January 1992

Polymorphic Rewriting Conserves Algebraic Confluence

Val Tannen
University of Pennsylvania, val@cis.upenn.edu

Jean H. Gallier
University of Pennsylvania, jean@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-37.
Revised: January 1992

Polymorphic Rewriting Conserves Algebraic Confluence

Abstract

We study combinations of many-sorted algebraic term rewriting systems and polymorphic lambda term rewriting. Algebraic and lambda terms are mixed by adding the symbols of the algebraic signature to the polymorphic lambda calculus, as higher-order constants. We show that if a many-sorted algebraic rewrite system R has the Church-Rosser property (is confluent), then R + β + type-β + type-η rewriting of mixed terms has the Church-Rosser property too. η reduction does not commute with algebraic reduction, in general. However, using long normal forms, we show that if R is canonical (confluent and strongly normalizing) then equational provability from R + β + η + type-β + type-η is still decidable.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-90-37.
Revised: January 1992

Polymorphic Rewriting Conserves Algebraic Confluence

MS-CIS-90-37
LOGIC & COMPUTATION 21

Val Breazu-Tannen
Jean Gallier

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

Revised
January 1992

Polymorphic Rewriting Conserves Algebraic Confluence [1]

Val Breazu-Tannen[2] *Jean Gallier*[3]

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd St., Philadelphia, PA 19104, USA

Abstract. We study combinations of many-sorted algebraic term rewriting systems and polymorphic lambda term rewriting. Algebraic and lambda terms are mixed by adding the symbols of the algebraic signature to the polymorphic lambda calculus, as higher-order constants.

We show that if a many-sorted algebraic rewrite system R has the Church-Rosser property (is confluent), then R + β + type-β + type-η rewriting of mixed terms has the Church-Rosser property too.

η reduction does not commute with algebraic reduction, in general. However, using long normal forms, we show that if R is canonical (confluent and strongly normalizing) then equational provability from R + β + η + type-β + type-η is still decidable.

1 Introduction

From a very general point of view, this paper is about the interaction between "first-order computation" modeled by algebraic rewriting, and "higher-order polymorphic computation" modeled by reduction in the Girard-Reynolds polymorphic lambda calculus. Our results permit us to conclude that this interaction is quite smooth and pleasant.

Changing the perspective, we regard algebraic rewrite systems as tools for the proof-theoretic analysis of algebraic equational theories, and we recall that such algebraic theories are used to model data type specifications [EM85]. Then, our results continue to confirm a thesis put forward in a series of papers [MR86, BTM87, BT88] , namely that *strongly normalizing type disciplines* interact nicely with algebraic data type specifications.

The preservation of the confluence of algebraic rewriting is a case in point. We show in this paper that the very powerful, impredicative, but strongly normalizing, polymorphic type discipline yields confluent rewriting when combined with confluent algebraic rewriting. In contrast, this fails for type disciplines which allow the type-checking of *fixed points*, as in lambda calculi with recursive types, in particular in the untyped lambda calculus. Klop has shown that the untyped lambda

[1] To appear in **Information and Computation**
[2] Partially supported by ONR Grant NOOO14-88-K-0634 and by ARO Grant DAAG29-84-K-0061
[3] Partially supported by ONR Grant NOOO14-88-K-0593.

calculus enriched with surjective pairing reduction does not have the Church-Rosser (CR) property (see [Klo80], or [Bar84], pp. 403–407; the proof uses Turing's fixed point combinator), even though the rewrite system consisting of the surjective pairing rules alone is Church-Rosser, and, of course, β-reduction in isolation is CR. Another counterexample can be adapted from [BTM87], see [BT88]. We present here a further simplification observed by J. W. Klop (personal communication).

Consider the following algebraic rewrite system, call it R. There is one sort int, the signature is

$$minus : int \to int \to int \qquad succ : int \to int \qquad 0\,,\ 1 : int$$

and the rules are

$$\begin{aligned} minus\ x\ x &\longrightarrow 0 \\ minus\ (succ\ x)\ x &\longrightarrow 1\,. \end{aligned}$$

(We write the algebraic terms in *curried* form, anticipating their mixing with lambda terms.)

This algebraic system has the CR property (use Newman's Lemma [New42]). However, the CR property *fails* for βR-reduction on typed lambda terms with *recursive types* (in particular untyped lambda terms) constructed using also symbols from R's signature.

Indeed, let ξ be a type such that $\xi = \xi \to int$ and let

$$\Phi \stackrel{\text{def}}{=} (\,\lambda x{:}\xi.\ succ\,(x x)\,)\,(\,\lambda x{:}\xi.\ succ\,(x x)\,)\,.$$

We see that Φ type-checks with type int and is a fixed point of $succ$ in the sense that $\Phi \xrightarrow{\beta} succ\ \Phi$. Thus, we have

$$0 \xleftarrow{R} minus\ \Phi\ \Phi \xrightarrow{\beta} minus\ (succ\ \Phi)\ \Phi \xrightarrow{R} 1$$

All these counterexamples exploit the capability of expressing fixed points. Because of the normalization property, no such fixed points can be expressed in the polymorphic lambda calculus ($\lambda^{\forall}$). And, in fact, we make essential use of the normalization property to prove the main result of this paper, (see section 4) which states that combining a confluent many-sorted algebraic rewrite system with almost all kinds (except η) of polymorphic term reduction notions gives a system that, globally, is confluent.

A brief summary of the technical setting for our result goes as follows. Given a many-sorted signature Σ, we construct *mixed* lambda terms with the sorts of Σ as constant "base" types and from the symbols in Σ, seen, by currying, as higher-order constants. Then, given a set R of rewrite rules between algebraic Σ-terms, we show that if R is CR on algebraic Σ-terms, then $R + \beta +$ type-β + type-η rewriting of mixed terms has the Church-Rosser property too. (Notice the absence of η; a counterexample appears in section 4.) An obvious, but important, feature of R-rewriting on mixed terms is that this is done such that the variables occurring in the algebraic rules can be instantiated with any mixed terms, as long as they are of the same "base" type as the variables they replace.

Our result and its proof are direct generalizations of the corresponding result for the simply typed lambda calculus presented in [BT88]. However, since the publication of [BT88], we have found an error in the proof of one of its lemmas (specifically lemma 2.2) used there for the confluence result.

In this paper we correct the error, and generalize the statement of the lemma—from simply typed normal forms to arbitrary polymorphic terms (see theorem 3.19).

We compare this result with those of [Toy87] and [Klo80]. Toyama shows that the direct sum of two CR algebraic rewriting systems is also CR. For the direct sum, the two components are required to have disjoint signatures. In our case, note that while the symbols of the algebraic signature do not play any special role in defining β-reduction, there is one "operation" which is implicit in algebraic rewriting and which is therefore shared with β-reduction, namely *application*, and indeed, Toyama's methods do not seem to help in this situation. Our putting together of an algebraic rewrite system and a lambda calculus is more like Klop's direct sum of *combinatory reduction systems* for which, as shown in [Klo80], preservation of the CR property fails, in general, (see the examples above). Klop proves preservation of CR under certain restrictions, but he keeps the untyped lambda calculus as one of the components and imposes the restrictions on the algebraic reduction rules. In contrast, our algebraic reduction rules are totally arbitrary, but we restrict the lambda terms using the polymorphic type discipline. [4]

Some related work has been done since [BT88] appeared. Dougherty [Dou91] shows that our reduction mapping technique (see section 4) can also be used to show conservation of CR when one adds algebraic rewriting to β-reduction of strongly normalizing terms of the untyped lambda calculus. It is not clear how one could, from such a result about untyped terms, directly derive the main result of this paper, or even a weaker version of it involving just one-sorted algebraic theories. Working in a different direction, Howard and Mitchell [HM90] impose restrictions on the algebraic rewrite systems similar to those used in [Klo80], and show conservation of CR when such rewriting is added to the simply typed lambda calculus enriched with fixed point operators.

Our result about CR preservation is relevant to the implementation of functional programming languages, especially using parallel reduction strategies (see [Hud86] for a survey). Since it guarantees that results are independent of the computational strategy, the Church-Rosser property is the theoretical foundation for parallel evaluation. For functional languages based on the untyped lambda calculus (such as SCHEME [AS85]) CR depends on the choice of the first-order computational rules. Useful optimizations such as $(x - x) \longrightarrow 0$ and $(succ(x) - x) \longrightarrow 1$ (see the counterexample above) or $(\textbf{if } b \textbf{ then } x \textbf{ else } x) \longrightarrow x$ (see [Klo80]) are ruled out. Our result shows that, in contrast, *strongly typed* functional languages (such as ML [GMW79] and Miranda [Tur85]) are completely flexible from this point of view. Beware: even typed functional languages feature *recursion* which causes the failure of CR just like the untyped fixed points do. The difference is that in languages in which the use of recursion can be decidably isolated one can identify the chunks of program for which CR holds and parallel-execute them. This is not the case in untyped languages where non-typable "hacks" may hide the failure of CR.

Combining our result with the one on strong normalization in a companion paper [BTG91], we obtain the following: if R is canonical (confluent and strongly normalizing) on algebraic terms, then $R + \beta +$ type-$\beta +$ type-η is canonical on mixed terms. Again, we should point out that even

[4] In the presence of types, the surjective pairing rules must be postulated for every pair of types, which takes us out of the framework of algebraic rewrite systems. Nonetheless, it is still true that the simply typed lambda calculus with product types and surjective pairing has the CR property [Pot81]. The weak CR property is easy to check, hence, by Newman's Lemma [New42], the CR result also follows from the fact that the typed lambda calculus with surjective pairing is strongly normalizing (SN) [LS86]. (We also know of three unpublished proofs of this SN result, all obtained independently [dV82, Ber84, Dou86].)

direct sums of canonical systems are not necessarily canonical, as was shown by Barendregt and Klop [Klo87].

The reader may wonder what happens with η-reduction. An example is given in section 4 which shows that η-reduction does not commute even with the simplest kind of algebraic reduction. We do not regard this as a significant fact since the computational interpretation of η-reduction is quite unclear. However, η, regarded as an *equational axiom*, may be useful when reasoning about programs. In view of this, we examine the problem of deciding equational reasoning from $R + \beta + \eta +$ type-β + type-η. We show in section 5 that if R is canonical then such reasoning is still decidable.

