Technical Report

UCAM-CL-TR-218
ISSN 1476-2986

Number 218

Higher-order critical pairs

Tobias Nipkow

April 1991

ISSN 1476-2986

Higher-Order Critical Pairs[*]

Tobias Nipkow[†]
University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
England
`Tobias.Nipkow@cl.cam.ac.uk`

Abstract

We consider rewrite systems over simply typed λ-terms with restricted left-hand sides. This gives rise to a one-step reduction relation whose transitive, reflexive and symmetric closure coincides with equality. The main result of the paper is a decidable confluence criterion which extends the well-known *critical pairs* to a higher-order setting. Several applications to typed λ-calculi and proof theory are shown.

[*]To appear in Proceedings of the 6th IEEE Symposium *Logic in Computer Science*, 1991.

[†]Research supported by ESPRIT BRA 3245, *Logical Frameworks*.

1 Introduction

In 1972, Knuth and Bendix published their seminal paper [12] which shows that confluence of terminating rewrite systems is decidable: a simple test of confluence for the finite set of so called critical pairs suffices. The objective of this paper is to generalize this construction from first-order rewrite systems (all functions are first-order) to rewrite systems over simply typed λ-terms. The aim of this generalization is to lift the rich theory developed around first-order rewrite systems and apply it to the manipulation of terms with bound variables such as programs, theorems and proofs.

The quest for a unified theory of first-order rewrite systems and λ-calculus goes back to Aczel [1] and the ground-breaking work by Klop [11]. Both authors use a meta-language of λ-terms with conversion rules built in at the meta-level. Within this framework the reduction rules of various λ-calculi can be expressed as object-level rewrite rules. This is in contrast to recent work by, for example, Breazu-Tannen [3], which does not distinguish meta and object-level, and where a fixed set of reduction rules for λ-terms are combined with arbitrary first-order rewrite rules.

The work in this paper is very close to that of Klop, although the formal foundations go back to Church [4]. Instead of defining our own meta-language, we use Church's simply typed λ-calculus. The latter comes with a well-defined equational theory and is part of a richer logical system which could be the basis for future extensions like conditional rewriting. In particular this paper can be seen as an investigation of the meta-theory of computer systems like λProlog [14] and Isabelle [15].

Section 2 reviews basic terminology and notation. Section 3 introduces a subclass of λ-terms, called *patterns*, which have unification properties resembling those of first-order terms. *Higher-order rewrite systems* are defined to be rewrite systems over λ-terms whose left-hand sides are patterns: this guarantees that the rewrite relation is easily computable. In Section 4 the notion of *critical pair* is generalized to higher-order rewrite systems and the analogue of the critical pair lemma is proved. The restricted nature of patterns is instrumental in obtaining these results. Finally, Section 5 applies the critical pair lemma to a number of λ-calculi and some first-order logic formalized by higher-order rewrite systems.

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 [8, 5, 10].

Starting with some fixed set of *base types* $\mathcal{B}$, the set $\mathcal{T}$ of all *types* is the closure of $\mathcal{B}$ under the function space constructor "$\rightarrow$". The letter τ represents types. Function types associate to the right: $\tau_1 \rightarrow \tau_1 \rightarrow \tau_3$ means $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. Instead of $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ we write $\overline{\tau_m} \rightarrow \tau$. The latter form is used only 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 $V_\tau \cap V_{\tau'} = C_\tau \cap C_{\tau'} = \{\}$ if $\tau \neq \tau'$, by λ-abstraction and application. Variables are denoted by x, y and z, 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. The inductive definition of *simply typed λ-terms* is as follows:

$$\frac{x \in V_\tau}{x : \tau} \qquad\qquad \frac{c \in C_\tau}{c : \tau}$$

$$\frac{s : \tau \rightarrow \tau' \quad t : \tau}{(s\ t) : \tau'} \qquad\qquad \frac{x : \tau \quad s : \tau'}{(\lambda x.s) : \tau \rightarrow \tau'}$$

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

Instead of $\lambda x_1 \ldots \lambda x_n.s$ we write $\lambda x_1, \ldots, x_n.s$ or just $\lambda \overline{x_n}.s$, where the x_i are assumed to be distinct. Similarly we write $t(u_1, \ldots, u_n)$ or just $t(\overline{u_n})$ instead of $(\ldots(t\ u_1)\ldots)u_n$. The *free* and *bound* variables occurring in a term are denoted by $\mathcal{FV}(s)$ and $\mathcal{BV}(s)$, respectively. Capital letters denote free variables.

We assume the usual definition of α, β and η conversion [8] between λ-terms. We write $s =_\gamma t$, where $\gamma \in \{\alpha, \beta, \eta\}$ if s and t are equivalent modulo γ-conversion, and $s \equiv t$ iff s and t are equivalent modulo α, β and η conversion. We write $s \rightarrow_\beta t$ if t is the result of a single β-reduction of s.

