in the first-order case. This can be viewed as a justification of our definition of $\rightarrow_R$.

THEOREM 3.9. ((Nipkow, 1993a; Mayr and Nipkow, 1998)). *If R is an HRS then $s =_R t \;\Leftrightarrow\; s\updownarrow_\beta^\eta \overset{*}{\leftrightarrow}_R t\updownarrow_\beta^\eta$.*

Thus we know that undirected rewriting and equational logic coincide.

Now we can use the fact that for confluent reductions convertibility and existence of a common reduct coincide:

COROLLARY 3.10. *If R is a confluent HRS then $s =_R t \;\Leftrightarrow\; s\updownarrow_\beta^\eta \downarrow_R t\updownarrow_\beta^\eta$.*

It should be pointed out that Theorem 3.9 fails for rules of function type. The relation $\overset{*}{\leftrightarrow}$ induced by the one-rule system

$$R = \{\lambda x.c(x, F(x)) \rightarrow \lambda x.d(F(x), x)\}$$

is strictly weaker than $=_R$: the equivalence $c(a, f(a)) =_R d(f(a), a)$ holds but $c(a, f(a)) \overset{*}{\leftrightarrow} d(f(a), a)$ does not hold because the definition of $\rightarrow$ insists on rewriting β-normal forms only. Otherwise one could β-expand $c(f(a), a)$ to $(\lambda x.c(x, f(x)))a$ before rewriting it to $(\lambda x.d(f(x), x))a$.

4. The Critical Pair Lemma

In 1972, Knuth and Bendix (Knuth and Bendix, 1970) showed that confluence of terminating term-rewriting systems is decidable: a simple test of confluence for the finite set of so called critical pairs suffices. This section generalizes the idea to PRSs (see Nipkow (Nipkow, 1991)) and prepares the ground for a further generalization to confluence modulo equality, which is treated in the following section.

DEFINITION 4.1. Let there be two rewrite rules $l_i \rightarrow r_i$, $i = 1, 2$, in a PRS and a position $p \in Pos(l_1)$ such that

- $fv(l_1) \cap bv(l_1) = \emptyset$,
- the head of l_1/p is not a free variable in l_1, and
- the two patterns $\lambda\overline{x_k}.(l_1/p)$ and $\lambda\overline{x_k}.(\sigma l_2)$, where $\{\overline{x_k}\} = bv(l_1, p)$ and σ is an $\overline{x_k}$-lifter of l_2 away from $fv(l_1)$, have a most general unifier θ.

Then the pattern l_1 **overlaps** the pattern l_2 at position p. The rewrite rules determine the **critical pair** $\langle \theta r_1, \theta(l_1[\sigma r_2]_p)\rangle$. Note that because $\lambda\overline{x_k}.(l_1/p)$ and $\lambda\overline{x_k}.(\sigma l_2)$ unify, l_1/p must be of the form $f(\ldots)$.

Two redexes t/p_1 and t/p_2 in a term t are **overlapping** if there are rewrite rules $l_i \rightarrow r_i$, $i = 1, 2$, such that $p_1 \leq p_2$, $t/p_1 = \theta_1 l_1$, $t/p_2 = \theta_2 l_2$ and l_1 overlaps l_2 at position p_2/p_1.

The **critical pairs** of a PRS R are all the critical pairs arising from overlapping two left-hand sides of rules in R, except for a left-hand side of a rule overlapping itself at position ε. Note that it is possible that a left-hand side l of a rule $l \rightarrow r$ overlaps itself at positions $p \neq \epsilon$ thus giving rise to a critical pair.

Thus we come to the main result of this section. It looks exactly like its counterpart from the term rewriting literature, but its proof is considerably more complex (see (Mayr and Nipkow, 1998)).

LEMMA 4.2. (Critical Pair Lemma (Nipkow, 1993a)). *Let R be a PRS and $\rightarrow$ the corresponding reduction relation on terms. If $s \rightarrow s_i$, $i =$*

1, 2, then either $s_1 \downarrow s_2$*, or there are a critical pair* $\langle u_1, u_2\rangle$*, a substitution* δ *and a position* $p \in Pos(s)$*, such that* $s_i = s[\delta u_i]_p$, $i = 1, 2$.

It is now easy to prove

THEOREM 4.3. *The relation* $\to$ *is locally confluent iff* $u_1 \downarrow u_2$ *for all critical pairs* $\langle u_1, u_2\rangle$ *of the PRS R.*

As for terminating relations confluence and local confluence are equivalent, this yields a decision procedure for the confluence of a terminating PRS.

COROLLARY 4.4. *Confluence of terminating PRSs is decidable.*

EXAMPLE 4.5. The syntax of classical predicate logic can be described by the two types *term* and *form* of terms and formulae and by the following constants:

$$\begin{array}{rl} \neg: & form \to form \\ \wedge, \vee: & form \to form \to form \\ \forall, \exists: & (term \to form) \to form \end{array}$$

In the following, P and Q will be variables of type *form* whereas P' and Q' will be of type $term \to form$. For readability we write $\forall x.P'(x)$ instead of $\forall(\lambda x.P'(x))$ (and similarly for $\exists$) and use $\wedge$ and $\vee$ as infix symbols. Note that the rules of λ-calculus automatically guarantee that $\forall x.P'(x)$ matches any universally quantified formulae, whereas $\forall x.P$ matches only those where the bound variable does not occur in the body of the quatifier.

The **negation normal form** (Gallier, 1986), where $\neg$ is only applied to atomic formulae, can be defined via the following terminating rewrite system:

$$\begin{array}{rrcl} \neg\neg: & \neg\neg P & \to & P \\ \neg\wedge: & \neg(P \wedge Q) & \to & (\neg P) \vee (\neg Q) \\ \neg\vee: & \neg(P \vee Q) & \to & (\neg P) \wedge (\neg Q) \\ \neg\forall: & \neg\forall x.P'(x) & \to & \exists x.\neg P'(x) \\ \neg\exists: & \neg\exists x.P'(x) & \to & \forall x.\neg P'(x) \end{array}$$

There are 5 critical pairs, all of which arise by unifying the left-hand side of some rule with the subterm $\neg P$ of $\neg\neg P$, and all of which are joinable. For example

$$
\begin{array}{ccc}
 & & \exists x.P'(x) \\
 & \nearrow_{\neg\neg} & \\
\neg\neg\exists x.P'(x) & & \\
 & \searrow_{\neg\exists} & \\
 & & \neg\forall x.\neg P'(x)
\end{array}
$$

is joinable because $\neg\forall x.\neg P'(x) \to \exists x.\neg\neg P'(x) \to \exists x.P'(x)$. Hence the system is confluent, i.e. the negation normal form is uniquely determined.

EXAMPLE 4.6. Going back to Example 3.3, the pure lambda-calculus, we find that both *beta* and *eta* have no critical pairs with themselves, and hence that *beta* and *eta* on their own are locally confluent. Since *eta* also terminates, this implies confluence. For the non-terminating *beta* we have to wait for the notion of orthogonality to deduce confluence. Combining *beta* and *eta* yields two critical pairs:

$$
\begin{array}{ccc}
 & & app(S,T) \\
 & \nearrow_{\beta} & \\
app(abs(\lambda x.app(S,x)),T) & & \\
 & \searrow_{\eta} & \\
 & & app(S,T)
\end{array}
$$

$$
\begin{array}{ccc}
 & & abs(\lambda x.F(x)) \\
 & \nearrow_{\beta} & \\
abs(\lambda x.app(abs(\lambda y.F(y)),x)) & & \\
 & \searrow_{\eta} & \\
 & & abs(\lambda y.F(y))
\end{array}
$$

Both critical pairs are trivially joinable. Hence *beta* + *eta* is locally confluent.