2 Mixing algebra and polymorphic lambda calculus

This section is devoted to developing the notation used in the paper. Our notation will depart from that of recent presentations of the polymorphic lambda calculus [BMM90, BTC88]. These papers exhibit a notation using *typing judgements* or *typing relationships*, based on the ideas of [Rey74]. Such a notation allows elegant presentations of some of the equational proof systems and of the set-theoretic and categorical models. We feel however that it does not best support the intuition needed in proofs depending heavily on the *combinatorics* of terms. For example, the analysis of the reduction mechanisms is made more cumbersome by the presence of type assignments (contexts). Ideally, we would like a notation as simple as that developed for the untyped lambda calculus in [Bar84]. As demonstrated very well by Statman's work, the traditional notation for the simply typed lambda calculus (*e.g.*, [Fri75]) also helps the combinatorial intuition [Sta82]. This notation uses variables which come from an a priori type-indexed collection: therefore a variable has the same type everywhere it is used. Can the same be done in the polymorphic lambda calculus?

In fact, this is the notation used by Girard [Gir72] and later adopted in [Sta81, FLO83]. It poses the following conceptual problem: if x is a variable of type σ which occurs bound in a term M how do we define the result of a type substitution on M which might modify σ? We would rather avoid this problem, but, of course, we also want to avoid the use of judgements. The idea is to fix the types of the free variables, but only within each term in which they occur. Some checks will be needed in the definition of terms in order for this to be done consistently. Based on this idea and starting with an arbitrary many-sorted algebraic signature, we will define *mixed terms i.e.,* polymorphic lambda terms constructed with the symbols of the signature seen as higher-order constants, as follows.

Let S be a set of *sorts* and let Σ an S-sorted algebraic signature. Each function symbol $f \in \Sigma$ has an *arity*, which is a string $s_1 \cdots s_n \in S^*$, $n \geq 0$, and a *sort* $s \in S$ intending to symbolize a possibly heterogenous operation which takes arguments of sorts (in order) $s_1, \ldots, s_n$ and returns a result of sort s.

Definition 2.1 *(Types)*
Let $\mathcal{V}$ be a countably infinite set of *type variables*. The set $\mathcal{T}$ of *type expressions (types)* is defined by the following grammar:

$$\sigma ::= s \mid t \mid \sigma \rightarrow \sigma \mid \forall t.\, \sigma$$

where s ranges over S, and t over $\mathcal{V}$.

Therefore, the "base" types are exactly the sorts of the signature. Free and bound variables are defined in the usual way. We denote by $FTV(\sigma)$ the set of type variables which are free in σ. We will identify the type expressions which differ only in the name of the bound variables, and then adopt Barendregt's variable convention [Bar84]: in a given mathematical context, such as a definition or a proof, all bound variables are chosen to be different from all free variables.

A *type substitution* is a partial map $\theta : \mathcal{V} \longrightarrow \mathcal{T}$ with finite domain. In agreement with the variable convention, it is always assumed that the variables belonging to the domains of the substitutions differ from the bound variables used in the same mathematical context. The result of applying θ to σ (its straightforward definition is omitted) is denoted by $\sigma[\theta]$ and, if θ is the identity everywhere except $\theta(t) = \tau$, also by $\sigma[\tau/t]$.

We give now a simultaneous inductive definition of the terms M, their types, their set of free variables $FV(M)$, and the types those free variables have *in* M.

Definition 2.2 *(Terms)*
Let $\mathcal{X}$ be a countably infinite set of *(term) variables.*

Variables. For any $x \in \mathcal{X}$, and any $\sigma \in \mathcal{T}$, the pair $\langle x, \sigma \rangle$ is a term of type σ with exactly one free variable, x, $(FV(\langle x, \sigma \rangle) \stackrel{\text{def}}{=} \{x\})$ whose type in $\langle x, \sigma \rangle$ is σ.

Constants. For any $f \in \Sigma$, f is a term of type $s_1 \rightarrow \cdots \rightarrow s_n \rightarrow s$ (where $s_1 \cdots s_n$ and s are the arity and the sort of f) without free variables $(FV(f) \stackrel{\text{def}}{=} \emptyset)$.

Application. If M is a term of type $\sigma \rightarrow \tau$ and N is a term of type σ, and each common free variable of M and N has the same type in M and N, then (MN) is a term of type τ with $FV(MN) \stackrel{\text{def}}{=} FV(M) \cup FV(N)$ and such that the type of each free x in MN is the same as the type of x in M or else in N.

Abstraction. For any $x \in \mathcal{X}$, and any $\sigma \in \mathcal{T}$, if M is a term of type τ such that if $x \in FV(M)$ then x has type σ in M, then $(\lambda x{:}\sigma.\, M)$ is a term of type $\sigma \rightarrow \tau$, with $FV(\lambda x{:}\sigma.\, M) \stackrel{\text{def}}{=} FV(M) \backslash \{x\}$ and whose free variables have the same types as in M.

Type application. For any $\tau \in \mathcal{T}$, if M is a term of type $\forall t.\sigma$ then $(M\tau)$ is a term of type $\sigma[\tau/t]$, whose free variables are the same as those of M and have the same types as in M.

Type abstraction. For any $t \in \mathcal{V}$, if M is a term of type σ such that for any $x \in FV(M)$, t is not free in the type of x in M, then $(\lambda t.\, M)$ is a term of type $\forall t.\, \sigma$ whose free variables are the same as those of M and have the same types as in M.

We denote by Λ the set of all terms. This kind of definition produces only "well-typed" terms (compare with the approach using "raw" terms and type-checking judgements [BTC88]). We will sometimes abbreviate "the type of M is σ" as $M : \sigma$.

Once past the stage of formal definitions, we will never need to use the cumbersome notation $\langle x, \sigma \rangle$ for terms which consist of just a variable. The type will always be understood from the

mathematical context in which the term is used so we can omit it and write simply x. Moreover, we will make the convention that when we write (MN), it is understood that M and N satisfy the conditions in the (Application) clause above, and thus (MN) is a term. Similarly for the other term constructions. Of course, we adopt the usual notational conventions that facilitate using less parantheses, such as "application associates to the left", *etc.,* [Bar84].

Bound type variables and bound term variables in terms are defined as usual. We identify terms which differ only in the name of bound type variables or bound term variables, and we adopt again Barendregt's variable convention (see above).

Definition 2.3 *(Free type variables of a term)*
The set of *free type variables of a term*, notation $FTV(M)$, is defined as follows:

$$\begin{aligned}
FTV(\langle x,\sigma\rangle) &\stackrel{\text{def}}{=} FTV(\sigma)\\
FTV(f) &\stackrel{\text{def}}{=} \emptyset\\
FTV(MN) &\stackrel{\text{def}}{=} FTV(M)\cup FTV(N)\\
FTV(\lambda x{:}\sigma.\, M) &\stackrel{\text{def}}{=} FTV(\sigma)\cup FTV(M)\\
FTV(M\tau) &\stackrel{\text{def}}{=} FTV(M)\cup FTV(\tau)\\
FTV(\lambda t.\, M) &\stackrel{\text{def}}{=} FTV(M)\backslash\{t\}
\end{aligned}$$

Definition 2.4 *(Type substitution in a term)*
The result of applying a type substitution θ to a term M, notation $M[\theta]$, is defined as follows:

$$\begin{aligned}
\langle x,\sigma\rangle[\theta] &\stackrel{\text{def}}{=} \langle x,\sigma[\theta]\rangle\\
f[\theta] &\stackrel{\text{def}}{=} f\\
(MN)[\theta] &\stackrel{\text{def}}{=} M[\theta](N[\theta])\\
(\lambda x{:}\sigma.\, M)[\theta] &\stackrel{\text{def}}{=} \lambda x{:}\sigma[\theta].\, M[\theta]\\
(M\tau)[\theta] &\stackrel{\text{def}}{=} M[\theta](\tau[\theta])\\
(\lambda t.\, M)[\theta] &\stackrel{\text{def}}{=} \lambda t.\, M[\theta]
\end{aligned}$$

One can check that $M[\theta]$ is always defined, that it is a term, that its type is $\sigma[\theta]$, where σ is the type of M, that $FV(M[\theta]) = FV(M)$, and that the type of each free variable z in $M[\theta]$ is $\tau[\theta]$ where τ is the type of z in M.

Definition 2.5 *(Term substitution in a term)*
A *term substitution* is a partial map $\varphi : \mathcal{X} \longrightarrow \Lambda$ whose domain, denoted $dom\varphi$, is finite. As for type substitutions, it is always assumed that the variables belonging to the domains of the substitutions differ from the bound variables used in the same mathematical context. The result of applying a term substitution φ to a term M, notation $M[\varphi]$, is defined, when possible, as follows:

$$\begin{aligned}
\langle x,\sigma\rangle[\varphi] &\stackrel{\text{def}}{=} \textbf{if } \varphi(x) : \sigma \textbf{ then } \varphi(x) \textbf{ else undefined}\\
f[\varphi] &\stackrel{\text{def}}{=} f
\end{aligned}$$

$$
\begin{aligned}
(MN)[\varphi] &\stackrel{\text{def}}{=} M[\varphi](N[\varphi]) \text{ (if defined)} \\
(\lambda x{:}\sigma.\, M)[\varphi] &\stackrel{\text{def}}{=} \lambda x{:}\sigma.\, M[\varphi] \text{ (if defined)} \\
(M\tau)[\varphi] &\stackrel{\text{def}}{=} M[\varphi]\tau \text{ (if defined)} \\
(\lambda t.\, M)[\varphi] &\stackrel{\text{def}}{=} \lambda t.\, M[\varphi] \text{ (if defined)}
\end{aligned}
$$

Thus, $M[\varphi]$ is not always defined, but when it is, its type is the same as that of M, and one can also give a characterization of the set of free variables of $M[\varphi]$ and their types in $M[\varphi]$ (this is a bit tedious to state but straightforward). Again, we will make the convention that whenever we write $M[\varphi]$, it is understood that M and φ satisfy sufficient conditions for $M[\varphi]$ to be defined.

We also denote by $[M_1/x_1, \ldots, M_n/x_n]$ the substitution φ such that $dom\varphi = \{x_1, \ldots, x_n\}$ and $\varphi(x_i) = M_i$, (hence we denote $M[\varphi]$ by $M[M_1/x_1, \ldots, M_n/x_n]$).