The simply typed λ-calculus is confluent and terminating w.r.t. β-reduction (η-reduction) and the *β-normal form* (*η-normal form*) of a term t is denoted by $t{\downarrow_\beta}$ ($t{\downarrow_\eta}$). Let t be in β-normal form. Then t is of the form $\lambda \overline{x_n}.a(\overline{u_m})$, 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 $t : \overline{\tau_{n+k}} \rightarrow \tau$ and $x_{n+1}, \ldots, x_{n+k} \notin \mathcal{FV}(\overline{u_m})$. Instead of $t{\downarrow_\beta}{\uparrow_\eta}$ we write $t{\updownarrow}$ or $\hat{t}$. A term t is in *$\beta\eta l$-form* if $t = \hat{t}$. It is well known [8] that $s \equiv t$ iff $\hat{s} =_\alpha \hat{t}$.

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 [10, 5]. 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. Note that natural numbers are valid positions. We write $p \leq q$ if p is a prefix of q. In that case there is a p' such that $pp' = q$ and q/p is defined as the suffix p'. If neither $p \leq q$ nor $p \geq q$, we write $p \parallel q$, indicating that p and q are in different subtrees. 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.

Abstractions and applications yield the following trees:

$$\begin{array}{c} \lambda x \\ | \\ s \end{array} \qquad\qquad \begin{array}{ccc} & \cdot & \\ / & & \backslash \\ s & & t \end{array}$$

Hence positions in λ terms are sequences over $\{1, 2\}$. Note that the bound variable in an abstraction is not a separate subterm and can therefore not be accessed by the s/p notation.

It has to be stressed that we do *not* work with α-equivalence classes of terms. Otherwise the notation s/p would not make sense because $=_\alpha$ is not a congruence w.r.t. $/$: $s_1 =_\alpha s_2$ does not imply $s_1/p =_\alpha s_2/p$ because s_i/p may contain free variables which are bound in s_i.

Substitutions are finite mappings from variables to terms of the same type. Substitutions are denoted by θ. If $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ we define $\mathcal{D}om(\theta) = \{x_1, \ldots, x_n\}$, $\mathcal{C}od(\theta) = \{t_1, \ldots, t_n\}$. The application of a substitution to a term is defined by $\theta(t) = (\lambda\overline{x_n}.t)(\overline{t_n})\downarrow$. A *renaming* ρ is an injective substitution with $\mathcal{C}od(\rho) \subset V$. Renamings are always denoted by ρ.

Given a relation $\rightarrow$, $\rightarrow^*$ denotes the transitive and reflexive closure of $\rightarrow$. We write $s \downarrow t$ iff there is a u such that $s \rightarrow^* u$ and $t \rightarrow^* u$. The relation $\rightarrow$ is *(locally) confluent* if $r \rightarrow^* s$ $(r \rightarrow s)$ and $r \rightarrow^* t$ $(r \rightarrow t)$ imply $s \downarrow t$. The relation $\rightarrow$ is *terminating* if there is no infinite sequence $s_i \rightarrow s_{i+1}$ for all $i \in \mathbb{N}$. It is well known that terminating relations are confluent iff they are locally confluent [10].

3 Higher-Order Rewrite Systems

Higher-Order Rewrite Systems (HRS) are similar to Klop's *Combinatory Reduction Systems (CRS)* [11]. Both are generalizations of first-order rewrite systems [5] to terms with higher-order functions and bound variables. The main difference is that Klop's positive results cover mainly regular systems, i.e. no overlaps and no repeated variables on the left-hand side, whereas this paper makes no such restrictions[1].

Definition 3.1 A 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.

Examples of higher-order patterns are $\lambda x.c(x)$, X, $\lambda x.F(\lambda z.x(z))$, and $\lambda x,y.F(y,x)$, examples of non-patterns are $F(c)$, $\lambda x.F(x,x)$ and $\lambda x.F(F(x))$.

The following crucial result about unification of patterns is due to Dale Miller [13]:

Theorem 3.2 *It is decidable whether two patterns are unifiable; if they are unifiable, a most general unifier can be computed.*

This result will ensure both the computability of the rewrite relation defined by an HRS and of critical pairs. Appendix A presents a simplified form of Miller's unification algorithm.

Definition 3.3 A *rewrite rule* is a pair $l \rightarrow r$ such that l is a pattern but not η-equivalent to a free variable, l and r are of the same type, and $\mathcal{FV}(l) \supseteq \mathcal{FV}(r)$. A *Higher-Order Rewrite System* (for short: *HRS*) is a finite set of rewrite rules. The letter R always denotes an HRS. An HRS R induces a relation $\rightarrow_R$ on terms:

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

[1]Of course we pay the price of obtaining only *local* confluence results.

Note that $\rightarrow_R$ is invariant under $\equiv$: $s' \equiv s \rightarrow_R t \equiv t'$ implies $s' \rightarrow_R t'$. Therefore $\rightarrow_R$ is infinitely branching and should be thought of as a rewrite rule between $\equiv$ equivalence classes.

Although this definition of rewriting is very restrictive (only subterms of terms in $\beta\eta l$-form may be rewritten) it is only decidable because of Theorem 3.2. For more general left-hand sides, the existence of a matching substitution θ may not be decidable.