EXAMPLE 4.7. An interesting extension of *beta*+*eta* can be obtained by adding a constant $\perp : term$ representing the completely undefined term, and the two rules

$$
\begin{array}{rrcl}
\perp app: & app(\perp,S) & \to & \perp \\
\perp abs: & abs(\lambda x.\perp) & \to & \perp
\end{array}
$$

This system has two more critical pairs: *eta* overlaps $\perp app$ yielding the pair $\langle \perp, abs(\lambda x.\perp)\rangle$, and *beta* overlaps $\perp abs$ yielding $\langle \perp, app(\perp,S)\rangle$.

This is a nice example of completion: if either of the two rules had been omitted, it would have followed from the other one as a critical pair. Hence the system is locally confluent.

The confluence proof for both of these examples requires stronger theorems. See the end of Section 6.

One of the main selling points of critical pairs has been the fact that they come with a so-called "completion algorithm": a non-confluent rewrite system can be transformed into an equivalent (w.r.t. the equational theory) but confluent system by adding critical pairs as new reduction rules. As the last example indicates, this is also possible in our higher-order situation. However, higher-order critical pairs may no longer be pattern rewrite rules in case neither of the two components is a pattern. It is easy to see that this unfortunate state cannot arise if the original PRS we start with contains only rewrite rules where both the left and the right-hand sides are patterns, a rare situation in practice.

5. Confluence modulo Equality

The most important consequence of confluence is the uniqueness of normal forms. However, there are cases of non-confluent systems where the normal forms of any term are not unique, but somehow similar to each other.

EXAMPLE 5.1. In Example 4.5 we showed how to express predicate logic formulae as λ-terms. The **prenex normal form** can be described by a rewrite system R consisting of the rules

$$\begin{array}{lcrcl} \mathsf{Q}* & : & (\mathsf{Q}x.P'(x)) * Q & \to & \mathsf{Q}x.(P'(x) * Q) \\ *\mathsf{Q} & : & P * (\mathsf{Q}x.Q'(x)) & \to & \mathsf{Q}x.(P * Q'(x)) \end{array}$$

for all $\mathsf{Q} \in \{\forall, \exists\}$ and $* \in \{\wedge, \vee\}$, together with the rules $\neg\forall$ and $\neg\exists$ from Example 4.5.

This system terminates but is not confluent. The reason is that the term $(\mathsf{Q}x.P'(x)) * (\mathsf{Q}y.Q'(y))$ gives rise to the critical pair

$$\langle r, s\rangle = \langle \mathsf{Q}x.(P'(x) * (\mathsf{Q}y.Q'(y))), \mathsf{Q}y.((\mathsf{Q}x.P'(x)) * Q'(y))\rangle$$

and this critical pair has the two distinct normal forms $\mathsf{Q}x.\mathsf{Q}y.(P'(x) * Q'(y))$ and $\mathsf{Q}y.\mathsf{Q}x.(P'(x) * Q'(y))$.

This example shows that commutativity of quantifiers needs to be taken into account as well. Huet (Huet, 1980) introduced the notion of confluence *modulo equality* of first-order term rewriting systems. Below we present the generalizations of some of his results to PRSs as obtained by Mayr and Nipkow (Mayr and Nipkow, 1998).

Let us first look at abstract relations $\rightarrow$ and $\sim$.

DEFINITION 5.2. A relation $\rightarrow$ is **confluent modulo an equivalence relation** $\sim$ iff

$$x \sim y \wedge x \xrightarrow{*} x' \wedge y \xrightarrow{*} y' \;\;\Rightarrow\;\; \exists x'', y''.\; x' \xrightarrow{*} x'' \wedge y' \xrightarrow{*} y'' \wedge x'' \sim y''$$

A relation $\rightarrow$ is called **normalizing** if for every x there is a y in $\rightarrow$-normal form such that $x \xrightarrow{*} y$.

LEMMA 5.3. (Huet (Huet, 1980)). *Let $\rightarrow$ be normalizing and let $x{\downarrow}$ denote an arbitrary normal form of x. Then $\rightarrow$ is confluent modulo an equivalence relation $\sim$ iff $x \equiv y \;\;\Rightarrow\;\; x{\downarrow} \sim y{\downarrow}$ where $\equiv$ is $(\leftrightarrow \cup \sim)^*$.*

Thus we can replace the test for equivalence modulo $\rightarrow \cup \sim$ by a test of $\sim$-equivalence of $\rightarrow$-normal forms. If $\rightarrow$-normal forms are computable and $\sim$ is decidable, so is $\equiv$.

In the sequel $\cdot$ denotes composition of relations:

$$\rightarrow \cdot \sim \;\; = \;\; \{(x,z) \mid \exists y.\; x \rightarrow y \wedge y \sim z\}$$

We will now concentrate on the application of confluence modulo to PRSs. To make things work we have to assume that the PRS modulo which we compute is symmetric, i.e. for every rule $l \rightarrow r$ it also contains the rule $r \rightarrow l$. Although this is trivial to achieve by adding the converse of every rule, it means that, because the result must still be a PRS, both l and r must be patterns, neither l nor r are free variables and $fv(l) = fv(r)$.

DEFINITION 5.4. Let R be a PRS and $\mathcal{E}$ a symmetric PRS. Then $\langle R, \mathcal{E} \rangle$ is called an **equational PRS**.

The PRS R defines a relation $\rightarrow$ on terms as usual. The symmetric reduction relation on terms defined by $\mathcal{E}$ is denoted by $\vdash\!\dashv_{\mathcal{E}}$ or simply $\vdash\!\dashv$. The transitive reflexive closure $\sim \; = \; \overset{*}{\vdash\!\dashv}$ is an equivalence relation. It is called the equivalence relation defined by $\mathcal{E}$.

DEFINITION 5.5. An equational PRS $\langle R, \mathcal{E} \rangle$ is called **confluent** iff the relation $\rightarrow$ defined by R is confluent modulo the equivalence relation $\sim$ defined by $\mathcal{E}$.

DEFINITION 5.6. Let $\langle R, \mathcal{E}\rangle$ be an equational PRS. The **critical pairs of** $\mathcal{E}/R$ are defined as all critical pairs arising from overlapping (analogous to Definition 4.1) the left-hand sides of rules $(l_1 \to r_1)$ and $(l_2 \to r_2)$, where either $(l_1 \to r_1) \in \mathcal{E}$ and $(l_2 \to r_2) \in R$ or $(l_1 \to r_1) \in R$ and $(l_2 \to r_2) \in \mathcal{E}$.

DEFINITION 5.7. The **critical pairs of** $\langle R, \mathcal{E}\rangle$ are the union of the critical pairs of R and the critical pairs of $\mathcal{E}/R$.

The following reformulation of Huet's theorem yields as a sufficient condition for the confluence of an equational PRS and is due to (Mayr and Nipkow, 1998):

THEOREM 5.8. *Let $\langle R, \mathcal{E}\rangle$ be an equational PRS, such that R is left-linear and $(\to \cdot \sim)$ terminates. Then the equational PRS $\langle R, \mathcal{E}\rangle$ is confluent iff $u_1{\downarrow} \sim u_2{\downarrow}$ for all critical pairs $\langle u_1, u_2\rangle$ of $\langle R, \mathcal{E}\rangle$, where $u_i{\downarrow}$ is an arbitrary $\to$-normal form of u_i.*

EXAMPLE 5.9. Now we can prove that the PRS R from Example 5.1 is confluent modulo the symmetric PRS

$$\mathcal{E} = \{\mathsf{Q}x.\mathsf{Q}y.H(x,y) \leftrightarrow \mathsf{Q}y.\mathsf{Q}x.H(x,y) \mid \mathsf{Q} \in \{\forall, \exists\}\}.$$

As usual, $\sim$ denotes the equivalence relation defined by $\mathcal{E}$. It is easily checked that all critical pairs of R arise by overlapping $\mathsf{Q}*$ with $*\mathsf{Q}$, and that their two normal forms are $\langle r{\downarrow}, s{\downarrow}\rangle = \langle \mathsf{Q}x.\mathsf{Q}y.(P'(x) * Q'(y)), \mathsf{Q}y.\mathsf{Q}x.(P'(x) * Q'(y))\rangle$. Thus we have $r{\downarrow} \sim s{\downarrow}$.