We have followed Barendregt [Bar84] in our definitions of substitutions. As pointed out in [Bar84], appendix C, the strictly rigorous approach is to define substitution *before* identifying expressions which differ only in the name of bound variables (α-congruent expressions) and then show that substitution is compatible with α-congruence, hence is well-defined on α-congruence classes. However, when this is done, the resulting substitution operation will coincide with the one given above in a manner that exploits the variable convention.

In defining term rewriting, it is convenient to use contexts [Bar84]. We will only need contexts with exactly one hole. Let $\bigcirc$ be a new symbol, distinct from both the symbols in Σ and from the variables.

Definition 2.6 *(Contexts)*
Let $\omega \in \mathcal{T}$. *Contexts* with a hole of type ω, their types, their set of free variables, and the types those free variables have, are given by a simultaneous inductive definition using the same clauses we gave for terms (definition 2.2), plus *exactly one* use of the following clause

Hole. The pair $\langle \bigcirc, \omega \rangle$ is a context of type ω and with no free variables.

Instead of "C is a context" we will often write just $C[\,]$.

Definition 2.7 *(Placing a term in a context)*
The result of placing a term M in $C[\,]$, notation $C[M]$, is defined as follows:

$$
\begin{aligned}
\langle \bigcirc, \omega \rangle[M] &\stackrel{\text{def}}{=} \textbf{if } M : \omega \textbf{ then } M \textbf{ else undefined} \\
(CN)[M] &\stackrel{\text{def}}{=} C[M]\, N \text{ (if defined)} \\
(NC)[M] &\stackrel{\text{def}}{=} N\, C[M] \text{ (if defined)} \\
(\lambda x{:}\sigma.\, C)[M] &\stackrel{\text{def}}{=} \lambda x{:}\sigma.\, C[M] \text{ (if defined)} \\
(C\tau)[M] &\stackrel{\text{def}}{=} C[M]\tau \text{ (if defined)} \\
(\lambda t.\, C)[M] &\stackrel{\text{def}}{=} \lambda t.\, C[M] \text{ (if defined)}
\end{aligned}
$$

Thus, $C[M]$ is not always defined, but when it is, one can see that it is a term, that its type is the same as that of C, and that one can also give a characterization of the set of free variables of $C[M]$ and their types in $C[M]$ (this is again a bit tedious to state but straightforward). Yet again, we will make the convention that whenever we write $C[M]$, it is understood that $C[\,]$ and M satisfy sufficient conditions for $C[M]$ to be defined.

It is important to note that contexts are *not* considered modulo α-congruence. An essential feature of contexts is that a free variable of M may become bound in $C[M]$. However, $C[M]$ is a term and thus it is again considered modulo α-congruence. Note also that for $C[M]$ to be defined, it is not sufficient that M have the same type as the hole in $C[\,]$. For example, if x is free in M with type σ_1 and we want to place M in a context of the form $\lambda x{:}\sigma_2.C$, and, moreover, x is still free in $C[M]$, then we must have $\sigma_1 \equiv \sigma_2$.

We are now ready to define the usual *reduction relations.*

Definition 2.8 *(Reduction)*

(β-reduction) $M \xrightarrow{\beta} N$ iff
there exist $C[\,], x, \sigma, X, Y$ such that $M \equiv C[(\lambda x{:}\sigma.\, X)Y]$ and $N \equiv C[X[Y/x]]$.

(η-reduction) $M \xrightarrow{\eta} N$ iff
there exist $C[\,], x, \sigma, Z$, where $x \notin FV(Z)$, such that $M \equiv C[\lambda x{:}\sigma.\, Zx]$ and $N \equiv C[Z]$.

(type-β reduction) $M \xrightarrow{\mathcal{T}\beta} N$ iff
there exist $C[\,], t, \tau, X$ such that $M \equiv C[(\lambda t.\, X)\tau]$ and $N \equiv C[X[\tau/t]]$.

(type-η reduction) $M \xrightarrow{\mathcal{T}\eta} N$ iff
there exist $C[\,], t, Z$, where $t \notin FTV(Z)$, such that $M \equiv C[\lambda t.\, Zt]$ and $N \equiv C[Z]$.

Clearly, if $M \xrightarrow{\rho} N$, where ρ is any one of $\beta, \eta, \mathcal{T}\beta$ or $\mathcal{T}\eta$, then M and N have the same type. Moreover, $FV(M) \supseteq FV(N)$ and any common free variable has the same type in both terms. Let

$$\xrightarrow{\lambda^\forall} \stackrel{\text{def}}{=} \xrightarrow{\beta} \cup \xrightarrow{\eta} \cup \xrightarrow{\mathcal{T}\beta} \cup \xrightarrow{\mathcal{T}\eta}$$

and

$$\xrightarrow{\lambda^-} \stackrel{\text{def}}{=} \xrightarrow{\beta} \cup \xrightarrow{\mathcal{T}\beta} \cup \xrightarrow{\mathcal{T}\eta}$$

It is well-known that both $\lambda^\forall$-reduction and λ^--reduction are canonical (*i.e.*, strongly normalizing and confluent) on all terms. We denote by $\lambda^\forall nf(X)$ and $\lambda^- nf(X)$ the corresponding normal forms of an arbitrary term X.

Next, we will introduce our notation for algebraic terms and algebraic rewrite rules. There is a well-known transformation, known as *currying*, that maps algebraic Σ-terms into applicative (mixed) terms. This transformation is an injection. In view of that, we will use directly the curried notation.

Definition 2.9 *(Algebraic terms)*
Algebraic terms A, their sorts, their set of occurring variables, $V(A)$, and the sorts those variables have *in* A are defined by simultaneous induction, as follows.

Variables. For any $x \in \mathcal{X}$, and any $s \in S$, the pair $\langle x, s\rangle$ is an algebraic term of sort s with exactly one variable, x, $(V(\langle x, s\rangle) \stackrel{\text{def}}{=} \{x\})$ whose sort in $\langle x, s\rangle$ is s.

Application. If $f \in \Sigma$ has arity $s_1 \cdots s_n$ and sort s, and if $A_1, \ldots, A_n$ are algebraic terms of sorts $s_1, \ldots, s_n$ respectively, and such that any variable in $V(A_1) \cup \cdots \cup V(A_n)$ has the same sort in all the terms in which it occurs, then $(\cdots (f\, A_1) \cdots A_n)$ is an algebraic term, of sort s, with $V(f\, A_1 \cdots A_n) \stackrel{\text{def}}{=} V(A_1) \cup \cdots \cup V(A_n)$ and such that the sort of each x in $f\, A_1 \cdots A_n$ is the same as the sort of x in the A_i's in which it occurs.

Clearly, any algebraic term A is a term, its type is its sort, $FV(A) = V(A)$, and the types its free variables have in A are the sorts they have in A.

Definition 2.10 *(Algebraic rewrite rules)*
An *algebraic rewrite rule* is an ordered pair of algebraic terms, written $A \rightharpoonup B$, such that

- A and B have the same sort,
- $FV(A) \supseteq FV(B)$ and any common variable has the same sort in both terms, and
- A is not a variable.

Each algebraic rewrite rule determines a reduction relation on *all* mixed terms, not only the algebraic ones.

Definition 2.11 *(Algebraic reduction)*
Given an algebraic rewrite rule $r \equiv A \rightharpoonup B$, we define a reduction relation on terms as follows

$$M \stackrel{r}{\longrightarrow} N \quad \text{iff}$$

there exists a context C and a term substitution φ such that

$$M \equiv C[A[\varphi]] \qquad N \equiv C[B[\varphi]]$$

Note that the range of φ is *not* restricted to algebraic terms. Clearly, if $M \stackrel{r}{\longrightarrow} N$ then M and N have the same type. Moreover, $FV(M) \supseteq FV(N)$ and any common free variable has the same type in both terms. One can easily check the following fact.

Lemma 2.12 *If A is algebraic, r is an algebraic rewrite rule, and $A \stackrel{r}{\longrightarrow} M$, then M is algebraic.*

Thus, we can talk about algebraic rewriting on algebraic terms. It is easy to see that currying establishes the expected relation between many-sorted algebraic rewriting of Σ-terms [MG85] and our definition of algebraic rewriting. Indeed, for any many-sorted Σ-rewrite rule $m \equiv p \to p'$ and any many-sorted Σ-terms q, q'

$$q \xrightarrow{m} q' \quad \text{iff} \quad curry(q) \xrightarrow{c(m)} curry(q')$$

where $c(m) \equiv curry(p) \to curry(p')$.

Definition 2.13
Let R be a set of algebraic rewrite rules. Define the following reduction relations on terms:

$$\xrightarrow{R} \stackrel{\text{def}}{=} \bigcup_{r \in R} \xrightarrow{r} \ , \quad \xrightarrow{\lambda^\forall R} \stackrel{\text{def}}{=} \xrightarrow{\lambda^\forall} \cup \xrightarrow{R} \ , \quad \xrightarrow{\lambda^- R} \stackrel{\text{def}}{=} \xrightarrow{\lambda^-} \cup \xrightarrow{R} \ .$$

For any reduction relation $\xrightarrow{\rho}$, we will denote by $\xrightarrow{\rho}\!\!\!\!\!\rightarrow$ its reflexive and transitive closure, by $\xleftarrow{\rho}$ its converse, and by $\xleftarrow{\rho}\!\!\!\!\!\leftarrow$ the converse of $\xrightarrow{\rho}\!\!\!\!\!\rightarrow$. Moreover, the equivalence relation generated by $\xrightarrow{\rho}$ is called the *ρ-convertibility* relation while $\xleftrightarrow{\rho} \stackrel{\text{def}}{=} \xrightarrow{\rho} \cup \xleftarrow{\rho}$ is called the *one-step ρ-convertibility* relation. Clearly, the ρ-convertibility relation is the same as the reflexive and transitive closure of $\xleftrightarrow{\rho}$ and also the same as the reflexive and transitive closure of $\xrightarrow{\rho}\!\!\!\!\!\rightarrow \cup \xleftarrow{\rho}\!\!\!\!\!\leftarrow$.

Finally, we state precisely our main result:

(Conservation of Confluence.) If $\xrightarrow{R}$ is confluent on algebraic terms then $\xrightarrow{\lambda^- R}$ is confluent on all terms.

3 Algebraic rewriting of higher-order terms

In this section, we show that if algebraic reduction has the Church-Rosser property on algebraic terms then it also has this property on arbitrary mixed terms. The main result of this section is the following claim, proved later as theorem 3.19.