It may be tempting to think that the restriction to patterns as left-hand sides is "only" due to computability considerations. However, there are other reasons as well. Consider the rule

$$f(c(F(X), F(a))) \rightarrow f(X), \tag{1}$$

where the left-hand side is not a pattern. Substituting $\lambda x.Y$ for F yields $f(c(Y,Y)) \rightarrow f(X)$, which is not a proper rewrite rule since X occurs on the right but not the left-hand side. This problem surfaces when rewriting the term $f(c(a,a))$, which forces that very substitution and yields the new term $f(X)$, thus introducing a new free variable. These problems are caused by the fact that we rewrite modulo an equational theory, the λ-calculus, which is irregular: $s \equiv t$ does not imply $\mathcal{FV}(s) = \mathcal{FV}(t)$. It is the restriction to patterns as left-hand sides which guarantees that the matching substitution θ is ground.

In addition to the operational $\rightarrow_R$ we have the logical notion of *equality modulo* R. The latter is formalized by taking α, β, and η-conversion together with all instances of R

$$\{\theta l = \theta r \mid (l \rightarrow r) \in R\}$$

as axioms and closing under reflexivity, symmetry, transitivity and the two congruence laws

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

The resulting relation is called $=_R$. For a concise statement of the connection between rewriting and equality the following notation is useful: given an equivalence $\cong$ and a relation $\rightarrow$ on terms, we write $[s]_\cong \rightarrow [t]_\cong$ iff there are s' and t' such that $s \cong s' \rightarrow t' \cong t$. The following theorem says that, modulo $\equiv$, $\leftrightarrow_R^*$ coincides with $=_R$:

Theorem 3.4 *If all rules in R are of base type, then*

$$s =_R t \quad \Leftrightarrow \quad [s]_\equiv \overset{*}{\underset{R}{\leftrightarrow}} [t]_\equiv$$

Thus we know in particular that, if $\rightarrow_R$ is terminating and confluent, two terms are equivalent modulo $=_R$ iff their $\rightarrow_R$ normal forms are equivalent modulo $\equiv$.

Notice that this equivalence between $=_R$ and $\leftrightarrow_R^*$ does only work for rules of base type. Given the rule

$$\lambda x.c(x, F(x)) \ \rightarrow\ \lambda x.d(F(x), x),$$

the relation $\leftrightarrow_R^*$ is strictly weaker than $=_R$: although $c(a, f(a)) =_R d(f(a), a)$ holds, $c(a, f(a)) \leftrightarrow_R^* d(f(a), a)$ does not hold because both terms are in normal form w.r.t. R. The reason is the restrictive definition of $\rightarrow_R$ which insists on rewriting only terms in β-normal form. Otherwise it would be easy to rewrite $c(a, f(a)) \equiv (\lambda x.c(x, f(x)))(a)$. It is not obvious that the restriction to β-normal forms can be dropped without sacrificing decidability of $\rightarrow_R$.

It should be noted that another obvious way out, pulling all rules down to base type by supplying them with new free variables, fails. In the above case this yields the new rule $c(X, F(X)) \rightarrow d(F(X), X)$, where the left-hand side is not a pattern, thus leading to all the problems described in connection with rule (1) above.

The remainder of this section sketches the proof of Theorem 3.4, which proceeds via an auxiliary rewrite relation

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

The whole point of $\rightarrow_{[R]}$ is to allow rewriting of terms not in β-normal form.

The following two lemmas provide simple relationships between $\rightarrow_R$ and $\rightarrow_{[R]}$:

Lemma 3.5 *If $s \rightarrow_R t$ then $\hat{s} \rightarrow_{[R]} \hat{t}$.*

Lemma 3.6 *If all rules in R are of base type, then $s{\downarrow_\beta} \rightarrow_{[R]} t$ implies $s \rightarrow_R t$.*

Since $\rightarrow_{[R]}$ preserves β-normal form we have:

Corollary 3.7 *If all rules in R are of base type, then $s{\downarrow_\beta} \rightarrow_{[R]}^* t$ implies $s \rightarrow_R^* t$.*

A careful analysis of redexes proves

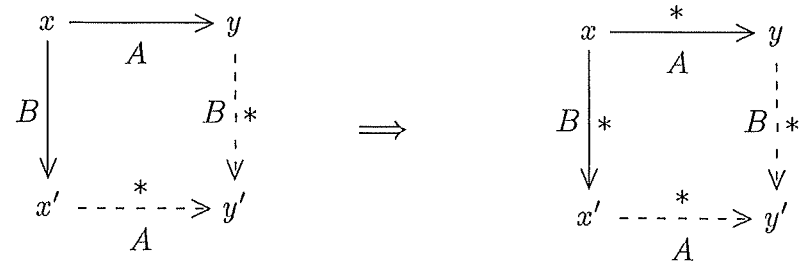
Lemma 3.8 *Let R be an HRS were all rules are of base type. If $[s]_{=_\alpha} \rightarrow_{[R]} [t]_{=_\alpha}$ and $[s]_{=_\alpha} \rightarrow_\beta [s']_{=_\alpha}$ then there is a t' such that $[s']_{=_\alpha} \rightarrow_{[R]}^* [t']_{=_\alpha}$ and $[t]_{=_\alpha} \rightarrow_\beta^* [t']_{=_\alpha}$.*

To extend this lemma to many-step reductions, we need the following result about abstract reduction relations:

Lemma 3.9 *Let $\rightarrow_A$, $\rightarrow_B$ and $>$ be relations on some set S such that $>$ is a terminating partial order, $s \rightarrow_A t$ implies $s \geq t$, and $s \rightarrow_B t$ implies $s > t$. Then*

$$\begin{gathered}\forall x, x', y.\ x \rightarrow_B x' \wedge x \rightarrow_A y \ \Rightarrow\ \exists y'.\ x' \rightarrow_A^* y' \wedge y \rightarrow_B^* y' \\ \Rightarrow \\ \forall x, x', y.\ x \rightarrow_B^* x' \wedge x \rightarrow_A^* y \ \Rightarrow\ \exists y'.\ x' \rightarrow_A^* y' \wedge y \rightarrow_B^* y'.\end{gathered}$$

In pictures:



$$\begin{tikzcd} x \arrow[r, "A"'] \arrow[d, "B"'] & y \arrow[d, dashed, "B" ', "*"] \\ x' \arrow[r, dashed, "*", "A"'] & y' \end{tikzcd} \quad \Longrightarrow \quad \begin{tikzcd} x \arrow[r, "*", "A"'] \arrow[d, "B"', "*"] & y \arrow[d, dashed, "B"', "*"] \\ x' \arrow[r, dashed, "*", "A"'] & y' \end{tikzcd}$$

The proof proceeds by well-founded induction on x using the relation $>$.

Combining Lemmas 3.8 and 3.9 yields

Lemma 3.10 *Let all rules in R be of base type. If $[s]_{=_\alpha} \rightarrow_\beta^* [s']_{=_\alpha}$ and $[s]_{=_\alpha} \rightarrow_{[R]}^* [t]_{=_\alpha}$, then there is a t' such that $[s']_{=_\alpha} \rightarrow_{[R]}^* [t']_{=_\alpha}$ and $[t]_{=_\alpha} \rightarrow_\beta^* [t']_{=_\alpha}$.*

The proof uses the ordering $>$ induced by the length of the longest chain of β-reductions starting from a term.

Finally we can prove that, modulo $=_\alpha$, $\rightarrow_{[R]}^*$ is contained in $\rightarrow_R^*$.

Theorem 3.11 *If all rules in R are of base type then $[s]_{=_\alpha} \rightarrow_{[R]}^* [t]_{=_\alpha}$ implies $[s]_{=_\alpha} \rightarrow_R^* [t]_{=_\alpha}$.*

Proof From $[s]_{=_\alpha} \rightarrow_{[R]}^* [t]_{=_\alpha}$ it follows by Lemma 3.10 that $[s{\downarrow_\beta}]_{=_\alpha} \rightarrow_{[R]}^* [t']_{=_\alpha}$ for some $t' \equiv t$. Corollary 3.7 yields $[s{\downarrow_\beta}]_{=_\alpha} \rightarrow_R^* [t']_{=_\alpha}$ and the proposition follows because $\rightarrow_R$ is invariant under $\equiv$. □

At last, we connect rewriting and directed equality: let $\geq_R$ be the subset of $=_R$ obtained by omitting the symmetry rule.

Theorem 3.12 *If all rules in R are of base type, then*

$$s \geq_R t \quad \Leftrightarrow \quad [s]_\equiv \overset{*}{\underset{R}{\rightarrow}} [t]_\equiv$$

Proof The $\Leftarrow$-direction follows easily from the definitions. The $\Rightarrow$-direction is proved by induction on the structure of $\geq_R$ derivations. For the conversion rules and the laws of reflexivity and transitivity this is trivial. For congruence w.r.t. abstraction it follows because $s \rightarrow_R t$ implies $\lambda x.s \rightarrow_R \lambda x.t$. The only tricky case is congruence w.r.t. application, the *raison d'être* for $\rightarrow_{[R]}$. If $s_i \geq_R t_i$, the induction hypothesis implies $[s_i]_\equiv \rightarrow_R^* [t_i]_\equiv$ and hence $[s_i]_{=_\alpha} \rightarrow_R^* [t_i]_{=_\alpha}$. Hence it follows from Lemma 3.5 that $[\hat{s}_i]_{=_\alpha} \rightarrow_{[R]}^* [\hat{t}_i]_{=_\alpha}$. The definition of $\rightarrow_{[R]}$ implies that $[(\hat{s}_1\ \hat{s}_2)]_{=_\alpha} \rightarrow_{[R]}^* [(\hat{t}_1\ \hat{t}_2)]_{=_\alpha}$ and Theorem 3.11 yields $[(\hat{s}_1\ \hat{s}_2)]_{=_\alpha} \rightarrow_R^* [(\hat{t}_1\ \hat{t}_2)]_{=_\alpha}$. Because $\rightarrow_R$ is invariant under $\equiv$, this implies $[(s_1\ s_2)]_\equiv \rightarrow_R^* [(t_1\ t_2)]_\equiv$, thus concluding the proof. □

Theorem 3.4 is a direct consequence of Theorem 3.12.

