Higher-Order Unification via Combinators

Daniel J. Dougherty
Department of Mathematics
Wesleyan University
Middletown, CT 06457 USA
ddougherty@eagle.wesleyan.edu

Abstract

We present an algorithm for unification in the simply typed lambda calculus which enumerates complete sets of unifiers using a finitely branching search space. In fact, the types of terms may contain type-variables, so that a solution may involve type-substitution as well as term-substitution. the problem is first translated into the problem of unification with respect to extensional equality in combinatory logic, and the algorithm is defined in terms of transformations on systems of combinatory terms. These transformations are based on a new method (itself based on systems) for deciding extensional equality between typed combinatory logic terms.

1 Introduction

This paper develops a new algorithm for higher-order unification. A higher-order unification problem is specified by two terms F and G of the explicitly simply typed lambda calculus $\mathcal{LC}$; a solution is a substitution σ such that $\sigma F =_{\beta\eta} \sigma G$. We will always assume the extensionality axiom η in this paper.

In fact we treat the more general problem in which the types of terms contain type variables, which are eligible to be instantiated by our answer substitutions. This might be described as unification in the middle ground between the "Church view" and the "Curry view" of typing. This increased generality is actually necessary in order that our algorithm behave nicely, as will be explained below.

Typed combinatory logic $\mathcal{CL}$ is an alternative, algebraic, formalization of higher-order logic, and unification in algebraic theories has been the focus of much recent research, So it is natural to try to solve higher-order unification problems by passing to $\mathcal{CL}$. Under any of the standard effective translations between $\mathcal{LC}$ and $\mathcal{CL}$, one might translate the relevant $\mathcal{LC}$-terms into $\mathcal{CL}$-terms and attempt to unify these. The fact that the basic $\mathcal{CL}$ axioms admit a convergent (i.e., confluent and terminating) rewrite system makes this program particularly appealing, since such systems support narrowing as a unification procedure.

A difficulty arises immediately, however. The equality generated by the axioms for I, K, and S (called *weak equality*) is *not* the equality induced by $\beta\eta$-equality under the translation, and no convergent rewrite system is known for this induced equality (called *extensional combinatory equality*).

We solve this difficulty by defining a notion of reduction on *systems* of $\mathcal{CL}$ terms which decides extensional equality. This method may be of interest in its own right. The key fact

for this paper, however, is that this reduction supports the standard unification strategy of narrowing. Indeed, we present an algorithm which is essentially a normalized narrowing algorithm, described in terms of transformations on systems. The procedure enumerates a complete set of extensional-equality-unifiers for any system of $\mathcal{CL}$ terms, and so provides a solution to the higher-order unification problem. This represents an *algebraic* approach to the higher-order problem, without the complexities of bound variables. In particular, we are spared the usual expansion of $\mathcal{LC}$ terms to their η-long form.

The search space of our procedure is finitely branching, eliminating the most glaring obstacle to an implementation of complete higher-order unification based on Huet's classical method [Hue73], [Hue75].

Further theoretical investigation should be able to take advantage of research in term rewriting and first-order unification. Indeed, the work here provides a uniform setting for first-order and higher-order unification; Section 4 has a brief discussion of some work in progress along these lines.

There is considerable recent interest in compiling functional programming languages into combinators, motivated by the inefficiencies (seemingly) inherent in instantiating terms in the presence of bound variables; see [Pey87], especially Ch. 13, for a discussion. We expect that implementations of higher-order unification can enjoy similar benefit from the passage to combinators. Implementations in hardware of combinator-reduction are described in [Sto85] and [Sch86].

Pure higher-order unification has found application in automated theorem proving in higher-order logic, specification of higher-logics, program transformation, machine learning, type inference in polymorphic lambda calculus, and extensions of logic programming.

There have been attempts to extend classical higher-order unification to allow more flexible typing-schemes. Nipkow treats a λ-calculus with type-variables (and a notion of constraints on type-variables), and the procedure given there has been incorporated into the generic theorem prover Isabelle [Pau90]; Elliott ([Ell89]) presents an algorithm for unification in the presence of dependent function types, designed as the basis for a generalization of the programming language λ-Prolog [Nad87]. Each of these algorithms is based on Huet's method; neither of them is a complete unification procedure.