Claim. If $\xrightarrow{R}$ is confluent on algebraic terms then $\xrightarrow{R}$ is confluent on all terms.

The proof is surprisingly involved, and requires a number of auxiliary lemmas. To understand where the difficulty lies, we begin sketching the proof.

We show by induction on the size of M that R-confluence holds from M. The only case in which the induction hypothesis does not immediately apply is the case of an application term. For an application term $M \equiv H\, T_1 \cdots T_k$ such that H is an abstraction, a type abstraction, a variable, or a constant which takes $> k$ arguments, each R-reduction out of M is completely inside H or inside one of the T_i's. By the induction hypothesis, confluence holds from each of these, and it is easy to see that this implies that confluence holds from M.

This only leaves us with the case when H is a constant which takes exactly k arguments, in which case the type of M is a sort. We need to analyze algebraic reductions on such terms, in particular

to separate "trunk" (close to the "root" of terms) algebraic reductions from other reductions. However, this analysis is quite subtle because non-linear rewrite rules (i.e., the left-hand side of the rule contains multiple occurrences of some variable) can cause problems, as example 3.4 will show. But first, we develop the necessary technical tool, the notion of trunk decomposition (Toyama defines a similar concept in [Toy87]).

Definition 3.1 *(Algebraic trunk decomposition)*
An *algebraic trunk decomposition* of a term M consists of an algebraic term A (the "trunk") and a term substitution φ such that $M \equiv A[\varphi]$, $dom\varphi = FV(A)$, each variable in A occurs only once, and for all $x \in FV(A)$ the term $\varphi(x)$ has the form $H\,T_1 \cdots T_k$ where H can only be an abstraction, a type abstraction, or a variable, and $T_1, \ldots, T_k$ are terms or types.

The following terminology will be useful. A term whose type is a sort and which has the form $H\,T_1 \cdots T_k$ where H can only be an abstraction, a type abstraction, or a variable, and $T_1, \ldots, T_k$ are terms or types, is called a *nontrunk term.* A term $f\,T_1 \ldots T_k$ whose type is a sort and where f is a constant taking k arguments, is called a *trunk term.*

Clearly the type of any term that has an algebraic trunk decomposition must be a sort, but in fact that's all it takes:

Lemma 3.2
Any term M whose type is a sort has an algebraic trunk decomposition $M \equiv A[\varphi]$. Moreover, this decomposition is unique up to renaming the free variables of A, and when M is a trunk term, A is not a variable.

Equipped with this, we attempt to finish the proof of the claim. For an algebraic trunk decomposition $M \equiv A[\varphi]$, an algebraic redex must occur either entirely within one of the subterms $\varphi(x)$, or "essentially" within the trunk part. It will be useful to distinguish between such reduction steps.

Definition 3.3
We say that $A[\varphi] \xrightarrow{R} A'[\varphi']$ is an *algebraic trunk reduction step* if the R-redex is *not* a subterm of one of the $\varphi(x)$'s. We shall denote algebraic *trunk reductions* by $\xrightarrow{tR}$, and algebraic reductions in the non-trunk part by $\xrightarrow{ntR}$ (*non-trunk reductions*). A rewrite step $A[\varphi] \xrightarrow{tR} \varphi(x)$ for some $x \in FV(A)$ is called an *erasing step*, and is denoted as $A[\varphi] \xrightarrow{eR} \varphi(x)$.

Separating the trunk reductions is somewhat subtle because algebraic rewrite rules may be non-linear, or may erase some of their arguments. Part of the proof of lemma 2.2 (page 85) of [BT88] is invalidated by this problem. However, the argument can be repaired, as shown in the rest of this section. The following example shows exactly what the problem is: $A[\varphi] \xrightarrow{R} A'[\varphi']$ does not necessarily imply that $A \xrightarrow{R} A'$.

Example 3.4
Consider the signature $\{f, g, a, b, c\}$ with one sort s, where f is binary, g is ternary, and a, b, c

are nullary, and the term rewrite system $R = \{fxx \longrightarrow gxxx,\ a \longrightarrow b,\ b \longrightarrow c\}$. Let z be a higher-order variable of type $s \rightarrow s$. While we have the rewrite sequence

$$\begin{aligned} f(za)(zb) &\xrightarrow{ntR} f(zb)(zb) \\ &\xrightarrow{tR} g(zb)(zb)(zb) \\ &\xrightarrow{ntR} g(zb)(zc)(zb), \end{aligned}$$

we do not have that $fx_1x_2 \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} gy_1y_2y_3$ even if we rename the y's.

Example 3.4 also shows that nontrunk rewrite steps and trunk rewrite steps cannot always be permuted. The problem is caused by non-linear rewrite rules.

On the positive side, it is important to note that if a nontrunk term M R-reduces to another term N, then N cannot be a trunk term. This implies that for a non-trunk reduction $M \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} N$, if $M \equiv A[\varphi]$ is a trunk decomposition of M, then $N \equiv A[\varphi']$ for *the same trunk* A, i.e., the trunk does not grow in a non-trunk R-reduction.

We will proceed now with the formal development of the proof.

Lemma 3.5
If $M \equiv A[\varphi] \overset{R}{\longrightarrow} N$, *then the following holds:*

(1) if $M \overset{tR}{\longrightarrow} N$, *then we can write* $N \equiv A'[\varphi']$, *where for every* $y \in dom\varphi'$, *there is some* $x \in dom\varphi$ *such that* $\varphi'(y) \equiv \varphi(x)$, *and* A' *is some algebraic term;*

(2) if $M \overset{ntR}{\longrightarrow} N$, *then we can write* $N \equiv A[\varphi']$, *where* $\varphi(x_i) \overset{R}{\longrightarrow} \varphi'(x_i)$ *for some* $x_i \in dom\varphi$ *and* $\varphi'(x_j) \equiv \varphi(x_j)$ *for all* $j \neq i$.

Note that case (2) holds because a nontrunk term cannot rewrite to a trunk term. Thus, the trunk cannot grow.

Definition 3.6
Given two substitutions φ_1 and φ_2, we write $\varphi_1 \sqsubseteq \varphi_2$ iff for every $y \in dom\varphi_2$ there is some $x \in dom\varphi_1$ such that $\varphi_1(x) \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi_2(y)$.

Lemma 3.7
If $M \equiv A[\varphi] \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} M' \equiv A'[\varphi']$, *then* $\varphi \sqsubseteq \varphi'$.

Proof. An easy induction on the number of rewrite steps using lemma 3.5. □

Another key observation leading to the proof of the main theorem of this section is the following: $M \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} N$ iff

$$M \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} \circ \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow} M_1 \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} \circ \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow} \ldots \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} \circ \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow} M_{n-1} \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} \circ \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow} N,$$

for some $M_1, \ldots, M_{n-1}$, where $\circ$ is relation composition. Stated more concisely, $\xrightarrow{R}$ is the reflexive transitive closure of $\xrightarrow{ntR} \circ \xrightarrow{tR}$, notation $\xrightarrow{R} = (\xrightarrow{ntR} \circ \xrightarrow{tR})^*$.

Then, observe that if we can show the confluence of each square ("tile") in the diagram below, then by an induction on the number of such tiles, it is possible to prove our result.

$$\begin{array}{ccccc}
M & \xrightarrow{ntR} & M' & \xrightarrow{tR} & P \\
\downarrow ntR & & \downarrow ntR & & \downarrow ntR \\
M'' & \xrightarrow{ntR} & Q' & \xrightarrow{tR} & P' \\
\downarrow tR & & \downarrow tR & & \downarrow tR \\
N & \xrightarrow{ntR} & N' & \xrightarrow{tR} & Q
\end{array}$$

However, there are some technical difficulties. In particular, the bottom leftmost and top rightmost squares only commute if certain conditions are met. In order to state these conditions, it is convenient to define the relation $\propto$ (this relation was introduced by Toyama [Toy87]). The relation $\propto$ is needed to deal with rewrite rules that are not left-linear.

Definition 3.8
Given two term substitutions with the same domain, φ_1 and φ_2, we write $\varphi_1 \propto \varphi_2$ iff $\varphi_2(x) \equiv \varphi_2(y)$ for any x, y such that $\varphi_1(x) \equiv \varphi_1(y)$. Given two trunk terms $M_1 \equiv A_1[\varphi_1]$ and $M_2 \equiv A_2[\varphi_2]$, we write $M_1 \propto M_2$ iff $A_1 \equiv A_2$ and $\varphi_1 \propto \varphi_2$.

Given any trunk term $M \equiv A[\varphi_1]$, if $M \xrightarrow{tR} M'$, we know by lemma 3.5 that $M' \equiv A'[\varphi_1']$ and that for every $y \in dom\varphi_1'$, there is some $x \in dom\varphi_1$ such that $\varphi_1'(y) \equiv \varphi_1(x)$. Thus, we can define a function $h: dom\varphi_1' \to dom\varphi_1$ such that $\varphi_1'(y) \equiv \varphi_1(h(y))$ for every $y \in dom\varphi_1'$. The following lemmas show the significance of the relation $\propto$.

Lemma 3.9
Let M and N be trunk terms such that $M \propto N$. If $M \xrightarrow{tR} M'$, then there is some N' such that $N \xrightarrow{tR} N'$ and $M' \propto N'$. Furthermore, if $M \equiv A[\varphi_1]$, $M' \equiv A'[\varphi_1']$, and $N \equiv A[\varphi_2]$ (with $dom\varphi_1 = dom\varphi_2$), letting $h: dom\varphi_1' \to dom\varphi_1$ be any function such that $\varphi_1'(y) \equiv \varphi_1(h(y))$ for every $y \in dom\varphi_1'$, we have $N' \equiv A'[\varphi_2']$ where $dom\varphi_2' = dom\varphi_1'$ and $\varphi_2'(y) \equiv \varphi_2(h(y))$ for every $y \in dom\varphi_2'$.

Proof. Since $M \propto N$, we have $\varphi_1 \propto \varphi_2$. The left-hand side of the rule used in $M \xrightarrow{tR} M'$ occurs completely within A, and since $\varphi_1 \propto \varphi_2$, this same rule also applies to N. It is easily seen by lemma 3.5 that defining φ_2' such that $dom\varphi_2' = dom\varphi_1'$ and $\varphi_2'(y) \equiv \varphi_2(h(y))$ for every $y \in dom\varphi_2'$, letting $N' \equiv A'[\varphi_2']$, we have $M' \xrightarrow{tR} N'$ and $M' \propto N'$. □

Definition 3.10
We introduce a notation that will be convenient to use in what follows: if $FV(A) = \{x_1, \ldots, x_m\}$ then we will sometimes write $A[B_1, \ldots, B_m]$ instead of $A[B_1/x_1, \ldots, B_m/x_m]$.