In addition we can overlap the rules $\mathsf{Q}*$ and $*\mathsf{Q}$ with the rules in $\mathcal{E}$ which gives rise to the critical pairs of $\mathcal{E}/R$, whose normal forms are

$$\langle \mathsf{Q}x.\mathsf{Q}y.(H(x,y) * Q), \mathsf{Q}y.\mathsf{Q}x.(H(x,y) * Q)\rangle$$

and

$$\langle \mathsf{Q}x.\mathsf{Q}y.(Q * H(x,y)), \mathsf{Q}y.\mathsf{Q}x.(Q * H(x,y))\rangle,$$

both of which are contained in $\sim$.

As R is left-linear and $(\to \cdot \sim)$ terminates, it follows from Theorem 5.8 that $\langle R, \mathcal{E}\rangle$ is a confluent equational PRS: modulo quantifier-commutativity, R computes a unique prenex normal form. As $=_\mathcal{E}$ is decidable the relation $=_{R\cup\mathcal{E}}$ is decidable by Lemma 5.3.

It is interesting to note that confluence is destroyed by a frequently employed optimization in computing prenex forms:

$$\begin{array}{lcl}(\forall x.P'(x)) \wedge (\forall y.Q'(y)) & \to & \forall x.(P'(x) \wedge Q'(x)) \\ (\exists x.P'(x)) \vee (\exists y.Q'(y)) & \to & \exists x.(P'(x) \vee Q'(x))\end{array}$$

These new rules give rise to non-trivial critical pairs with R, requiring, for example, the further rule $\forall x.\forall y.(P'(x) \wedge Q'(y)) \to \forall x.(P'(x) \wedge Q'(x))$. It seems unlikely that confluence can be regained by some form of completion.

6. Orthogonal Pattern Rewrite Systems

We now turn our attention to a subclass of PRSs, the so called orthogonal ones. An **Orthogonal Pattern Rewrite System** (**OPRS**) is a PRS that is left linear and has no critical pairs. This means that there are no rules whose left-hand sides overlap (see Definition 4.1).

Orthogonal term-rewriting systems have a long history (O'Donnell, 1977; Huet and Lévy, 1991; Klop, 1992). They have been studied very closely because of their similarity to functional programs with pattern matching. The key property of orthogonal systems is their confluence, regardless of whether they terminate or not. We show that this holds for OPRSs as well. The main idea is to define a relation $\geqslant$ on terms such that $\to \subseteq \geqslant \subseteq \xrightarrow{*}$, which implies $\geqslant^* = \xrightarrow{*}$. It is well known that in this case $\to$ is confluent if $\geqslant$ has the **diamond property**: $r \geqslant s \wedge r \geqslant t \Rightarrow \exists u.\, s \geqslant u \wedge t \geqslant u$.

In the following, we sketch the confluence proof for OPRSs. The proof proceeds roughly like the one for the untyped λ-calculus due to Tait and Martin-Löf (Barendregt, 1984). For an alternative proof in the style of Takahashi (Takahashi, 1993) we refer to (Mayr and Nipkow, 1998).

Given a fixed HRS R, **parallel reduction** w.r.t. R is the smallest relation $\geqslant$ on terms which is closed under the following inference rules:

$$\frac{s_i \geqslant t_i \quad (i = 1, \ldots, n)}{a(\overline{s_n}) \geqslant a(\overline{t_n})}\ (\mathrm{A}) \qquad \frac{s \geqslant t}{\lambda x.s \geqslant \lambda x.t}\ (\mathrm{L})$$

$$\frac{s_i \geqslant t_i \quad (i = 1, \ldots, n) \quad c(\overline{t_n}) = \theta l \quad (l \to r) \in R}{c(\overline{s_n}) \geqslant \theta r}\ (\mathrm{R})$$

where a is an atom of type $\overline{\tau_n} \to \tau$ and $s_i : \tau_i$.

Parallel reduction has a number of interesting properties.

LEMMA 6.1. *$s \geqslant s$ holds for all terms s in long $\beta\eta$-normal form.*

More importantly we find that $\to$ and $\geqslant$ are strongly related:

LEMMA 6.2. $\to \subseteq \geqslant \subseteq \xrightarrow{*}$

Next we show that $\geqslant$ interacts very nicely with substitutions. The proof is similar to the one of Theorem 3.7 but considerably simpler.

LEMMA 6.3. *If $s \geqslant s'$ and $\theta \geqslant \theta'$ then $\theta s \geqslant \theta' s'$.*

As a consequence of some fairly technical lemma we obtain

LEMMA 6.4. *Let R be an OPRS and let l be the left-hand side of a rule in R. If $\theta l = a(\overline{s_n})$ and $s_i \geqslant t_i$ then there exists a θ' such that $\theta' l = a(\overline{t_n})$ and $\theta \geqslant \theta'$.*

As a direct consequence this yields

COROLLARY 6.5. (Coherence). *Let R be an OPRS. If $(l \to r) \in R$, $\theta l = a(\overline{s_n})$ and $s_i \geqslant t_i$, $i = 1, \ldots, n$, then there exists a θ' such that $\theta' l = a(\overline{t_n})$ and $\theta r \geqslant \theta' r$.*

Coherence is best explained in terms of the following diagram:

$$\begin{array}{ccccc} a(s_1,\ldots,s_n) & = & \theta l & \to & \theta r \\ \;\;\;\veeeq \quad\quad \veeeq & & & & \veeeq \\ a(t_1,\ldots,t_n) & = & \theta' l & \to & \theta' r \end{array}$$

We can finally show that $\geqslant$ has the diamond property, i.e. if $s \geqslant t_i$, $i = 0, 1$, then there exists a u such that $t_i \geqslant u$, $i = 0, 1$.

THEOREM 6.6. *(*Nipkow, 1993b*;* Mayr and Nipkow, 1998*)*
If R is an OPRS then $\geqslant$ has the diamond property.

The proof is by induction on the definition of $\geqslant$. As a corollary we obtain

COROLLARY 6.7. *Every OPRS is confluent.*

Going back to Example 3.3, we find that *beta* and *eta* on their own each form an OPRS. Thus the above corollary yields confluence of *beta* and *eta* reduction for untyped λ-calculus. However, as shown in Example 4.6, both rules together give rise to critical pairs and thus do not form an OPRS.

A more complex analysis of reductions shows that all *weakly orthogonal* PRSs are confluent (Oostrom and Raamsdonk, 1994; Raamsdonk, 1996). Although these papers define weak orthogonality directly rather than in terms of critical pairs, it can be translated as follows:

DEFINITION 6.8. A PRS R is called **weakly orthogonal** iff it is left-linear and all of its critical pairs are of the form $\langle u, u \rangle$.

Looking at the critical pairs in Example 4.6 once more, we find that they are both of the form $\langle u, u\rangle$. Hence the combination of *beta* and *eta* reduction is a weakly orthogonal PRS and therefore confluent as well.

Van Oostrom (Oostrom, 1997) has gone one step further and proved an analogue of Huet's parallel closure condition (Huet, 1980) for PRSs. He calls a PRS R **development-closed** if $s \mathrel{-\!\!\circ\!\!\!\rightarrow}_R r$ for every critical pair $\langle r, s\rangle$ of R, where $s \mathrel{-\!\!\circ\!\!\!\rightarrow}_R r$ means that a "complete development" (in the sense of λ-calculus) of all R-redexes of s yields r. He shows that every left-linear development-closed PRS is confluent. It is easy to verify that the four rule system in Example 4.7 is development-closed.

7. Termination

Termination of rewriting is undecidable, but there exist many results for terminating classes of rewrite systems or semi-decision procedures (see for example (Dershowitz and Jouannaud, 1990)). Termination of some HRS R can be proved by exhibiting a terminating ordering $>$ such that $\rightarrow_R \subseteq >$. The latter property can be proved by showing that $R \subseteq >$ and that this extends to $\rightarrow_R$. For the first-order case, there exist large classes of orderings that are known to extend to $\rightarrow_R$. In the higher-order case, such orderings are more difficult, as $>$ must be preserved by higher-order substitutions.