Higher-order unification is undecidable [Hue73], even when restricted to second-order terms [Gol81]. The first complete enumeration methods for higher-order unification are due to Pietrzykowski [Pie71] for second-order logic and Jensen and Pietrzykowski [JP73] for full higher-order logic. Huet's seminal work [Hue75] refined these methods, pointed out the importance of *preunification*, and gave a practical algorithm for the latter. A comparison of our work with Huet's method will be found at the end of Section 3. Complete sets of transformations for higher-order unification were developed by Gallier and Snyder [GS89b]. The use of narrowing as an algebraic unification procedure originates with Fay [Fay79]; normalized narrowing was proposed by Réty [Ret87]. There is a notion of *strong reduction* in the literature which captures extensional equality (see [CHS72] or [HS86]), but it lacks many of the nice properties of algebraic rewriting systems and does not seem suitable as a foundation for unification.

Although a naive algorithm for translating $\mathcal{LC}$ terms into $\mathcal{CL}$ terms can use quadratic time and space, Statman [Sta86] has given an $O(n \log n)$ time and space translation. We conclude that the translation process is not itself a source of intractability.

1.1 Preliminaries

We will often draw upon classical results about the lambda calculus and combinatory logic; [HS86] is a particularly good source for the relationship between $\mathcal{LC}$ and $\mathcal{CL}$.

In the course of testing equality or unifiability of terms we will find it convenient to introduce constants not occurring in any terms under consideration; this is the motivation for the set *Args* defined below. It will also be convenient to arrange that distinct term-variables do not become identical by virtue of a type-substitution; this requires a precise notion of type-erasure for term-variables.

Terms and equalities

The *types* are formed by closing a set of *base types* and *type-variables* under the operation: $(\alpha_1 \to \alpha_2)$.

Fix an infinite well-ordered set of *indeterminates* and an infinite well-ordered set of *parameters*. A *term-variable* is an ordered pair consisting of an indeterminate and a type; a *constant* is an ordered pair consisting of a parameter and a type; an *atom* is either a term-variable or a constant. The *type-erasure* of an atom is the first element of the pair.

We assume that the set of parameters has a distinguished infinite subset *Args*, and when discussing combinatory logic we assume that the non-*Args* include the symbols I, K, and S.

$\mathcal{LC}$ is the set of explicitly simply typed lambda terms over the atoms excluding I, K, and S; $\mathcal{CL}$ is the set of explicitly simply typed combinatory logic terms over these atoms together with the various I, K, and S typed as usual. The *support* of a term T, $Supp(T)$, is the set of type-variables occurring in T together with the indeterminates occurring among the type-erasures of the atoms in T; a *pure* term is a term in which no constant occurs whose erasure is in *Args*. A *fresh* indeterminate or parameter is one not occurring in any term in the current context; we will often refer to a choice of a term T with fresh variables, by which we mean that $Supp(T)$ is disjoint from all type-variables and indeterminates in the current context.

The typed I, K, and S are called *redex* atoms. A $\mathcal{CL}$-term is *functional* if it is of one of the forms: I, K, KA, S, SA, or SAB; it is *passive* if it is of the form $hM_1 \cdots M_k$ $(k \geq 0)$ where h (the *head* of the term) is a non-redex atom; a passive term is *flexible* if it has a variable at the head, otherwise it is *rigid*. These latter notions have already been defined for $\mathcal{LC}$ terms by Huet [Hue75]; we will justify our usage in Section 3.4.

We will not explicitly indicate the types of terms unless it is necessary.

There are well-known effective translations between $\mathcal{LC}$ and $\mathcal{CL}$. For concreteness, we define $\Lambda : \mathcal{CL} \to \mathcal{LC}$ and $\mathcal{H} : \mathcal{LC} \to \mathcal{CL}$ as follows.