We now prove lemmas that show that each kind of tile involved in the diagram showed earlier commutes, provided that appropriate conditions hold.

Lemma 3.11
Let M be a trunk term. If $M \xrightarrow{tR}\!\!\!\!\!\rightarrow P$, $M \xrightarrow{ntR}\!\!\!\!\!\rightarrow N$, and $M \propto N$, then there is some Q such that $P \xrightarrow{ntR}\!\!\!\!\!\rightarrow Q$, $P \propto Q$, and $N \xrightarrow{tR}\!\!\!\!\!\rightarrow Q$.

Proof. First, note that if $M \xrightarrow{tR}\!\!\!\!\!\rightarrow P$ contains some erasing step, because all the steps are trunk rewrites, it must be the last step. We first prove that if $M \xrightarrow{tR}\!\!\!\!\!\rightarrow P$, $M \xrightarrow{ntR}\!\!\!\!\!\rightarrow N$, and $M \propto N$, then there is some Q such that $P \xrightarrow{ntR}\!\!\!\!\!\rightarrow Q$, $P \propto Q$, and $N \xrightarrow{tR}\!\!\!\!\!\rightarrow Q$.

$$\begin{array}{ccc} M & \xrightarrow{tR} & P \\ \downarrow{\scriptstyle ntR} & & \downarrow{\scriptstyle ntR} \\ N & \xrightarrow{tR} & Q \end{array}$$

If $M \xrightarrow{tR} P$ is not an erasing step, the claim follows from lemma 3.9. If $M \xrightarrow{tR} P$ is an erasing step, then $M \equiv A[B_1, \ldots, B_m] \xrightarrow{eR} B_i \equiv P$. Since $M \propto N$, we have $N \equiv A[B'_1, \ldots, B'_m]$ where $\langle B_1, \ldots, B_m \rangle \propto \langle B'_1, \ldots, B'_m \rangle$ and $\langle B_1, \ldots, B_m \rangle \xrightarrow{ntR}\!\!\!\!\!\rightarrow \langle B'_1, \ldots, B'_m \rangle$. Because $M \propto N$, the rule $l \rightarrow x_i$ applied to M also applies to N, and the claim holds:

$$\begin{array}{ccc} M & \xrightarrow{eR} & B_i \\ \downarrow{\scriptstyle ntR} & & \downarrow{\scriptstyle ntR} \\ N & \xrightarrow{eR} & B'_i \end{array}$$

We conclude by induction on the length of the reduction sequence $M \xrightarrow{tR}\!\!\!\!\!\rightarrow P$, as indicated by the diagram below:

$$\begin{array}{ccccc} M & \xrightarrow{tR}\!\!\!\!\!\rightarrow & M' & \xrightarrow{tR} & P \\ \downarrow{\scriptstyle ntR} & & \downarrow{\scriptstyle ntR} & & \downarrow{\scriptstyle ntR} \\ N & \xrightarrow{tR}\!\!\!\!\!\rightarrow & Q' & \xrightarrow{tR} & Q \end{array}$$

The details are straightforward. □

We add a few more convenient notations.

Definition 3.12
Given φ and φ' with $dom\varphi = dom\varphi'$, the notation $\varphi \xrightarrow{R}\!\!\!\!\!\rightarrow \varphi'$ means that $\varphi(x) \xrightarrow{R}\!\!\!\!\!\rightarrow \varphi'(x)$ for every $x \in dom\varphi$.

Definition 3.13
For any term M, we write $CR(M)$ iff confluence holds from M, that is, whenever $M \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} M_1$ and $M \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} M_2$, there is some N such that $M_1 \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} N$ and $M_2 \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} N$. For any two terms M, N, we write $M \downarrow N$ iff there is some Q such that $M \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} Q$ and $N \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} Q$.

Let $S = \{M_1, \ldots, M_n\}$ be a finite set of terms, and assume that $CR(M_i)$ holds for every $M_i \in S$. If $M_i \downarrow M_j$ and $M_j \downarrow M_k$, then using the confluence from M_j, we also have $M_i \downarrow M_k$. Thus, $\downarrow$ is an equivalence relation on S. Then, for every equivalence class C of $\downarrow$, using the confluence from each M in C, it is easily seen that there is some term $M_C \in C$ such that $M \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} M_C$ for every $M \in C$. Consequently, we have the following lemma.

Lemma 3.14
Let $\varphi = [M_1/x_1, \ldots, M_n/x_n]$ and assume that $CR(M_i)$ holds for every M_i. Then there is some $\varphi' = [M_1'/x_1, \ldots, M_n'/x_n]$ such that $\varphi \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi'$, and $M_i \downarrow M_j$ implies that $M_i' \equiv M_j'$.

Using lemma 3.14, as in Toyama [Toy87], we have the following.

Lemma 3.15
Let $\varphi = [M_1/x_1, \ldots, M_n/x_n]$ and assume that $CR(M_i)$ for every M_i. If $\varphi \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi_1$ and $\varphi \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi_2$, then there is some φ' such that $\varphi_1 \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi'$, $\varphi_2 \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi'$, and $\varphi_1 \propto \varphi'$, $\varphi_2 \propto \varphi'$.

Using lemma 3.15, we can show the following result analogous to a result of Toyama [Toy87].

Lemma 3.16
Let $M \equiv A[\varphi]$ be a term such that $CR(\varphi(x_i))$ holds for every $x_i \in dom\varphi$. If $M \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} N$ and $M \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} P$, then there is some Q such that $N \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} Q$, $N \propto Q$, $P \overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} Q$, and $P \propto Q$.

Proof. If M is a trunk term, we have $N \equiv A[\varphi_1]$ and $P \equiv A[\varphi_2]$ where $\varphi \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi_1$ and $\varphi \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi_2$. Using lemma 3.15, we obtain some φ' such that $\varphi_1 \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi'$, $\varphi_2 \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \varphi'$, and $\varphi_1 \propto \varphi'$, $\varphi_2 \propto \varphi'$. Thus we can take $Q \equiv A[\varphi']$. If M is a nontrunk term, the lemma holds trivially because $M \equiv \varphi(x_i)$ for some x_i and $CR(\varphi(x_i))$ holds by hypothesis. □

Lemma 3.17
Let M be a trunk term, and assume that R is confluent on algebraic terms. If $M \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow} N$ and $M \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow} P$, then there is some Q such that $N \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow} Q$ and $P \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow} Q$.

Proof. Since all the steps are trunk rewrites, every redex occurs within the trunk, and confluence follows from the confluence of R on algebraic terms. □

We can now prove confluence on terms $M \equiv A[\varphi]$, provided that confluence holds for its nontrunk subterms. We use the fact noted earlier that $\overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} = (\overset{ntR}{\longrightarrow\!\!\!\!\!\rightarrow} \circ \overset{tR}{\longrightarrow\!\!\!\!\!\rightarrow})^*$.

Lemma 3.18
Let $M \equiv A[\varphi]$. If $CR(\varphi(x_i))$ holds for every $x_i \in dom\varphi$ and R is confluent on algebraic terms then $CR(M)$ also holds.

Proof. We first prove that if $M \xrightarrow{ntR} \circ \xrightarrow{tR} N$ and $M \xrightarrow{ntR} \circ \xrightarrow{tR} P$, there is some Q such that $N \xrightarrow{ntR} \circ \xrightarrow{tR} Q$ and $P \xrightarrow{ntR} \circ \xrightarrow{tR} Q$. The result follows from lemma 3.16, lemma 3.11, and lemma 3.17, which allow us to obtain the following diagram where $M' \propto Q'$, $M'' \propto Q'$, $P \propto P'$, and $N \propto N'$:

$$\begin{array}{ccccc}
M & \xrightarrow{ntR} & M' & \xrightarrow{tR} & P \\
\downarrow ntR & & \downarrow ntR & & \downarrow ntR \\
M'' & \xrightarrow{ntR} & Q' & \xrightarrow{tR} & P' \\
\downarrow tR & & \downarrow tR & & \downarrow tR \\
N & \xrightarrow{ntR} & N' & \xrightarrow{tR} & Q
\end{array}$$

From lemma 3.7, if $M \equiv A[\varphi] \xrightarrow{R} M' \equiv A'[\varphi']$, then $\varphi \sqsubseteq \varphi'$. Since $CR(\varphi(x_i))$ holds for every $x_i \in dom\varphi$ and for every $y \in dom\varphi'$ there is some $x \in dom\varphi$ such that $\varphi(x) \xrightarrow{R} \varphi'(y)$, we conclude that $CR(\varphi'(y))$ holds for every $y \in dom\varphi'$. Thus, we can use induction on the number of blocks of $\xrightarrow{ntR} \circ \xrightarrow{tR}$ steps to obtain the following confluence diagram:

$$\begin{array}{ccccccc}
M & \xrightarrow{R} & P' & \xrightarrow{ntR} & P'' & \xrightarrow{tR} & P \\
\downarrow R & & \downarrow R & & & & \downarrow R \\
N' & \xrightarrow{R} & Q' & \xrightarrow{ntR} & Q_1 & \xrightarrow{tR} & P_1 \\
\downarrow ntR & & \downarrow ntR & & \downarrow ntR & & \downarrow ntR \\
N'' & & Q_2 & \xrightarrow{ntR} & Q_3 & \xrightarrow{tR} & Q_4 \\
\downarrow tR & & \downarrow tR & & \downarrow tR & & \downarrow tR \\
N & \xrightarrow{R} & P_2 & \xrightarrow{ntR} & Q_5 & \xrightarrow{tR} & Q
\end{array}$$

□

We can finally prove the main theorem of this section.

Theorem 3.19
If $\xrightarrow{R}$ is confluent on algebraic terms, then $\xrightarrow{R}$ is confluent on all terms.