One current approach (Kahrs, 1995a; Pol, 1994) aims at finding appropriate semantic domains in order to interpret higher-order rewriting. In particular, strictly monotonic interpretations of terms in monotonic domains are used in (Pol, 1994). That is, (higher-order) symbols are interpreted by monotonic functions. It is shown that an HRS terminates if the interpretation of the right hand-side is smaller than the left-hand side for each rule. Alternatives which extend syntactic orderings on first-order terms (Dershowitz and Jouannaud, 1990) to higher-order terms are pursued in (Loría-Sáenz, 1993; Avenhaus and Loría-Sáenz, 1994; Lysne and Piris, 1995; Jouannaud and Rubio, 1996). These papers use first-order interpretations on (often restricted) λ-terms. A main problem is that basic first-order properties of orderings, e.g. stability under substitutions is more involved in λ-calculus.

A confluent and terminating HRS R is called **convergent**. It follows from Corollary 3.10 for convergent R that $s =_R t$ can be decided by comparing $s{\downarrow_R}$ and $t{\downarrow_R}$.

8. Unification of Higher-Order Patterns

Unification of higher-order patterns is a special case of higher-order unification that proceeds similarly to first-order unification. The important result is that most general unifiers exist for patterns. In addition to the first-order case, bound variables must be treated correctly: the unification problem

$$\lambda x.sin(F(x)) =^? \lambda x.sin(cos(x))$$

has solution $\{F \mapsto \lambda y.cos(y)\}$, whereas

$$\lambda x.F =^? \lambda x.sin(cos(x))$$

is unsolvable, as the left-hand side does not depend on x.

The following set of rules for unification of higher-order patterns is due to Nipkow (Nipkow, 1993a). The transformations in Figure 1 work on lists (in Prolog's [_ | _] notation), as the order of application is important for termination (Nipkow, 1993a). The problem is that the algorithm introduces new variables on the way and repeating this eagerly may lead to non-termination. For instance, consider $\{c(X) = Y, Y =^? X\} \Rightarrow_{PU} \{X =^? Y_1, c(Y_1) =^? X\}$. Here the occurs check applies only after an elimination whereas repeated imitation diverges.

The algorithm coincides with standard first-order unification algorithms, e.g. (Jouannaud and Kirchner, 1991), for first-order terms. Notice that in the Flex-Flex rules any permutation of the bound variables $\overline{z_p}$ is sufficient for computing a most general unifier.

It can be shown that PU terminates (Grobauer, 1997). More importantly, it computes most general unifiers in the following sense:

THEOREM 8.1. ((Miller, 1991b; Nipkow, 1993a)). *Let S_0 be a list of unification problems $s =^? t$, where both s and t are patterns.*

1. *If S_0 has a unifier, then S_0 has exactly one PU-normal form, namely [], and for every reduction sequence $S_0 \Rightarrow^{\theta_1} S_1 \ldots \Rightarrow^{\theta_n} []$ the composition $\theta_n \cdots \theta_1$ is a most general unifier.*
2. *If S_0 has no unifier, every PU-normal form of S_0 is distinct from [].*

Deletion

$$[t =^? t \mid S] \quad \Rightarrow \quad S$$

Decomposition

$$[\lambda\overline{x_k}.f(\overline{t_n}) =^? \lambda\overline{x_k}.f(\overline{t'_n}) \mid S] \quad \Rightarrow \quad [\overline{\lambda\overline{x_k}.t_n =^? \lambda\overline{x_k}.t'_n} \mid S]$$

Elimination

$$[F =^? t \mid S] \quad \Rightarrow^\theta \quad \theta S \quad \text{if } F \notin fv(t) \text{ and where } \theta = \{F \mapsto t\}$$

Imitation/Projection

$$[\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m}) \mid S] \quad \Rightarrow^\theta \quad [\overline{\lambda\overline{x_k}.H_m(\overline{y_n}) =^? \lambda\overline{x_k}.\theta t'_m} \mid \theta S]$$

where
$\theta = \{F \mapsto \lambda\overline{y_n}.v(\overline{H_m(\overline{y_n})})\}$,
v is a constant or $v \in \{\overline{y_n}\}$,
$\overline{H_m}$ are fresh variables,
and $F \notin fv(\lambda\overline{x_k}.v(\overline{t'_m}))$

Flex-Flex Same

$$[\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.F(\overline{y'_n}) \mid S] \quad \Rightarrow^\theta \quad \theta S$$

where $\theta = \{F \mapsto \lambda\overline{x_n}.F'(\overline{z_p})\}$
F' is a fresh variable, and
$\{\overline{z_p}\} = \{y_i \mid y_i = y'_i\}, i = 1, \ldots, n$

Flex-Flex Diff

$$[\lambda\overline{x_k}.F(\overline{y_n}) =^? \lambda\overline{x_k}.F'(\overline{y'_m}) \mid S] \quad \Rightarrow^\theta \quad \theta S \quad \text{if } F \neq F' \text{ and where}$$

$$\theta = \{F \mapsto \lambda\overline{y_n}.H(\overline{z_p}),\ F' \mapsto \lambda\overline{y'_m}.H(\overline{z_p})\},$$

H is a fresh variable,
and $\{\overline{z_p}\} = \{\overline{y_n}\} \cap \{\overline{y'_m}\}$

Figure 1. System PU for Pattern Unification

9. Higher-Order Unification by Transformations

We present in the following a version of the transformation system PT for higher-order unification of Snyder and Gallier (Snyder and Gallier, 1989).

Full higher-order unification is substantially more difficult than pattern unification. First, the former is undecidable and second, there may not be a single most general unifier. We even have to resort to a weaker form of unification, called pre-unification, as there may not even be a minimal complete set of unifiers (MCSU (Prehofer, 1997)). Pre-unification differs from unification by the handling of so-called **flex-flex pairs.** These are equations of the form $\lambda\overline{x_k}.P(\ldots) =^? \lambda\overline{x_k}.P'(\ldots)$. Huet (Huet, 1976) showed that such pairs of order three may not have a MCSU: there may exist an infinite chain of unifiers, one more general than the other, without any most general one. The important idea to remedy this situation is that flex-flex pairs are guaranteed to have at least one unifier, e.g. $\{P \mapsto \lambda\overline{x_m}.a, P' \mapsto \lambda\overline{x_n}.a\}$. The idea of pre-unification is to handle flex-flex pairs as constraints and not to attempt to solve them explicitly.

Consider solving an equation $\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})$ where v is not a free variable. Such equations are called **flex-rigid**. Clearly, for any solution θ to F the term $\theta F(\overline{t_n})$ must have (after β-reduction) the symbol v as its head. There are two possibilities:

- In the first case, v already occurs in (the solution to) some t_i. For instance, consider the equation $F(a) =^? a$, where $\{F \mapsto \lambda x.x\}$ is a solution based on a **projection**. In general, a projection binding for F is of the from $\{F \mapsto \lambda\overline{x_n}.x_i(\ldots)\}$. As some argument, here a, is carried to the head of the term, such a binding is called projection. This name was introduced in (Jensen and Pietrzykowski, 1976).
- The second case is that the head of the solution to F is just the desired symbol v. For instance, in the last example, an alternative solution is $\{F \mapsto \lambda x.a\}$. This is called **imitation.** Notice that imitation is not possible if v is a bound variable.

To solve a flex-rigid pair, the strategy is to guess an appropriate imitation or projection binding only for one rigid symbol, here a, and thus approximate the solution to F. Unification proceeds by iterating this process which focuses only on the outermost symbol. Roughly speaking, the rest of the solution for F is left open by introducing new variables. Formally, this is defined as follows.