4 The Critical Pair Lemma

Confluence of terminating (first-order) term rewriting systems is a decidable property [12]. The decision procedure is based on an analysis of so called *critical pairs* [10], which are patterns of interference between rules. They arise by unifying the left-hand side of one rule with the subterm of the left-hand side of another rule:

$$\begin{aligned} (x \times y) \times z &\rightarrow x \times (y \times z) \\ i(x \times y) &\rightarrow i(y) \times i(x) \end{aligned}$$

gives rise to two critical pairs, one of which is the result of reducing $i((x \times y) \times z)$ in two different ways:

$$\begin{tikzcd}
i((x \times y) \times z) \arrow[r] \arrow[d] & i(z) \times i(x \times y) \arrow[d, dashed, "*"] \\
i(x \times (y \times z)) \arrow[r, dashed, "*"] & i(z) \times (i(y) \times i(x))
\end{tikzcd}$$

The dashed arrows indicate the common reduct of the critical pair. In the above case both critical pairs have a common reduct[2].

Due to the presence of bound variables, critical pairs for HRSs are more complex:

Definition 4.1 Let $l_i \rightarrow r_i$, $i = 1,2$, be two rules such that $\mathcal{FV}(l_1) \cap \mathcal{BV}(l_1) = \{\}$, let $p \in \mathcal{P}os(l_1)$ such that the head of l_1/p is not a free variable, let $\overline{x_k} = \mathcal{FV}(l_1/p) \cap \mathcal{BV}(l_1)$ where $x_i : \tau_i$, let ρ be a renaming such that $\mathcal{D}om(\rho) = \mathcal{FV}(l_2)$, $\mathcal{C}od(\rho) \cap \mathcal{FV}(l_1) = \{\}$, and $\rho(x) : \overline{\tau_k} \rightarrow \tau$ if $x : \tau$, let $\sigma = \{x \mapsto \rho(x)(\overline{x_k}) \mid x \in \mathcal{FV}(l_2)\}$, and let θ be a most general unifier of $\lambda\overline{x_k}.l_1/p$ and $\lambda\overline{x_k}.\sigma l_2$, both of which are patterns. Then

$$r_1 = l_1[\sigma r_2]_p$$

is called a *critical pair.*

Lemma 4.2 (Critical Pair Lemma) *Let R be an HRS where all rules are of base type. If $s \rightarrow_R t_1$ and $s \rightarrow_R t_2$, then either $t_1 \downarrow_R t_2$, or there are a critical pair $u_1 = u_2$, a substitution δ, and a position $p_1 \in \mathcal{P}os(\hat{s})$ such that $t_i \equiv \hat{s}[\delta u_i]_{p_1}$ for $i = 1,2$.*

The proof proceeds roughly as in the first-order case [10], but is considerably more subtle in its details. Many derivations that are trivial for first-order terms require explicit sublemmas. These sublemmas depend crucially on the fact that the left-hand sides in an HRS are patterns. Even a slight generalization invalidates the lemma. Consider the rules

$$\begin{aligned} c_1(\lambda x.F(x,x)) &\rightarrow c_2(F) \\ d_1(X,X) &\rightarrow d_2 \end{aligned}$$

where $c_1(\lambda x.F(x,x))$ is not a pattern because of the repeated x. The term $c_1(\lambda x.d_1(x,x))$ rewrites to both $c_2(d_1)$ and $c_1(\lambda x.d_2)$, which have no common reduct (because they are both in normal form), and there is no critical pair either. This shows that the whole setup is very sensitive to the form of the rewrite rules.

The following corollary is an easy consequence of the Critical Pair Lemma:

Corollary 4.3 *Let R be an HRS where all rules are of base type. If $u_1 \downarrow_R u_2$ for all critical pairs $u_1 = u_2$, then $\rightarrow_R$ is locally confluent.*

[2]The other case is $(x \times y) \times (z \times v) \longleftarrow ((x \times y) \times z) \times v \rightarrow (x \times (y \times z)) \times v$ with common reduct $x \times (y \times (z \times v))$.

For terminating HRSs this yields a decision procedure for local confluence and hence for confluence.

Although we have not mentioned this aspect so far, it is obvious that the critical pair lemma gives rise to a completion algorithm: critical pairs without common reduct are turned into rewrite rules and added to the non-confluent system. This process may need to be repeated to generate a confluent system. It is not clear whether the standard results on completion (e.g. [2]) carry over to higher-order systems. In order to automate the completion process it is also necessary to test for termination of HRSs, an issue that seems completely unexplored.

5 Applications

Higher-order rewrite systems have the same logical basis as systems like Isabelle [15] and λProlog [14] which are designed for the manipulation of terms with bound variables. Hence it is hardly surprising that many logical reduction calculi can be expressed as HRSs. This section presents a number of such calculi and applies the critical pair lemma. The syntax of each calculus is defined by a signature which is a set of types plus a set of typed constants. Terms are generated by application and abstraction, as defined in Section 2, subject to the type constraints in the signature.

5.1 λ-Calculi

Untyped λ-calculi tend to be nonterminating systems, for which confluence does not follow from local confluence. However, it is easy to see that confluence of the terminating fragment, i.e. the set of terms all of whose reductions terminate, does follow. This encompasses in particular all those typed variants which are known to terminate.