Proof. We proceed by induction on the size of terms. The case of a variable of non-base type is trivial, and so is the case of a variable of base type or an algebraic constant of base type since R is

confluent on algebraic terms. In the case of a term of the form $\lambda x{:}\sigma.\, M$ or $\lambda t.\, M$, since algebraic rewrite rules only apply within M, we apply the induction hypothesis. In case of an application, the only case in which the induction hypothesis does not immediately apply is the case of a trunk term $M \equiv fM_1 \ldots M_n$. However, if we decompose M as $M \equiv A[\varphi]$, since A is not a variable, each $\varphi(x_i)$ has size strictly smaller that the size of M, and by the induction hypothesis, $CR(\varphi(x_i))$ holds for every $x_i \in dom\varphi$. We conclude by applying lemma 3.18. □

4 Conservation of the Church-Rosser property

The key to the conservation result is the following lemma which shows that algebraic reduction "commutes" with λ^--reduction to normal form.

Lemma 4.1
Let r be an algebraic rewrite rule and M, N two terms. If $M \xrightarrow{r} N$ then $\lambda^- nf(M) \xrightarrow{r} \lambda^- nf(N)$.

Proof. Let $r \equiv A \longrightarrow B$, let $\{x_1, \ldots, x_n\} \equiv FV(A)$, let $x_1{:}s_1, \ldots, x_n{:}s_n$ be the sorts that these variables have in A (and B), and let s be the sort of A (and B). Since $M \xrightarrow{r} N$, there exist $C[\,]$ and a substitution φ such that $M \equiv C[A[\varphi]]$ and $N \equiv C[B[\varphi]]$. Let $P_i \stackrel{\text{def}}{=} \varphi(x_i)$ $(i = 1, \ldots, n)$ so we can write

$$M \equiv C[A[P_1/x_1, \ldots, P_n/x_n]] \qquad N \equiv C[B[P_1/x_1, \ldots, P_n/x_n]]$$

Introducing the notation $\lambda\vec{x}{:}\vec{s}.\, D \stackrel{\text{def}}{=} \lambda x_1{:}s_1.\ \cdots.\ \lambda x_n{:}s_n.\, D$, let

$$M' \stackrel{\text{def}}{=} C[(\lambda\vec{x}{:}\vec{s}.\, A)P_1 \cdots P_n] \qquad N' \stackrel{\text{def}}{=} C[(\lambda\vec{x}{:}\vec{s}.\, B)P_1 \cdots P_n]$$

Clearly, $M' \xrightarrow{\beta} M$ and $N' \xrightarrow{\beta} N$. Let z be a fresh variable of type $s_1 \to \cdots \to s_n \to s$. Then

$$M' \equiv C[z\, P_1 \cdots P_n][\lambda\vec{x}{:}\vec{s}.\, A \,/\, z] \qquad N' \equiv C[z\, P_1 \cdots P_n][\lambda\vec{x}{:}\vec{s}.\, B \,/\, z]$$

Let $Q \stackrel{\text{def}}{=} \lambda^- nf(C[z\, P1 \cdots P_n])$. We claim that Q has the following property:

(∗) Any occurrence of z is at the head of a subterm of the form $z\, P'_1 \cdots P'_n$ where P'_i has type s_i $(i = 1, \ldots, n)$ and $z\, P'_1 \cdots P'_n$ has type s (and thus cannot be further applied to terms or types).

Indeed, property (∗) holds for $C[z\, P_1 \cdots P_n]$ and it is easy to check that it is preserved under β-reduction, $\mathcal{T}\beta$-reduction and $\mathcal{T}\eta$-reduction (but not under η-reduction; see example 4.4).

Let

$$M'' \stackrel{\text{def}}{=} \beta nf(Q[\lambda\vec{x}{:}\vec{s}.\, A \,/\, z]) \qquad N'' \stackrel{\text{def}}{=} \beta nf(Q[\lambda\vec{x}{:}\vec{s}.\, B \,/\, z])\ .$$

We will show that M'' is in λ^--normal form and since clearly M λ^--converts to M'', we must have $M'' \equiv \lambda^- nf(M)$. Similarly, $N'' \equiv \lambda^- nf(N)$. It remains then to prove that $M'' \xrightarrow{r} N''$. Both

the fact that M'' and N'' are in λ^--normal form and the fact that $M'' \xrightarrow{r} N''$ are consequences of the following claim.

Claim. If Z is a term in λ^--normal form having property $(*)$ then

$$X \stackrel{\text{def}}{=} \beta nf(Z[\lambda\vec{x}{:}\vec{s}.\, A \,/\, z]) \qquad\qquad Y \stackrel{\text{def}}{=} \beta nf(Z[\lambda\vec{x}{:}\vec{s}.\, B \,/\, z])$$

are in λ^--normal form and $X \xrightarrow{r} Y$.

The proof of the claim is by induction on the size of Z. Since Z is in λ^--normal form, $Z \equiv \lambda v_1.\cdots.\lambda v_k.h\,T_1\cdots T_m$ where the v_i's are either type variables or of the form $y{:}\tau$, h is a variable or a constant, the T_j's are either types or terms in λ^--normal form, and, we do *not* have $v_k \equiv T_m \equiv t$ for some type variable t (to avoid having a $\mathcal{T}\eta$-redex). As before, we introduce the simpler notation $\lambda\vec{v}.\,h\,T_1\cdots T_m \stackrel{\text{def}}{=} \lambda v_1.\cdots.\lambda v_k.\,h\,T_1\cdots T_m$. We distinguish two cases.

($h \not\equiv z$) Let D be A or B. Then, $\beta nf(Z[\lambda\vec{x}{:}\vec{s}.\, D \,/\, z]) \equiv \lambda\vec{v}.\,h\,T_1'\cdots T_m'$ where $T_j' \stackrel{\text{def}}{=} T_j$ if T_j is a type and $T_j' \stackrel{\text{def}}{=} \beta nf(T_j[\lambda\vec{x}{:}\vec{s}.\, D \,/\, z])$ if T_j is a term. In the latter case, T_j is a λ^--normal form of strictly smaller size than Z. Since property $(*)$ is inherited by subterms, we can apply the induction hypothesis and the statement of the claim for Z easily follows.

($h \equiv z$) In this case, by property $(*)$, $m = n$ and $Z \equiv \lambda\vec{v}.\, z\, Z_1\cdots Z_n$ where Z_i is a term of type s_i $(i = 1,\ldots,n)$. Each of the Z_i's is a λ^--normal form having property $(*)$ and of strictly smaller size than Z so the induction hypothesis applies. Let

$$X_i \stackrel{\text{def}}{=} \beta nf(Z_i[\lambda\vec{x}{:}\vec{s}.\, A \,/\, z]) \qquad Y_i \stackrel{\text{def}}{=} \beta nf(Z_i[\lambda\vec{x}{:}\vec{s}.\, B \,/\, z]) \qquad (i = 1,\ldots,n)$$

Consider $X' \stackrel{\text{def}}{=} \lambda\vec{v}.\, A[X_1/x_1,\ldots,X_n/x_n]$. By the induction hypothesis, the X_i's are in λ^--normal form and since their type is a sort, they cannot create λ^--redexes by substitution. Thus X' is in λ^--normal form. Since $Z[\lambda\vec{x}{:}\vec{s}.\, A \,/\, z]$ β-reduces to X' and since X' is, in particular, also in β-normal form, we have $X \equiv X'$. Similarly, $Y \equiv \lambda\vec{v}.B[Y_1/x_1,\ldots,Y_n/x_n]$ and Y is in λ^--normal form. Moreover, by induction hypothesis $X_i \xrightarrow{r} Y_i$ $(i = 1,\ldots,n)$, hence $X \xrightarrow{r} Y$.

This ends the proof of the claim and that of the lemma. □

Remark. At first glance, the previous proof may seem unnecessary complex . Note, however, that, in general, the simple minded

$$\lambda^- nf(C[A[P_1/x_1,\ldots,P_n/x_n]]) \equiv \lambda^- nf(C)[A[\lambda^- nf(P_1)/x_1,\ldots,\lambda^- nf(P_n)/x_n]]$$

fails. Our solution protects the r-redex through β-expansion in order to trace its behavior during λ^--normalization. Note also that the normalization process can make copies of the r-redex, modify the arguments P_i, and even substitute copies of the modified redex inside the arguments of another copy of the redex. This "nesting" is resolved by noting the invariance of the property $(*)$ and by the slightly more general statement that we prove in the claim.

Lemma 4.2 *(Reduction mapping)*

Let R be a set of algebraic rewrite rules, and M, N two terms. If $M \overset{\lambda^- R}{\longrightarrow\!\!\!\!\!\rightarrow} N$ then $\lambda^- nf(M) \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda^- nf(N)$.

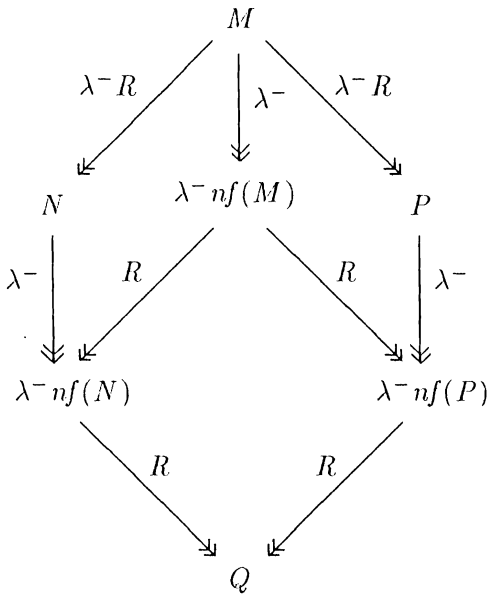
Proof. By induction on the length of the reduction chain from M to N. Immediate from lemma 4.1. □

Finally, the main result of the paper:

Theorem 4.3

If R-reduction is confluent on algebraic terms then $\lambda^- R$-reduction is confluent on all terms.

Proof. Suppose that $N \overset{\lambda^- R}{\longleftarrow\!\!\!\!\!\leftarrow} M \overset{\lambda^- R}{\longrightarrow\!\!\!\!\!\rightarrow} P$. By mapping everything to λ^--normal form, we obtain from lemma 4.2 that $\lambda^- nf(N) \overset{R}{\longleftarrow\!\!\!\!\!\leftarrow} \lambda^- nf(M) \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda^- nf(P)$. Then, by theorem 3.19, there exists a Q such that $\lambda^- nf(N) \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} Q \overset{R}{\longleftarrow\!\!\!\!\!\leftarrow} \lambda^- nf(P)$. Thus $N \overset{\lambda^-}{\longrightarrow\!\!\!\!\!\rightarrow} \lambda^- nf(N) \overset{R}{\longrightarrow\!\!\!\!\!\rightarrow} Q \overset{R}{\longleftarrow\!\!\!\!\!\leftarrow} \lambda^- nf(P) \overset{\lambda^-}{\longleftarrow\!\!\!\!\!\leftarrow} P$. The proof is summarized by the following diagram.