Assume an equation $\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m})$, where all terms are in long $\beta\eta$-normal form. An **imitation binding** for F is of the form

$$F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})$$

where $\overline{H_m}$ are new variables of appropriate type. A **projection binding** for F is of the form

$$F \mapsto \lambda\overline{x_n}.x_i(\overline{H_p(\overline{x_n})})$$

where $\overline{H_p}$ are new variables with $\overline{H_p : \tau_p}$ and $x_i : \overline{\tau_p} \to \tau$. A **partial binding** is an imitation or a projection binding. Notice that in the above definition, the bindings are not written in long $\beta\eta$-normal form. The long $\beta\eta$-normal form of an imitation or projection binding can be written as

$$F \mapsto \lambda\overline{x_n}.v(\overline{\lambda\overline{z_{j_p}}.H_p(\overline{x_n},\overline{z_{j_p}})}).$$

A full exhibition of the types involved can be found in (Snyder and Gallier, 1989).

The transformation rules PT for higher-order unification in Figure 2 consist of the basic rules for unification, such as Deletion, Elimination and Decomposition plus the two rules explained above: Imitation and Projection. The rules work on pairs of terms to be unified, written as $\{u =^? v, \ldots\}$, and return a set of new pairs. We extend the transformation rules on goals to sets of goals in the canonical way: $\{s =^? t\} \cup S \Rightarrow^\theta \{\overline{s_n =^? t_n}\} \cup \theta S$ if $s =^? t \Rightarrow^\theta \{\overline{s_n =^? t_n}\}$. We abbreviate a sequence of transformations

$$G_0 \Rightarrow^{\delta_1}_{PT} G_1 \Rightarrow^{\delta_2}_{PT} \ldots \Rightarrow^{\delta_{n-1}}_{PT} G_{n-1} \Rightarrow^{\delta_n}_{PT} G_n$$

by $\stackrel{*}{\Rightarrow}{}^{\delta}_{PT}$, where $\delta = \delta_n \ldots \delta_1$.

A substitution θ is a **pre-unifier** of s and t if the equation $\theta s =^? \theta t$ can be simplified by Deletion, Decomposition and Elimination to a set of Flex-Flex pairs. In other words, $\theta s =^? \theta t$ only differ at subterms that have variable heads.

The following soundness lemma is easy to show:

THEOREM 9.1. *System PT is a sound transformation system for higher-order pre-unification.*

THEOREM 9.2. (Completeness of PT). *If $s =^? t$ has solution θ, i.e. $\theta s = \theta t$, then $\{s =^? t\} \stackrel{*}{\Rightarrow}{}^{\delta}_{PT} F$ such that δ is more general than θ and F is a set of flex-flex goals.*

Deletion

$$t =^? t \quad \Rightarrow \quad \emptyset$$

Decomposition

$$\lambda\overline{x_k}.v(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_n}) \quad \Rightarrow \quad \{\overline{\lambda\overline{x_k}.t_n =^? \lambda\overline{x_k}.t'_n}\}$$

Elimination

$$F =^? t \quad \Rightarrow^\theta \quad \emptyset \quad \text{if } F \notin fv(t) \text{ and where } \theta = \{F \mapsto t\}$$

Imitation

$$\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.f(\overline{t'_m}) \quad \Rightarrow^\theta \quad \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) =^? \lambda\overline{x_k}.\theta t'_m}\},$$

where $\theta = \{F \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$ is an appropriate imitation binding

Projection

$$\lambda\overline{x_k}.F(\overline{t_n}) =^? \lambda\overline{x_k}.v(\overline{t'_m}) \quad \Rightarrow^\theta \quad \{\lambda\overline{x_k}.\theta t_i(\overline{H_j(\overline{t_n})}) =^? \lambda\overline{x_k}.v(\overline{\theta t'_m})\},$$

where $\theta = \{F \mapsto \lambda\overline{x_n}.x_i(\overline{H_j(\overline{x_n})})\}$ is an appropriate projection binding

Figure 2. System PT for Higher-Order Unification

The above result does not show a practically important property of PT: it only computes incomparable solutions, if used properly. For this, we have to examine the application of rules more carefully. When applying the rules of system PT to a set of equations, there are two sources of non-determinism:

1. Which rule to apply
2. to which equation.

It was shown in the early work by Huet (Huet, 1976) that completeness does not depend on how the equations are selected. This is implicit in the proof in (Snyder and Gallier, 1989), and is also explained at the end of this section. Furthermore, the only branching occurs when both Imitation and Projection apply to some equation. In other words, application of the first three rules is deterministic (Snyder and Gallier, 1989). When examining the substitutions computed for the Imitation and the Projection rules, it can be seen that only incomparable solutions are computed.

Since higher-order unification is undecidable in general, we are interested in classes where higher-order unification is decidable. In many works concerning higher-order unification (Wolfram, 1993; Baader and Siekmann, 1994; Nadathur, 1987; Paulson, 1994), it is observed that non-termination of higher-order (pre-)unification occurs very rarely in practice. A large part of these observations are due to higher-order patterns, which cover most practical cases.

There are a few other results. It is shown in (Prehofer, 1994a) that the unification of a linear higher-order pattern s with an arbitrary second-order term that shares no variables with s is decidable. In particular, we do not have to resort to pre-unification, flex-flex pairs can be finitely solved in this case. The main application of this result is the unification of linear left-hand sides of rewrite rules with second-order terms, as employed in higher-order narrowing, which is the topic of the next section.

10. Lazy Narrowing

In this section, we introduce the narrowing calculus which is used for solving equational goals with free (or logic) variables modulo an equational theory. This is also called equational or E-unification.

In our setting we consider directed goals of the form $s \rightarrow^? t$ for ground terms t, where a substitution θ is a solution if $\theta s \stackrel{*}{\rightarrow}_R t$. Note that we have to generalize this to $\theta s \stackrel{*}{\rightarrow}_R \theta t$ in most theorems. To solve undirected equations of the form $s =^?_R t$ wrt confluent rewrite rules, we can just add a rule $(X =^? X) \rightarrow \mathit{true}$ and solve $(s =^? t) \rightarrow^? \mathit{true}$ and hence treat $=^?$ as a new function symbol. The focus on directed goals simplifies many technicalities and is essential for programming applications as shown later.

For the narrowing rules we need some notation. Let $s \stackrel{?}{\leftrightarrow} t$ stand for one of $s \rightarrow^? t$ and $t \rightarrow^? s$. For goals of the form $s \stackrel{?}{\leftrightarrow} t$, the rules are intended to preserve the orientation of $\stackrel{?}{\leftrightarrow}$. We extend the transformation rules on goals to sets of goals in the canonical way: $\{\overline{s \rightarrow^? t}\} \cup S \Rightarrow^\theta \{\overline{s_n \rightarrow^? t_n}\} \cup \theta S$ if $s \rightarrow^? t \Rightarrow^\theta \{\overline{s_n \rightarrow^? t_n}\}$. For a sequence $\Rightarrow^{\theta_1} \ldots \Rightarrow^{\theta_n}$ of LN steps, we write $\stackrel{*}{\Rightarrow}{}^\theta$, where $\theta = \theta_n \ldots \theta_1$.

System LN for lazy higher-order narrowing essentially consists of the rules for higher-order unification of the last section plus the Lazy Narrowing rules in Figure 3. Note that the rules for higher-order unification apply, which are presented with undirected goals apply in both directions to directed goals. For a first impression of lazy narrowing, we start with a few examples. Assuming the rules

$$\begin{array}{lcl} map(F,[X|Y]) & \to & [F(X)|map(F,Y)] \\ map(F,[]) & \to & [] \\ \\ father(mary) & \to & john \\ father(john) & \to & art \end{array}$$

we solve the goal $R(mary) \to^{?} art$ by

$$\begin{array}{lll} R(mary) \to^{?} art & \Rightarrow_{LN} & \text{Narrowing at Variable,} \\ & & R \mapsto \lambda x.father(R_1(x)) \\ R_1(mary) \to^{?} john, art \to^{?} art & \Rightarrow_{LN} & \text{Deletion or Decomposition} \\ R_1(mary) \to^{?} john & \Rightarrow_{LN} & \text{Narrowing at Variable,} \\ & & R_1 \mapsto \lambda x.father(R_2(x)) \\ R_2(mary) \to^{?} mary, john \to^{?} john & \Rightarrow_{LN} & \text{Projection} \\ mary \to^{?} mary, john \to^{?} john & \stackrel{*}{\Rightarrow}_{LN} & \text{Solved by Deletion} \end{array}$$

Thus we get the solution $R \mapsto \lambda x.father(father(x))$ by composing the partial bindings. Notice that the trivial solution $R \mapsto \lambda x.art$ is also possible here, but it is easy to avoid by further constraints as e.g. in the following example.