Let

- $\Lambda(a) \equiv a$, when a is a non-redex atom,
- $\Lambda(I) \equiv \lambda x.x$,
- $\Lambda(K) \equiv \lambda xy.x$,
- $\Lambda(S) \equiv \lambda xyz.xz(yz)$, and
- $\Lambda(MN) \equiv \Lambda(M)\Lambda(N)$;

and let

- $\mathcal{H}(a) \equiv a$, when a is an atom,
- $\mathcal{H}(PQ) \equiv \mathcal{H}(P)\mathcal{H}(Q)$, and
- $\mathcal{H}(\lambda x.L) \equiv [x]\mathcal{H}(L)$, where
  - $[x]M \equiv KM$ when x does not occur in M,
  - $[x]x \equiv I$,
  - $[x](Mx) \equiv M$ when x does not occur in M,
  - otherwise $[x](MN) \equiv S([x]M)([x]N)$.

On $\mathcal{CL}$, *weak equality* is generated by *weak reduction*, determined by the rules $Ix \longrightarrow x$, $Kxy \longrightarrow x$ and $Sxyz \longrightarrow xz(yz)$. Each of $\beta\eta$-reduction and weak reduction is terminating and confluent on typed terms, so we can speak of *the $\beta\eta$-normal form* of a $\mathcal{LC}$ term and *the weak normal form* of a $\mathcal{CL}$ term.

The translations between $\mathcal{LC}$ and $\mathcal{CL}$ are not translations of the respective *theories*, since weak equality in $\mathcal{CL}$ is too coarse to reflect $\beta\eta$-equality in $\mathcal{LC}$. For instance, the terms SK and KI are distinct weak normal forms, but their translations as $\mathcal{LC}$-terms are $\beta\eta$-equal.

Define *extensional combinatory equality* by

$$P =_c Q \text{ iff } \Lambda(P) =_{\beta\eta} \Lambda(Q).$$

The translations above are such that

$$\Lambda(\mathcal{H}(G)) =_{\beta\eta} G,$$

and it follows that for any $\mathcal{LC}$-terms F and G,

$$F =_{\beta\eta} G \text{ iff } \mathcal{H}(F) =_c \mathcal{H}(G).$$

Substitutions and unification

A *type substitution* is an ordinary algebraic substitution over the algebra of types; a type substitution θ_0 induces a type-shifting mapping on terms in an obvious way, and we shall denote this map by θ_0 as well. A *term substitution* θ_1 is an ordinary (type-preserving) substitution on $\mathcal{LC}$ or $\mathcal{CL}$ terms, as appropriate. A *substitution* θ is a pair consisting of a type substitution θ_0 and a term substitution θ_1; such a pair induces a mapping on $\mathcal{LC}$ and on $\mathcal{CL}$, also denoted θ, by the rule: $\theta T \equiv \theta_1(\theta_0 T)$. (Application of a substitution to a term, as well as composition of substitutions, will be indicated by juxtaposition.) It will be notationally convenient to allow θ_1 to act as the identity on types, so that we may refer to $\theta\alpha$ when α is a type.

These dual substitutions behave in most ways just as ordinary substitutions, but there are many details to be checked. We develop a rudimentary theory of such substitutions in the Appendix.

If $\mathcal{S}$ is a set of type-variables and indeterminates, then $\theta \equiv \theta'[\mathcal{S}]$ means that

1. for every type-variable $t \in \mathcal{S}$, $\theta_0(t) \equiv \theta'_0(t)$, and

2. for every term-variable x whose erasure is in $\mathcal{S}$, $\theta_1(x) \equiv \theta'_1(x)$.

Similarly, $\theta =_c \theta'[\mathcal{S}]$ means that

1. for every type-variable $t \in \mathcal{S}$, $\theta_0(t) \equiv \theta'_0(t)$, and

2. for every term-variable x whose erasure is in $\mathcal{S}$, $\theta_1(x) =_c \theta'_1(x)$.

Define $\theta \leq \theta'[\mathcal{S}]$ to mean that for some substitution η, $\eta\theta \equiv \theta'[\mathcal{S}]$; define $\theta \leq_c \theta'[\mathcal{S}]$ to mean that for some substitution η, $\eta\theta =_c \theta'[\mathcal{S}]$.