□

The theorem fails if we replace λ^- with $\lambda^\forall$, as we can see from the following simple example.

Example 4.4

Let s be a sort, and $f : s \to s$. $a : s$ be constants. Consider the algebraic rule $r \equiv fx \longrightarrow a$ where x has type s and the term $M \equiv \lambda y{:}s.\, fy$. Then, $M \overset{r}{\longrightarrow} \lambda y{:}s.\, a$ and $M \overset{\eta}{\longrightarrow} f$. Since f and $\lambda y{:}s.\, a$ are $\lambda^\forall r$-normal forms, confluence fails.

It is instructive to see how the proof of lemma 4.1 breaks down if we try to extend it to $\lambda^{\forall}$-reduction. Take r and M as before and $N \equiv \lambda y{:}s.\, a$. Then $\lambda y{:}s.\, zy$ has property $(*)$, but after one η-reduction we obtain just z for which property $(*)$ fails.

Remark. In short, the proof of theorem 4.3 consists of the observation that the reduction mapping lemma (lemma 4.2) and the confluence of algebraic reduction on algebraic terms imply the confluence of mixed reduction on mixed terms. Thérèse Hardin uses similar reduction mapping lemmas to prove confluence results in the strong categorical combinatory logic (see the *interpretation method* in [Har89])[5]. Moreover, Hardin makes the nice observation that reduction mapping lemmas also work "in reverse". In our case, using also lemma 2.12, this comes down to the fact that the reduction mapping lemma (lemma 4.2) and the confluence of mixed reduction on mixed terms imply the confluence of algebraic reduction on algebraic terms. However, there is no need in our case for the reduction mapping lemma in order to show that the confluence of mixed reduction on mixed terms implies the confluence of algebraic reduction on algebraic terms. Indeed, let $B_1 \overset{R}{\twoheadleftarrow} A \overset{R}{\twoheadrightarrow} B_2$ be algebraic reductions on algebraic terms. By confluence of mixed reduction there exists M such that $B_1 \overset{\lambda^- R}{\twoheadrightarrow} M \overset{\lambda^- R}{\twoheadleftarrow} B_2$. But the B_i's cannot contain any λ^--redex and using lemma 2.12 we conclude that all the terms and reduction steps in the reduction chains $B_1 \overset{\lambda^- R}{\twoheadrightarrow} M \overset{\lambda^- R}{\twoheadleftarrow} B_2$ are actually algebraic.

5 Deciding equational reasoning (even with η)

If we set aside the operational semantics issues, the interest in rewrite systems stems from their use in automated equational reasoning. How are the results that we have established applicable to deciding equational provability? The answer depends on what kind of equational reasoning we have in mind as differences arise depending on whether we insist or not on models with *empty* sorts or types. Some care is needed to formulate equational reasoning that is sound in models with empty types [GM82, LS86, MMMS87]. In particular, one tags equations with finite sets of variables (which include, but may not be limited to, the variables that are actually free in the equation) and one defines truth by universally quantifying over *all* the variables in the tag set. Since we need to know over which parts of the semantic universe to quantify, we assign types to the variables in the tag sets.

Definition 5.1 *(Equations)*
A *declaration* (sometimes called a *type assignment*) is a partial function $\Delta : \mathcal{X} \longrightarrow \mathcal{T}$ with *finite* domain. At the same time, we will also regard declarations as finite sets of pairs $x{:}\sigma$ such that no x occurs twice. This allows us to write $\Delta \subseteq \Delta'$ instead of " $dom\Delta \subseteq dom\Delta'$ and $\Delta'(x) \equiv \Delta(x)$ for every $x \in dom\Delta$". We agree to write $\Delta, x{:}\sigma$ for $\Delta \cup \{x{:}\sigma\}$ and, by convention, the use of this notation implies that $x \notin dom\Delta$.

A *term M is compatible with a declaration* Δ if $FV(M) \subseteq dom\Delta$ and each $x \in FV(M)$ has type $\Delta(x)$ in M.

An *equation* is a triple $M \stackrel{\Delta}{=} N$ such that both M and N are compatible with Δ.

[5] We note that the observations were made independently, cf. [BT88].

We will consider equational proofs in the form of chains of one-step conversions. Just using the convertibility relation won't do, because we want to distinguish reasoning that is sound in models with empty types. This will be done using the declaration part of the equations.

Definition 5.2 *(Compatible convertibility)*
Let $\xrightarrow{\rho}$ be a reduction relation. *M and N are ρ-convertible under* Δ whenever there exist $P_0, \ldots, P_k$ $(k \geq 0)$ such that each P_i is compatible with Δ and such that

$$M \equiv P_0 \xleftrightarrow{\rho} \cdots \xleftrightarrow{\rho} P_k \equiv N \ .$$

Note that if M and N are ρ-convertible under Δ then, in particular, they are also ρ-convertible in the usual sense, and, moreover, M and N are compatible with Δ. We are now ready to define two kinds of equational provability, one that is sound in models which may have empty types and one that is sound in models which have all types non-empty.

Definition 5.3 *(Equational provability)*
Let $\xrightarrow{\rho}$ be a reduction relation.

- The equation $M \stackrel{\Delta}{=} N$ is (MAYBE EMPTY)-provable from ρ whenever M and N are ρ-convertible under Δ.
- The equation $M \stackrel{\Delta}{=} N$ is (NOT EMPTY)-provable from ρ whenever there exists $\Delta' \supseteq \Delta$ such that M and N are ρ-convertible under Δ'.

Remark. These notions of provability can be shown to be equivalent to others given by proof rules and axioms, as in [BTC88]. In that case, the correspondent of the algebraic rewrite rule $A \longrightarrow B$ would be the axiom $A \stackrel{\Delta}{=} B$ where $dom\Delta \stackrel{\text{def}}{=} FV(A)$ and $\Delta(x)$ is the type that x has in A. The difference between the correspondents of (MAYBE EMPTY) and (NOT EMPTY) would be that the latter would have the additional "discharge" rule

$$\frac{M \stackrel{\Delta, x:\sigma}{=} N}{M \stackrel{\Delta}{=} N} \qquad \text{where } x \notin FV(M) \cup FV(N) \ .$$

(See [GM82, LS86, MMMS87, BTC88] for more on these and related proof systems and their (in)completeness properties.)

As a corollary of the main result of this paper (theorem 4.3) and the main result of a companion paper (theorem 5.7 of [BTG91]), we obtain that if R is canonical (confluent and strongly normalizing) on algebraic terms then both (MAYBE EMPTY)- and (NOT EMPTY)-provability from $\lambda^- R \equiv R + \beta +$ type-β + type-η are decidable. This follows from the following simple fact.

Proposition 5.4
Let R be confluent on algebraic terms. Then, M and N are $\lambda^- R$-convertible under Δ iff M and N are compatible with Δ and there exists P such that $M \xrightarrow{\lambda^- R} P \xleftarrow{\lambda^- R} N$

Proof. By theorem 4.3 $\stackrel{\lambda^- R}{\longrightarrow}$ is confluent, therefore in one direction we can show by induction on the length of chain of conversions that we can obtain two chains of reductions to the same P (well-known argument). In the other direction, we need only observe that if M is compatible with Δ and $M \stackrel{\lambda^- R}{\longrightarrow\!\!\!\!\!\rightarrow} X$ then X is also compatible with Δ. □

This proposition, together with theorem 5.7 of [BTG91], yields immediately the following.

Corollary 5.5
If R is confluent on algebraic terms then $M \stackrel{\Delta}{=} N$ is (MAYBE EMPTY)*-provable from $\lambda^- R$ iff it is* (NOT EMPTY)*-provable from the same. Moreover, if R is also strongly normalizing on algebraic terms, the provabilities are further equivalent to $\lambda^- nf(M) \equiv \lambda^- nf(N)$.*

Therefore, when R is canonical on algebraic terms, the *decision procedure* for the provability (both kinds) of an equation from $\lambda^- R$ is to take both sides of the equation to $\lambda^- R$-normal form and to test if the results coincide.

Now, what happens if we insist that η be available too? In view of the counterexample presented in section 4 (example 4.4), there are algebraic rewrite systems R which are canonical but such that $\lambda^\forall R$-reduction is not confluent, so we cannot repeat the previous arguments. Nonetheless, we will show that we can still decide provability from $\lambda^\forall R$. This will require some formal development.

The decision procedure will use conversion to *long normal forms*, a straightforward generalization of the *η-expanded normal forms* in [Hue75] called *long $\beta\eta$-normal forms* in [Sta82].

Definition 5.6 *(Long normal form)*
A term M is in long normal form if $M \equiv \lambda v_1.\ \cdots.\ \lambda v_k.\, h\, T_1 \cdots T_m$ where the v_i's are either type variables or of the form $y{:}\tau$, h is a variable or a constant, the T_j's are either type expressions or (inductively) terms in long normal form, we do *not* have $v_k \equiv T_m \equiv t$ for some type variable t (to avoid having a $\mathcal{T}\eta$-redex), and the type of $h\, T_1 \cdots T_m$ is either a sort, or a type variable, or of the form $\forall t.\, \sigma$. (We will often use the shorter notation $\lambda \vec{v}.\, h\, T_1 \cdots T_m \stackrel{\text{def}}{=} \lambda v_1.\ \cdots.\ \lambda v_k.\, h\, T_1 \cdots T_m$).

While long normal forms are in general *not* in η-normal form, the name is justified by the following result.

Lemma 5.7
Any term is $\lambda^\forall$-convertible to a unique long normal form.

Proof. Since every long normal form is also a λ^--normal form, it is sufficient to show how to η-convert any λ^--normal form to a unique long normal form. If M is in λ^--normal form then already $M \equiv \lambda v_1.\ \cdots.\ \lambda v_k.\, h\, T_1 \cdots T_m$ where the v_i's are either type variables or of the form $y{:}\tau$, h is a variable or a constant, the T_j's are either type expressions or terms in λ^--normal form, and, we do *not* have $v_k \equiv T_m \equiv t$ for some type variable t. Suppose that in $\lambda \vec{v}.\, h\, T_1 \cdots T_m$ we have already (recursively and, for the uniqueness, inductively) η-converted those T_j which are terms (and therefore λ^--normal forms of strictly smaller size) to their unique long normal form.