Another, slightly more involved example is the following. We use functional evaluation in this example for brevity.

$$\begin{array}{lll} map(F,[mary,john]) \to^{?} [john,art] & \stackrel{*}{\Rightarrow}_{LN} & \text{Evaluation,} \\ & & \text{Decomposition} \\ F(mary) \to^{?} john, F(john) \to^{?} art & \stackrel{*}{\Rightarrow}_{LN} & \text{Narrowing at Variable,} \\ & & F \mapsto \lambda x.father(H(x)) \\ H(mary) \to^{?} mary, john \to^{?} john, & & \\ \quad father(H(john)) \to^{?} art & \Rightarrow_{LN} & Projection, H \mapsto \lambda x.x \\ mary \to^{?} mary, john \to^{?} john, & & \\ \quad father(john) \to^{?} art & & \end{array}$$

The last goals are easily solved by evaluation and Deletion. This yields the solution $F \mapsto \lambda x.father(x)$. Observe how in the last examples Lazy Narrowing at Variable is used to compute solutions for functional

Lazy Narrowing with Decomposition

$$\lambda\overline{x_k}.f(\overline{t_n}) \to^? \lambda\overline{x_k}.t \quad \Rightarrow \quad \{\overline{\lambda\overline{x_k}.t_n \to^? \lambda\overline{x_k}.l_n},\ \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.t\}$$

where $f(\overline{l_n}) \to r$ is an $\overline{x_k}$-lifted rule

Lazy Narrowing at Variable

$$\lambda\overline{x_k}.H(\overline{t_n}) \to^? \lambda\overline{x_k}.t \quad \Rightarrow^\theta \quad \{\overline{\lambda\overline{x_k}.H_m(\overline{\theta t_n}) \to^? \lambda\overline{x_k}.l_m},\ \lambda\overline{x_k}.r \to^? \lambda\overline{x_k}.\theta t\}$$

where $f(\overline{l_m}) \to r$ is an $\overline{x_k}$-lifted rule, and $\theta = \{H \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$ with fresh variables $\overline{H_m}$

Figure 3. Narrowing Rules of System LN

variables. Although this rule is very powerful, it also has a high degree of non-determinism and will be restricted in later refinements.

There are two sources of non-determinism for such systems of transformations: which rules to apply and how to select the equations. Completeness fortunately does not depend on the goal selection, as each subgoal is independently solvable.

DEFINITION 10.1. A (higher-order) narrowing calculus $\mathcal{N}$ is complete for some HRS R if the following holds: If $s \to^? t$ has solution θ, i.e. $\theta s \xrightarrow{*}_R \theta t$, then $\{s \to^? t\} \overset{*}{\Rightarrow}{}^{\delta}_{\mathcal{N}} F$ such that δ is more general, modulo the newly added variables, than θ. Moreover, F is a set of flex-flex goals.

THEOREM 10.2. *System LN is complete.*

The above result has been stated for PRS in (Prehofer, 1994b) and proved for HRS in (Prehofer, 1995c). It should be noted that other approaches to narrowing and superposition found in the literature, see e.g. (Bachmair, 1991), are not suitable in the higher-order case. The difference is that many superposition calculi model rewrite steps at inner positions in terms, whereas lazy narrowing strictly works outside-in. As the term structur of λ-terms is much more involved, these calculi are not suitable in the higher-order case, as examined in (Prehofer, 1994b; Prehofer, 1995c)

11. Refinements for Convergent Rewrite Rules

This section develops several refinements which exploit the determinism of convergent systems. For convergent systems it is sufficient to consider normalized solutions and particular reductions in the proofs, which allows to remove redundancies in the solutions. The following results have been proved for PRS in (Prehofer, 1995b; Prehofer, 1995c) and are are generalized to HRS in (Prehofer, 1997).

There exist several other refinements (Prehofer, 1994b; Prehofer, 1995a; Prehofer, 1995c) if linearity of the left-hand sides of rewrite rules is assumed, which is common in programming applications, e.g. functional programs. For a class of even further restricted orthogonal rewrite rules, called inductively-sequential, an optimal calculus was presented in (Hanus and Prehofer, 1996). It is optimal in the sense that it produces only incomparable substitutions.

11.1. *Avoiding Lazy Narrowing at Variables*

An essential refinement is to avoid narrowing at a variable occurring in a term of the form $X(\overline{x_n})$ for a R-normalized substitution θ with an HRS R. For a pattern $X(\overline{x_n})$, reducibility of a term $\theta X(\overline{x_n})$ implies that θ is not R-normalized, hence violating the assumption on θ. This result generalizes the first-order case, as for first-order terms narrowing at variable position is not needed. It is an important optimization, as narrowing at variable positions is highly unrestricted and thus creates large search spaces. We strongly conjecture that in practice, as in higher-order logic programming (Michaylov and Pfenning, 1992), most terms are patterns and hence narrowing at variables is not needed very often.

To establish this result we need innermost reductions which evaluate terms at inner redices first. For any solution there exists an innermost reduction, if R is convergent.

DEFINITION 11.1. System **LNN** is defined as the restriction of system LN where Lazy Narrowing at Variable is not applied to goals of the form $\lambda\overline{x_n}.X(\overline{y_m}) \rightarrow^? t$ if $\lambda\overline{x_n}.X(\overline{y_m})$ is a higher-order pattern.

THEOREM 11.2. *System LNN is complete for a convergent HRS R wrt R-normalized solutions.*

11.2. *Simplification via Normalization*

Simplification by normalization of goals is one of the earliest and one of the most important optimizations (Hanus, 1992). Its motivation is to prefer deterministic reduction over search within narrowing.

Observe that normalization coincides with deterministic evaluation as in functional languages. Also, as shown below, deterministic operations are possible as soon as the left-hand side of a goal has been simplified to a term with a constructor at its root. For instance, with the rule $f(1) \rightarrow 1$, we can simplify a goal $f(1) \rightarrow^? g(Y)$ by $\{f(1) \rightarrow^? g(Y), \ldots\} \Rightarrow \{1 \rightarrow^? g(Y), \ldots\}$ and deterministically detect a failure.

For oriented goals, normalization is only complete for goals $s \rightarrow^? t$, where θt is in R-normal form for a solution θ. For instance, it suffices if t is a ground term in R-normal form. For most applications, this is no real restriction and corresponds to the intuitive understanding of directed goals.

DEFINITION 11.3. **Normalizing Lazy Narrowing** is defined as the rules of LNN plus arbitrary simplification steps on the left-hand sides of goals.

THEOREM 11.4. *Normalizing Lazy Narrowing is complete for convergent R wrt R-normalized solutions.*

11.3. *Deterministic Eager Variable Elimination*

Eager variable elimination is a particular strategy which has been examined for general equational unification. The idea is to apply the Elimination rule as a deterministic operation whenever possible. That is, when elimination applies to a goal, all other rule applications are not considered.