The syntax of the pure λ-calculus involves just the type *term* of terms and two constants for abstraction and application:

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

The rewrite rules are:

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

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

The rules β and η on their own do not generate any (non-trivial) critical pair. Their combination, however, gives rise to two critical pairs which stem from the solutions to the two unification problems $abs(F) \equiv abs(\lambda x.app(S, x))$ and $\lambda x.app(S, x) \equiv \lambda x.app(abs(G(x)), T(x))$:

$$\begin{tikzcd}[column sep=small] & app(S,T) \\ app(abs(\lambda x.app(S,x)),T) \arrow[ur, "\beta"] \arrow[dr, "\eta"'] & \\ & app(S,T) \end{tikzcd}$$

$$\begin{tikzcd}[column sep=small] & abs(\lambda x.H(x)) \equiv abs(H) \\ abs(\lambda x.app(abs(H),x)) \arrow[ur, "\beta"] \arrow[dr, "\eta"'] & \\ & abs(H) \end{tikzcd}$$

Both critical pairs are trivially joinable.

Now we consider the combination of λ-calculus and "algebraic", i.e. first-order, reductions as in [3]. The first-order term $f(s_1,\ldots,s_n)$ translates to the *term* $app(\ldots(app(f,t_1),\ldots),t_n)$, where t_i is the translation of s_i. It is easy to see that the combination of β-reduction with the translation of algebraic term-rewriting systems does not generate any new critical pairs. In fact, we have the more general lemma:

Lemma 5.1 *Let R be a set of rules whose left-hand sides contain no abs and no subterm of the form $app(X,t)$ where X is free and t is arbitrary. If R is locally confluent, so is $R \cup \{\beta\}$.*

Comparing this with Theorem 2.3 by Breazu-Tannen [3] which states that if R is confluent, so is $R \cup \{\beta\}$, we find that the above lemma admits a larger class of rules R but requires termination of R to deduce confluence. The critical pair approach also explains why the addition of η to R may distroy confluence: new critical pairs may arise.

A well-known instance of R in Lemma 5.1 are the three rules for products with surjective pairing:

$$\begin{array}{rrcl} \pi_1: & \pi_1\langle S,T\rangle & \rightarrow & S \\ \pi_2: & \pi_2\langle S,T\rangle & \rightarrow & T \\ \pi: & \langle \pi_1 S, \pi_2 S\rangle & \rightarrow & S \end{array}$$

where

$$\begin{array}{rl} \langle\_,\_\rangle: & term \rightarrow term \rightarrow term \\ \pi_1,\pi_2: & term \rightarrow term \end{array}$$

Surjective pairing gives rise to the following two critical pairs with π_1 and π_2: $\langle S,T\rangle \leftarrow \langle\pi_1\langle S,T\rangle,\pi_2\langle S,T\rangle\rangle \rightarrow \langle S,\pi_2\langle S,T\rangle\rangle$ and $\langle S,T\rangle \leftarrow \langle\pi_1\langle S,T\rangle,\pi_2\langle S,T\rangle\rangle \rightarrow \langle\pi_1\langle S,T\rangle,T\rangle$; both reduce to $\langle S,T\rangle$. Hence the untyped λ-calculus with surjective pairing is locally confluent. This is independent of whether we consider β, η, or

$\beta + \eta$ because none of the pairing rules overlap with β or η. Confluence holds for terminating fragments, e.g. typable *terms*, but fails in general [11].

However, there is another formulation of pairs using the constant

$$split : \ (term \to term \to term) \to term \to term$$

instead of π_1 and π_2 and the single rule

$$split_\beta : \ split(F, \langle S, T\rangle) \ \to \ F(S,T)$$

instead of the two rules π_1 and π_2. This system is obviously locally confluent since there are no critical pairs. The constant π_i is now definable as $\lambda z.split(\lambda x_1, x_2.x_i, z)$ and $\pi_i(\langle S_1, S_2\rangle) \to S_i$ follows from $split_\beta$. Surjective pairing, however, does not hold: the $split_\beta$-normal form of $\langle \pi_1 S, \pi_2 S\rangle$, is $\langle split(\lambda x_1, x_2.x_1, S), split(\lambda x_1, x_2.x_2, S)\rangle$, which is not equivalent to S. This can be fixed with the additional rule

$$split_\eta : \ split(\lambda x, y.F(\langle x, y\rangle), P) \ \to \ F(P)$$

Surjective pairing is now derivable, but not by rewriting: not only is the left-hand side of $split_\eta$ not a pattern, but the combination of $split_\beta$ and $split_\eta$ is also non-confluent because of the following "critical pair":

$$\begin{tikzcd} & split(\lambda x, y.F(\lambda g.g(x,y)), P) \\ split(\lambda x, y.F(\lambda g.split(g, \langle x, y\rangle)), P) \arrow[ur, "split_\beta"] \arrow[dr, "split_\eta"'] & \\ & F(\lambda g.split(g, P)) \end{tikzcd}$$

This suggests adding a reduction between $split(\lambda x, y.F(\lambda g.g(x, y)), P)$ and $F(\lambda g.split(g, P))$. However, not only do we lose termination, no matter in which direction the reduction is oriented, but both sides are now so far removed from our patterns that a substantial extension of the theory is required to deal with it.