Let the type of $h\,T_1 \cdots T_m$ be $\sigma_1 \to \cdots \to \sigma_n \to \tau$ where $n \geq 0$ and τ is either a sort, or a type variable, or of the form $\forall t.\,\sigma$ (any type is of this form). From this, the unique long normal form is reached by performing the η-expansions that give $\lambda\vec{v}.\,\lambda x_1{:}\sigma_1.\,\cdots.\,\lambda x_n{:}\sigma_n.\,h\,T_1 \cdots T_m\,U_1 \cdots U_n$ where U_i is the long normal form of x_i. □

We denote by $lnf(M)$ the long normal form of M. It turns out that while in general we do not have a reduction mapping result for mapping to η-normal form, we will have such a result for mapping to long normal form.

Lemma 5.8
Let $r \in R$, and let M, N be two terms. If $M \xrightarrow{r} N$ then $lnf(M) \xrightarrow{r}\!\!\!\!\!\rightarrow lnf(N)$.

Proof. The proof is almost the same as that of lemma 4.1. The only notable addition is that one must check that property $(*)$ is preserved under the kind of η-expansion used to reach long normal form (see the proof of lemma 5.7). To see this, let Q' be a term of the form $\lambda\vec{v}.\,h\,T_1 \cdots T_m$ and such that the type of $h\,T_1 \cdots T_m$ is $\tau - \tau'$, and assume that Q' has property $(*)$. Since the type of $h\,T_1 \cdots T_m$ is of the form $\tau \to \tau'$, we can't have $h \equiv z$, hence z can only occur within the T_j's. Clearly then, $\lambda\vec{v}.\,\lambda y{:}\tau.\,h\,T_1 \cdots T_m\,y$ also has the property $(*)$. □

Lemma 5.9
If M $\lambda^{\forall}R$-converts to N then $lnf(M)$ R-converts to $lnf(N)$.

Proof. By induction on the length of the conversion chain from M to N. Immediate from lemma 5.8. □

When R is canonical on algebraic terms, it is also canonical on all terms, by theorem 3.19 of this paper and theorem 3.10 of the companion paper [BTG91]. In that case, we denote with $Rnf(M)$ the R-normal form of a term M.

Proposition 5.10
Let R be canonical on algebraic terms. Then, M and N are $\lambda^{\forall}R$-convertible under Δ iff M and N are compatible with Δ and $Rnf(lnf(M)) \equiv Rnf(lnf(N))$.

Proof. Suppose that M and N are $\lambda^{\forall}R$-convertible under Δ. Then, they are also $\lambda^{\forall}R$-convertible in the usual sense, hence by lemma 5.9 $lnf(M)$ and $lnf(N)$ are R-convertible, hence their R-normal forms coincide. For the converse, we need only observe that if M is compatible with Δ then for any X appearing in the conversion chain from M to $lnf(M)$ (see the proof of lemma 5.7) X is also compatible with Δ. Indeed, η-expansions (as opposed to other kinds of expansion) do not introduce new variables. □

Corollary 5.11
If R is canonical on algebraic terms then $M \triangleq N$ is (MAYBE EMPTY)*-provable from $\lambda^{\forall}R$ iff it is* (NOT EMPTY)*-provable from the same iff $Rnf(lnf(M)) \equiv Rnf(lnf(N))$.*

Therefore, when R is canonical on algebraic terms, the *decision procedure* for the provability (both kinds) of an equation from $\lambda^{\forall}R$ is to take both sides of the equation to long normal form, then to take these to R-normal form, and finally to test if the results coincide.

6 Directions for Further Research

Of course, one would also like to know what to do in the absence of an equivalent canonical rewrite system. We conjecture that the proof-theoretic reduction from simply typed theories with algebraic axioms to algebraic theories, given in [BT88], can be generalized to polymorphic theories.

Our results show that some important properties of algebraic systems are preserved when algebraic rewriting and polymorphic lambda-term rewriting are mixed. As applications to the results of this paper, we intend to investigate higher-order unification modulo an algebraic theory. For the simply-typed lambda calculus, we conjectured earlier that adding the lazy paramodulation rule investigated in [GS89a] to the set of higher-order transformations investigated in [GS89b] yields a complete set of transformations for higher-order E-unification. This has been confirmed by Snyder, using the reduction mapping result in lemma 5.8 [Sny90]. We also intend to investigate the possibility of extending Knuth-Bendix completion procedures to polymorphic theories with algebraic axioms.

Another direction of investigation is to consider more complicated type disciplines, such as that of the Calculus of Constructions [CH88].

More generally, we feel that the results of this paper are only a first step towards extending the important field of term rewriting systems to include higher-order rewriting. One of our main goals is to provide rigorous methods for understanding higher-order functional and logic programming. In particular, one is interested in rules which describe the behaviour of higher-order operations (such as *maplist*, for example). In any case, a lot of care will be needed with higher-order rules because, for example, fixed points are also described this way: $YF = F(YF)$. Rules in which higher-order variables are applied to one or more arguments in the left hand side term also cause problems. Consider a signature with one sort s, a unary operation f and a nullary operation a, and the higher-order rewrite rule $f(za) \xrightarrow{r} a$ where z is a variable of type $s \to s$. Then

$$fa \xleftarrow{\beta} f((\lambda x{:}\, s.\, x)a) \xrightarrow{r} a$$

Since fa and a are distinct βr-normal forms, confluence fails.

References

[AS85] H. Abelson and G. J. Sussman. *Structure and interpretation of computer programs.* The MIT Press, 1985.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, second edition, 1984.

[Ber84] I. Bercovici. Strong normalization for typed lambda calculus with surjective pairing—Tait's method. Unpublished manuscript, Laboratory for Computer Science, MIT, July 1984.

[BMM90] K. Bruce, A. Meyer, and J. Mitchell. The semantics of second-order lambda calculus. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 213–272 (Ch. 10). Addison-Wesley, 1990. Also to appear in Information and Computation.

[BT88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of the Symposium on Logic in Computer Science*, pages 82–90. IEEE, July 1988.

[BTC88] V. Breazu-Tannen and T. Coquand. Extensional models for polymorphism. *Theoretical Computer Science*, 59:85–114, 1988.

[BTG91] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83:3–28, 1991.

[BTM87] V. Breazu-Tannen and A. R. Meyer. Computable values can be classical. In *Proceedings of the 14th Symposium on Principles of Programming Languages*, pages 238–245. ACM, January 1987.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[Dou86] D. Dougherty. Personal communication, September 1986.

[Dou91] D. Dougherty. Adding algebraic rewriting to the untyped lambda calculus. *Information and Computation*, ??:??–??, 1991. To appear.

[dV82] R. C. de Vrier. Strong normalization in $N - HA^{\omega}$. Manuscript, 1982.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1: equations and initial semantics.* Springer-Verlag, 1985.

[FLO83] S. Fortune, D. Leivant, and M. O'Donnell. The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30(1):151–185, January 1983.

[Fri75] H. Friedman. Equality between functionals. In R. Parikh, editor, *Proceedings of the Logic Colloqium '73*, pages 22–37. *Lecture Notes in Mathematics*, Vol. 453, Springer-Verlag, 1975.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieure.* PhD thesis, Université Paris VII, 1972.

[GM82] J. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *SIGPLAN Notes*, 17:9–17, 1982.

[GMW79] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1979.

[GS89a] J. Gallier and W. Snyder. Complete sets of transformations for general E-Unification. *Theoretical Computer Science*, 67:203–260, 1989.

[GS89b] J. Gallier and W. Snyder. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[Har89] T. Hardin. Confluence results for the pure strong categorical logic CCL. λ-calculi as subsytems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[HM90] B. T. Howard and J. C. Mitchell. Operational and axiomatic semantics of PCF. In *Proceedings of the LISP and Functional Programming Conference, Nice*, pages 298–306, New York, June 1990. ACM.

[Hud86] P. Hudak. Para-functional programming. *Computer*, 18:60–70, August 1986.

[Hue75] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Klo80] J. W. Klop. Combinatory reduction systems. Tract 129, Mathematical Center, Amsterdam, 1980.

[Klo87] J. W. Klop. Term rewriting systems: a tutorial. *Bull. EATCS*, 32:143–182, June 1987.

[LS86] J. Lambek and P. J. Scott. *Introduction to higher-order categorical logic*, volume 7 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1986.

[MG85] J. Meseguer and J. Goguen. Deduction with many-sorted rewrite. Technical Report 42, CSLI, Stanford, 1985.

[MMMS87] A. R. Meyer, J. C. Mitchell, E. Moggi, and R. Statman. Empty types in polymorphic λ-calculus. In *Proceedings of the 14th Symposium on Principles of Programming Languages*, pages 253–262. ACM, January 1987. Reprinted with corrections in "Logical foundations of functional programming", G. Huet ed., Addison-Wesley 1990.

[MR86] A. R. Meyer and M. B. Reinhold. 'Type' is not a type: Preliminary report. In *Conf. Record Thirteenth Ann. Symp. Principles of Programming Languages*, pages 287–295. ACM, January 1986.

[New42] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Ann. Math*, 43:223–243, 1942.

[Pot81] G. Pottinger. The Church-Rosser theorem for the typed λ-calculus with surjective pairing. *Notre Dame J. of Formal Logic*, 22:264–268, 1981.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425. *Springer Lecture Notes in Computer Science*, Vol. 19, Springer-Verlag, 1974.

[Sny90] W. Snyder. Higher-order E-unification. In *Proceedings of the International Conference on Automated Deduction, Kaiserslautern*, pages 576–587, July 1990.

[Sta81] R. Statman. Number theoretic functions computable by polymorphic programs. In *22nd Symposium on Foundations of Computer Science*, pages 279–282. IEEE, 1981.

[Sta82] R. Statman. Completeness, invariance and λ-definability. *Journal of Symbolic Logic*, 47:17–26, 1982.

[Toy87] Y. Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1):128–143, January 1987.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. *Springer Lecture Notes in Computer Science*, Vol. 201, Springer-Verlag, 1985.