Eager variable elimination is of great practical value. Consider for instance the following example using the rules of Section 1:

$$\begin{array}{lll}
\{\lambda x.diff(\lambda y.1 * ln(F(y)), x) \rightarrow^? \lambda x.cos(x)/sin(x)\} & \stackrel{*}{\Rightarrow} & \text{Eval.} \\
\{\lambda x.diff(\lambda y.F(y), x)/F(x) \rightarrow^? \lambda x.cos(x)/sin(x)\} & \stackrel{*}{\Rightarrow} & \text{Decomp.} \\
\{\lambda x.diff(\lambda y.F(y), x) \rightarrow^? \lambda x.cos(x), & & \\
\;\lambda x.F(x) \rightarrow^? \lambda x.sin(x)\} & \stackrel{*}{\Rightarrow} & \text{Elim.}
\end{array}$$

At this point, we have to chose which goal to solve first. Since on the second Elimination applies, we prefer this and consider no further rules.

It remains the goal

$$\{\lambda x.diff(\lambda y.sin(y), x) \rightarrow^? \lambda x.cos(x)\}$$

Unfortunately, it is an open problem of general (first-order) equational unification if eager variable elimination is still complete (Snyder, 1991). In our case, with oriented goals, we obtain more precise results by differentiating the orientation of the goal to be eliminated. As we consider oriented equations, we can distinguish two cases of variable elimination. For goals $X \rightarrow^? t$, elimination is deterministic as we can show that for a solution θ, θt is in R-normal form.

THEOREM 11.5. *Lazy Narrowing with eager variable elimination on goals $X \rightarrow^? t$ where $X \notin fv(t)$ is complete for convergent HRS R*

12. Examples of Equational Reasoning

In the following, we present examples of reasoning in higher-order logic; more example can be found in (Prehofer, 1995c). We first come back to the introductory example in Section 1 and then consider an example of program transformation.

Assume the rules for differentiation of Section 1, plus the simplification rule $x * 1 \rightarrow x$. With these rules, we can solve the following goal with System LN:

$$\begin{array}{ll}
\{\lambda x.diff(\lambda y.ln(F(y)), x) \rightarrow^? \lambda x.cos(x)/sin(x)\} & \stackrel{*}{\Rightarrow} \text{Evaluation for } diff \\
\{\lambda x.diff(\lambda y.F(y), x)/F(x) \rightarrow^? \lambda x.cos(x)/sin(x)\} & \stackrel{*}{\Rightarrow} \text{Decomposition} \\
\{\lambda x.diff(\lambda y.F(y), x) \rightarrow^? \lambda x.cos(x), & \\
\ \lambda x.F(x) \rightarrow^? \lambda x.sin(x)\} & \stackrel{*}{\Rightarrow} \text{Elimination} \\
\{\lambda x.diff(\lambda y.sin(y), x) \rightarrow^? \lambda x.cos(x)\} & \stackrel{*}{\Rightarrow} \text{Evaluation} \\
\{\lambda x.cos(x) * diff(\lambda y.y, x) \rightarrow^? \lambda x.cos(x)\} & \stackrel{*}{\Rightarrow} \text{Evaluation} \\
\{\lambda x.cos(x) \rightarrow^? \lambda x.cos(x)\} & \stackrel{*}{\Rightarrow} \text{Decomposition} \\
\emptyset &
\end{array}$$

With a more refined narowing strategy (Prehofer, 1995c), it can be shown that there is no search necessary to find the solution $F \mapsto \lambda x.sin(x)$.

The utility of higher-order unification for program transformations has been shown nicely by Huet and Lang (Huet and Lang, 1978) and has been developed further in (Pfenning and Elliott, 1988; Hannan and Miller, 1988). The following example, taken from (Field and Harrison, 1988), applies unfold/fold program transformation and requires full instead of strict equality. We assume the following standard rules for lists

$$\begin{array}{lcl} map(F,[X|R]) & \rightarrow & [F(X)|map(F,R)] \\ foldl(G,[X|R]) & \rightarrow & G(X, foldl(G,R)) \end{array}$$

Now assume writing a function $g(F, L)$ by

$$g(F,L) \quad \rightarrow \quad foldl(\lambda x, y.plus(x,y), map(F,L))$$

that first maps F onto a list and then adds the elements via the function $plus$. This simple implementation for g is inefficient, since the list must be traversed twice. The goal is now to find an equivalent function definition that is more efficient. We can specify this with higher-order terms in a syntactic fashion by one simple equation:

$$\forall f, x, l.g(f,[x|l]) = B(f(x), g(f,l))$$

The variable B represents the body of the function to be computed and the first argument of B allows to use $f(x)$ in the body. The scheme on the right only allows recursing on l for g.

To solve this equation, we add a rule $X = X \rightarrow true$ and then apply narrowing. This yields the solution $\theta = \{B \mapsto \lambda fx, rec.plus(fx, rec)\}$, where fx and rec are bound variables. The solution can be illustrated as follows:

$$g(f,[x|l]) = \theta B(f(x), g(f,l)) = plus(f(x), g(f,l)).$$

This shows the more efficient definition of g. In this example, simplification can reduce the search space for narrowing drastically: it suffices to simplify the goal to

$$\begin{array}{ll} \lambda f, x, l.plus(f(x), foldl(plus, map(f,l))) & =^? \\ \lambda f, x, l.B(f(x), foldl(plus, map(f,l))), & \end{array}$$

where narrowing with the newly added rule $X = X \rightarrow true$ yields the two goals

$$\begin{array}{lcl} \lambda f, x, l.plus(f(x), foldl(plus, map(f,l))) & \rightarrow^? & \lambda f, x, l.X(f,x,l), \\ \lambda f, x, l.B(f(x), foldl(plus, map(f,l))) & \rightarrow^? & \lambda f, x, l.X(f,x,l). \end{array}$$

These can be solved by pure higher-order unification.

References

Avenhaus, J. and C. Loría-Sáenz: 1994, 'Higher-Order Conditional Rewriting and Narrowing'. In: J.-P. Jouannaud (ed.): *Constraints in Computational Logics*, Vol. 845 of *Lect. Notes in Comp. Sci.* pp. 269–284.

Baader, F. and T. Nipkow: 1998, *Term Rewriting and All That.* Cambridge University Press. To appear.

Baader, F. and J. H. Siekmann: 1994, 'Unification Theory'. In: D. Gabbay, C. Hogger, and J. Robinson (eds.): *Handbook of Logic in Artificial Intelligence and Logic Programming.* Oxford University Press.

Bachmair, L.: 1991, *Canonical Equational Proofs*, Progress in Theoretical Computer Science. Birkhäuser.

Barendregt, H. P.: 1984, *The Lambda Calculus, its Syntax and Semantics.* North-Holland, 2nd edition.

Barendregt, H. P.: 1992, 'Lambda Calculi with Types'. In: S. Abramsky, D. Gabbay, and T. Maibaum (eds.): *Handbook of Logic in Computer Science*, Vol. 2. Oxford University Press, pp. 118–309.

Dershowitz, N. and J.-P. Jouannaud: 1990, 'Rewrite Systems'. In: J. van Leeuwen (ed.): *Formal Models and Semantics, Handbook of Theoretical Computer Science, Vol. B.* Elsevier—MIT Press, pp. 243–320.

Field, A. J. and P. G. Harrison: 1988, *Functional Programming.* Addison-Wesley.

Gallier, J. H.: 1986, *Logic for Computer Science.* Harper & Row.

Goldfarb, W. D.: 1981, 'The Undecidability of the second-order unification problem'. *Theoretical Computer Science* **13**, 225–230.

Grobauer, B.: 1997, 'A verified Unification Algorithm for Higher-Order Patters'. Master's thesis, Institut für Informatik, TU München.

Hannan, J. and D. Miller: 1988, 'Uses of higher-order unification for implementing program transformers'. In: *Fifth International Logic Programming Conference.* pp. 942–959.

Hanus, M.: 1992, 'Improving Control of Logic Programs by Using Functional Logic Languages'. In: *Programming Language Implementation and Logic Programming*, Vol. 631 of *Lect. Notes in Comp. Sci.* pp. 1–23.

Hanus, M. and C. Prehofer: 1996, 'Higher-Order Narrowing with Definitional Trees'. In: H. Ganzinger (ed.): *Rewriting Techniques and Applications*, Vol. 1103 of *Lect. Notes in Comp. Sci.* pp. 138–152.