The same problems recur with disjoint unions, defined by the constants

$$\begin{aligned} inl, inr : \ & term \to term \\ case : \ & term \to (term \to term) \to (term \to term) \to term \end{aligned}$$

and the rules

$$\begin{aligned} case(inl(X), F, G) \ &\to \ F(X) \\ case(inr(X), F, G) \ &\to \ G(X) \\ case(U, \lambda x.F(inl(x)), \lambda x.F(inr(x))) \ &\to \ F(U) \end{aligned}$$

where the "critical pair" is

$$case(U, \lambda x.H(\lambda f, g.f(x)), \lambda x.H(\lambda f, g.g(x))) = H(\lambda f, g.case(U, f, g)),$$

generated by $case(U, \lambda x.H(\lambda f, g.case(inl(x), f, g)), \lambda x.H(\lambda f, g.case(inr(x), f, g)))$. The difficulty of obtaining confluent reductions for disjoint unions is also discussed by Dougherty [6].

5.2 First-Order Logic

This subsection discusses a number of normalization procedures for formulae in classical first-order logic. More precisely, we have two types *term* and *form* of terms and formulae, and the following constants:

$$\begin{aligned} \neg : &\ \mathit{form} \to \mathit{form} \\ \_\wedge\_, \_\vee\_ : &\ \mathit{form} \to \mathit{form} \to \mathit{form} \\ \forall, \exists : &\ (\mathit{term} \to \mathit{form}) \to \mathit{form} \end{aligned}$$

We allow ourselves the luxury of writing $\forall x.P(x)$ instead of $\forall(\lambda x.P(x))$. The symbols $\wedge$ and $\vee$, and $\forall$ and $\exists$ are completely dual to each other. Therefore we will only present half the rewrite rules for each system, the other half being the exact dual. All the systems terminate.

The *negation normal form* [7], where $\neg$ is only applied to atomic formulae, can be defined via the following rewrite system:

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

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

$$\begin{tikzcd} & \exists(P') \\ \neg\neg(\exists(P')) \arrow[ur, "\neg\neg"] \arrow[dr, "\neg\exists"'] & \\ & \neg\forall x.\neg P'(x) \end{tikzcd}$$

is joinable because $\neg\forall x.\neg P'(x) \to \exists x.\neg\neg P'(x) \to \exists x.P'(x) \equiv \exists(P')$. Hence the negation normal form is uniquely determined.

Prenex normal form [7] can also be described by a rewrite system containing the two rules

$$\begin{aligned} \forall\wedge : &\quad \forall(P') \wedge Q &\to&\quad \forall x.(P'(x) \wedge Q) \\ \wedge\forall : &\quad P \wedge \forall(Q') &\to&\quad \forall x.(P \wedge Q'(x)) \end{aligned}$$

This system is not confluent because $\forall(P') \wedge \forall(Q')$ gives rise to the critical pair $\forall x.(P'(x) \wedge \forall(Q')) = \forall x.(\forall(P') \wedge Q'(x))$ which has the two distinct normal forms $\forall x.\forall y.(P'(x) \wedge Q'(y))$ and $\forall x.\forall y.(P'(y) \wedge Q'(x))$. Commutativity of quantifiers needs to be taken into account as well.

The inverse operation *mini scoping*, i.e. pushing quantifiers inwards, is confluent:

$$\begin{aligned} \forall(P) &\to P \\ \forall x.(P'(x) \wedge Q'(x)) &\to \forall(P') \wedge \forall(Q') \\ \forall x.(P'(x) \vee Q) &\to \forall(P') \vee Q \\ \forall x.(P \vee Q'(x)) &\to P \vee \forall(Q') \end{aligned}$$

All critical pairs are joinable.

6 Future Extensions

The class of rewrite rules considered in this paper are only a fragment of a much larger class of higher-order rewrite rules allowing more general left-hand sides. Although Section 3 points out some problems with relaxing the restriction to patterns as left-hand sides, the rules for products and sums considered at the end of Section 5.1 require such a relaxation. Hence a corresponding extension of the theory seems highly desirable.

Acknowledgements

The author wishes to thank Dale Miller for help in debugging the pattern unification algorithm, Eugenio Moggi and Andrew Pitts for many discussions on topics related to this work, and the referees for their helpful comments.

References

[1] P. Aczel. A general Church-Rosser theorem. Technical report, University of Manchester, 1978.

[2] L. Bachmair. *Canonical Equational Proofs.* Research Notes in Theoretical Computer Science. Wiley and Sons, 1989.

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

[4] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier - The MIT Press, 1990.

[6] D. Dougherty. Some reduction properties of a lambda calculus with coproducts and recursive types. Technical report, Wesleyan University, 1990.

[7] J. Gallier. *Logic for Computer Science.* Harper & Row, 1986.

[8] J. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[9] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[10] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27:797–821, 1980.

[11] J. Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[12] D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

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

[14] G. Nadathur and D. Miller. An overview of λProlog. In *Proc. 5th Int. Logic Programming Conference*, pages 810–827, 1988.

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

[16] W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symbolic Computation*, 8:101–140, 1989.

A Unification of Higher-Order Patterns

Miller's original algorithm [13] deals with explicitly quantified unification problems. This section describes a simplified version which applies just to equalities between unquantified patterns. Unification proceeds essentially as in Huet's algorithm [9], except that projection and imitation coincide, and that flexible-flexible pairs admit most general unifiers. The presentation is inspired by the work of Snyder and Gallier [16]. The algorithm is expressed by rewrite rules on pairs $\langle L, \sigma\rangle$, where L is the list[3] of unification problems still to be solved and σ is the fragment of the solution computed so far. Solving such a pair means rewriting it to normal form. If the normal form is $\langle [], \sigma\rangle$, σ is the solution to the initial problem. Otherwise the initial problem has no solution.