Hindley, J. R. and J. P. Seldin: 1986, *Introduction to Combinators and λ-Calculus.* Cambridge University Press.

Huet, G.: 1976, 'Résolution d'équations dans les langages d'ordre $1, 2, ..., \omega$'. Thèse d'Etat, Université Paris VII.

Huet, G.: 1980, 'Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems'. *J. ACM* **27**, 797–821.

Huet, G. and B. Lang: 1978, 'Proving and Applying Program Transformations Expressed with Second-Order Patterns'. *Acta Informatica* **11**, 31–55.

Huet, G. and J.-J. Lévy: 1991, 'Computations in Orthogonal Rewriting Systems'. In: J.-L. Lassez and G. Plotkin (eds.): *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, pp. 395–443.

Jensen, D. and T. Pietrzykowski: 1976, 'Mechanizing ω-order type theory through unification'. *Theoretical Computer Science* **3**, 123–171.

Jouannaud, J.-P. and C. Kirchner: 1991, 'Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification'. In: J.-L. Lassez and G. Plotkin (eds.): *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, pp. 257–321.

Jouannaud, J.-P. and A. Rubio: 1996, 'A recursive path ordering for higher-order terms in η-long β-normal form'. In: H. Ganzinger (ed.): *Rewriting Techniques and Applications*, Vol. 1103 of *Lect. Notes in Comp. Sci.* pp. 108–122.

Kahrs, S.: 1995a, 'Towards a Domain Theory for Termination Proofs'. In: J. Hsiang (ed.): *Rewriting Techniques and Applications*, Vol. 914 of *Lect. Notes in Comp. Sci.* pp. 241–255.

Kahrs, S.: 1995b, 'The Variable Containment Problem. Extended Abstract'. In: *Workshop on Higher-Order Algebra, Logic and Term Rewriting*.

Klop, J. W.: 1980, *Combinatory Reduction Systems*, Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam.

Klop, J. W.: 1992, 'Term Rewriting Systems'. In: S. Abramsky, D. Gabbay, and T. Maibaum (eds.): *Handbook of Logic in Computer Science*, Vol. 2. Oxford University Press, pp. 2–116.

Knuth, D. E. and P. Bendix: 1970, 'Simple Word Problems in Universal Algebra'. In: J. Leech (ed.): *Computational Problems in Abstract Algebra*. Pergamon Press, pp. 263–297.

Loría-Sáenz, C. A.: 1993, 'A Theoretical Framework for Reasoning about Program Construction Based on Extensions of Rewrite Systems'. Ph.D. thesis, Universität Kaiserslautern.

Lysne, O. and J. Piris: 1995, 'A Termination Ordering for Higher-Order Rewrite Systems'. In: J. Hsiang (ed.): *Rewriting Techniques and Applications*, Vol. 914 of *Lect. Notes in Comp. Sci.* pp. 26–40.

Mayr, R. and T. Nipkow: 1998, 'Higher-Order Rewrite Systems and their Confluence'. *Theoretical Computer Science* **192**, 3–29.

Michaylov, S. and F. Pfenning: 1992, 'An Empirical Study of the Runtime Behaviour of Higher-Order Logic Programs'. In: D. Miller (ed.): *Proc. Workshop on λProlog*. University of Pennsylvania, pp. 257–271. Available as Tech. Rep. MS-CIS-92-86.

Miller, D.: 1991a, 'A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification'. *Journal of Logic and Computation* **1**(4), 497–536.

Miller, D.: 1991b, 'A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification'. In: P. Schroeder-Heister (ed.): *Extensions of Logic Programming*. pp. 253–281.

Nadathur, G.: 1987, 'A Higher-Order Logic as the Basis for Logic Programming'. Ph.D. thesis, University of Pennsylvania, Philadelphia.

Nipkow, T.: 1991, 'Higher-Order Critical Pairs'. In: *6th IEEE Symp. Logic in Computer Science*. pp. 342–349.

Nipkow, T.: 1993a, 'Functional Unification of Higher-Order Patterns'. In: *8th IEEE Symp. Logic in Computer Science*. pp. 64–74.

Nipkow, T.: 1993b, 'Orthogonal Higher-Order Rewrite Systems are Confluent'. In: M. Bezem and J. Groote (eds.): *Proc. Int. Conf. Typed Lambda Calculi and Applications*, Vol. 664 of *Lect. Notes in Comp. Sci.* pp. 306–317.

O'Donnell, M. J.: 1977, *Computing in Systems Described by Equations*, Vol. 58 of *Lect. Notes in Comp. Sci.* Springer-Verlag.

Oostrom, V. v.: 1997, 'Developing Developments'. *Theoretical Computer Science* **175**, 159–181.

Oostrom, V. v. and F. v. Raamsdonk: 1994, 'Weak Orthogonality Implies Confluence: The Higher-Order Case'. In: A. Nerode (ed.): *Logical Foundations of Computer Science*, Vol. 813 of *Lect. Notes in Comp. Sci.* pp. 379–392.

Padovani, V.: 1995, 'On Equivalence Classes of Interpolation Equations'. In: M. Dezani-Ciancaglini and G. Plotkin (eds.): *Typed Lambda Calculi and Applications*, Vol. 902 of *Lect. Notes in Comp. Sci.* pp. 335–349.

Paulson, L. C.: 1994, *Isabelle: A Generic Theorem Prover*, Vol. 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag.

Pfenning, F. and C. Elliott: 1988, 'Higher-Order Abstract Syntax'. In: *Proc. SIGPLAN '88 Symp. Programming Language Design and Implementation*. pp. 199–208.

Pol, J. v. d.: 1994, 'Termination Proofs for Higher-Order Rewrite Systems'. In: J. Heering, K. Meinke, B. Möller, and T. Nipkow (eds.): *Higher-Order Algebra, Logic and Term Rewriting*, Vol. 816 of *Lect. Notes in Comp. Sci.* pp. 305–325.

Prehofer, C.: 1994a, 'Decidable Higher-order Unification Problems'. In: A. Bundy (ed.): *Automated Deduction — CADE-12*, Vol. 814 of *Lect. Notes in Comp. Sci.*

Prehofer, C.: 1994b, 'Higher-order Narrowing'. In: *9th IEEE Symp. Logic in Computer Science*. pp. 507–516.

Prehofer, C.: 1995a, 'A Call-by-Need Strategy for Higher-Order Functional-Logic Programming'. In: J. Lloyd (ed.): *Logic Programming*. pp. 147–161.

Prehofer, C.: 1995b, 'Higher-Order Narrowing with Convergent Systems'. In: *4th Int. Conf. Algebraic Methodology and Software Technology*, Vol. 936 of *Lect. Notes in Comp. Sci.*

Prehofer, C.: 1995c, 'Solving Higher-Order Equations: From Logic to Programming'. Technical Report I9508, Technische Universität, München. PhD Thesis.

Prehofer, C.: 1997, *Solving Higher-order Equations: From Logic to Programming*. Birkhäuser PCTS Series.

Qian, Z.: 1996, 'Unification of Higher-Order Patterns in Linear Time and Space'. *Journal of Logic and Computation* **6**, 315–341.

Raamsdonk, F. v.: 1996, 'Confluence and Normalization for Higher-Order Rewriting'. Ph.D. thesis, Vrije Universiteit, Amsterdam.

Snyder, W.: 1991, *A Proof Theory for General Unification*. Birkäuser.

Snyder, W. and J. Gallier: 1989, 'Higher-Order Unification Revisited: Complete Sets of Transformations'. *J. Symbolic Computation* **8**, 101–140.

Takahashi, M.: 1993, 'λ-Calculi with Conditional Rules'. In: M. Bezem and J. Groote (eds.): *Typed Lambda Calculi and Applications*, Vol. 664 of *Lect. Notes in Comp. Sci.* pp. 406–417.

Wolfram, D. A.: 1993, *The Clausal Theory of Types*, Cambridge Tracts in Theoretical Computer Science 21. Cambridge University Press.