Unification problems are unordered pairs $s =^? t$ where s and t are patterns. All references to types are omitted because the algorithm works for both typed and untyped patterns. To ease notation we work with α-equivalence classes of terms. After each rewrite step the problem at the head of L is first put into β-normal form and then the binders are adjusted to be of equal length: the problem $\lambda\overline{x_k}.a(\overline{s_m}) =^? \lambda\overline{x_{k+n}}.t$ is η-expanded to $\lambda\overline{x_{k+n}}.a(\overline{s_m}, x_{k+1}, \ldots, x_{k+n}) =^? \lambda\overline{x_{k+n}}.t$. Note that in the typed case this expansion is valid because both terms must be of the same type.

Solutions to trivial unification problems are propagated:

$$\langle(\lambda\overline{x_k}.F(\overline{x_k}) \overset{?}{=} t) :: L, \sigma\rangle \implies \langle\{F \mapsto t\}(L), \{F \mapsto t\} \circ \sigma\rangle$$

if $F \notin \mathcal{FV}(t)$. To guarantee termination, this rule has priority over the following ones. Because the remaining rules do not modify the substitution component, only the transformation of the unification problems is shown below.

[3]Standard ML list notation is used.

Rigid-rigid pairs with identical heads are decomposed:

$$(\lambda\overline{x_k}.a(\overline{s_n}) \overset{?}{=} \lambda\overline{x_k}.a(\overline{u_n})) :: L \implies [\lambda\overline{x_k}.s_1 \overset{?}{=} \lambda\overline{x_k}.u_1, \ldots, \lambda\overline{x_k}.s_n \overset{?}{=} \lambda\overline{x_k}.u_n]@L$$

if $a \in C \cup \{\overline{x_k}\}$.

Imitation and projection coincide:

$$(\lambda\overline{x_k}.F(\overline{y_n}) \overset{?}{=} \lambda\overline{x_k}.a(\overline{s_m})) :: L \implies [F \overset{?}{=} \lambda\overline{y_n}.a(\overline{H_m(\overline{y_n})}), \lambda\overline{x_k}.F(\overline{y_n}) \overset{?}{=} \lambda\overline{x_k}.a(\overline{s_m})]@L$$

if $a \in C \cup \{\overline{y_n}\}$, $F \notin \mathcal{FV}(\overline{s_m})$, and the H_m are new.

Flexible-flexible pairs with identical heads:

$$(\lambda\overline{x_k}.F(\overline{s_n}) \overset{?}{=} \lambda\overline{x_k}.F(\overline{u_n})) :: L \implies (F \overset{?}{=} \lambda\overline{y_n}.H(\overline{v_p})) :: L$$

if $\overline{y_n} = \overline{s_n}\downarrow_\eta$, $\overline{z_n} = \overline{u_n}\downarrow_\eta$, $\{\overline{v_p}\} = \{y_i \mid y_i = z_i\}$, and H is new.

Flexible-flexible pairs with distinct heads:

$$(\lambda\overline{x_k}.F(\overline{s_n}) \overset{?}{=} \lambda\overline{x_k}.G(\overline{u_m})) :: L \implies [F \overset{?}{=} \lambda\overline{y_n}.H(\overline{v_p}), G \overset{?}{=} \lambda\overline{z_m}.H(\overline{v_p})]@L$$

if $\overline{y_n} = \overline{s_n}\downarrow_\eta$, $\overline{z_n} = \overline{u_n}\downarrow_\eta$, $F \neq G$, $\{\overline{v_p}\} = \{\overline{y_n}\} \cap \{\overline{z_m}\}$, and H is new.

Inverting the preconditions to the above rules yields the following two failure cases:

$$\lambda\overline{x_k}.a(\overline{s_n}) \overset{?}{=} \lambda\overline{x_k}.b(\overline{u_m})$$

where $a, b \in C \cup \{\overline{x_k}\}$ and $a \neq b$, and

$$\lambda\overline{x_k}.F(\overline{y_n}) \overset{?}{=} \lambda\overline{x_k}.a(\overline{s_m})$$

where $F \in \mathcal{FV}(\overline{s_m})$ or $a \in \{\overline{x_k}\} - \{\overline{y_n}\}$.

The following theorem holds for both typed and untyped patterns:

Theorem A.1 *The above set of rules yields a correct, complete, and terminating unification algorithm for higher-order patterns. A list of unification problems L has a solution iff $\langle L, \{\}\rangle \Longrightarrow^* \langle [], \sigma\rangle$, in which case σ is a most general unifier for L.*

Correctness and completeness follow by the same arguments used by Miller [13]. The termination proof, however, is different and relies on the fact that L is a list and not a multiset. Although Miller also uses lists, his algorithm can be rephrased in terms of multisets without loss of termination. The reason is the different treatment of flexible-rigid pairs: Miller manipulates the rigid part until the problem can be solved in one step, whereas the above formulation solves the problem in layers, introducing new variables on the way.