The justification for our strategy of translating the unification problem from $\mathcal{LC}$ to $\mathcal{CL}$ is embodied in the following lemma. If σ is an $\mathcal{LC}$-substitution let the $\mathcal{CL}$-substitution $(\mathcal{H} \circ \sigma)$ be defined by: $(\mathcal{H} \circ \sigma)X \equiv (\mathcal{H} \circ \sigma_1)(\sigma_0 X)$. Similarly, if θ is a $\mathcal{CL}$-substitution, let the $\mathcal{LC}$-substitution $(\Lambda \circ \theta)$ be defined by: $(\Lambda \circ \theta)F \equiv (\Lambda \circ \theta_1)(\theta_0 F)$.

Lemma 1.1 *Let F and G be $\mathcal{LC}$-terms. The $\mathcal{LC}$-substitutions σ such that $\sigma F =_{\beta\eta} \sigma G$ are (up to pointwise $\beta\eta$-conversion) those of the form $(\Lambda \circ \theta)$, where θ ranges over the $\mathcal{CL}$-substitutions such that $\theta\mathcal{H}(F) =_c \theta\mathcal{H}(G)$.*

Proof. The proof relies on the following facts, proved in the Appendix: for any $\mathcal{LC}$-term G and substitution σ,

$$\mathcal{H}(\sigma G) \equiv (\mathcal{H} \circ \sigma)\mathcal{H}(G),$$

and for any $\mathcal{CL}$-term Y and substitution θ,

$$\Lambda(\theta Y) \equiv (\Lambda \circ \theta)\Lambda(Y).$$

Suppose $\sigma F =_{\beta\eta} \sigma G$. Then $\mathcal{H}(\sigma F) =_c \mathcal{H}(\sigma G)$. By the first fact above, this means that $(\mathcal{H} \circ \sigma)\mathcal{H}(F) =_c (\mathcal{H} \circ \sigma)\mathcal{H}(G)$. But σ is pointwise $\beta\eta$-convertible with $\Lambda \circ (\mathcal{H} \circ \sigma)$.

Conversely, suppose $\theta\mathcal{H}(F) =_c \theta\mathcal{H}(G)$; we want to show that $(\Lambda \circ \theta)F =_{\beta\eta} (\Lambda \circ \theta)G$. We have $\Lambda(\theta\mathcal{H}(F)) =_{\beta\eta} \Lambda(\theta\mathcal{H}(G)$, and by the second fact above this means that $(\Lambda \circ \theta)\Lambda(\mathcal{H}(F)) =_{\beta\eta} (\Lambda \circ \theta)\Lambda(\mathcal{H}(G))$.

But since $\beta\eta$-conversion is preserved by substitutions, $(\Lambda \circ \theta)\Lambda(\mathcal{H}(F)) =_{\beta\eta} (\Lambda \circ \theta)F$ and $(\Lambda \circ \theta)\Lambda(\mathcal{H}(G)) =_{\beta\eta} (\Lambda \circ \theta)G$, and the result follows. □

If we define *extensional combinatory unification* as the problem of unifying $\mathcal{CL}$ terms with respect to extensional combinatory equality, the above discussion shows how a method for extensional combinatory unification yields a method for higher-order unification as originally presented.

The rest of this paper will be concerned with extensional combinatory equality, henceforth *C-equality*, and extensional combinatory unification, henceforth *C-unification*. The unqualified word "term" will mean "combinatory logic term".

2 C-Validity

Extensional equality can be obtained from weak equality by the addition of the extensionality rule:

$$\text{Infer } M =_c N \text{ from } Mz =_c Nz, \text{ when } z \text{ is not free in } M \text{ or in } N.$$

On the other hand, Curry constructed a set of four *equations* which generate C-equality when added to the defining equations for I, K, and S (see [HS86]). So C-equality has a

presentation as equational theory, in contrast to the presentation using the rule of inference above. Thus C-unification is an instance of general algebraic unification, and would submit to a universal unification procedure, as, e.g., in [GS89a] or [DJ90]. But typed C-equality is decidable, by simply passing to $\mathcal{LC}$ and using (convergent) $\beta\eta$-reduction, so we might hope for a convergent rewrite system in $\mathcal{CL}$ itself. Such systems are typically of fundamental importance as foundations for unification algorithms.

Unfortunately, no convergent rewrite system is known for C-equality. This section addresses this problem by defining a variation on rewriting as a method for determining when terms are C-equal.

By the extensionality rule, deciding C-equality reduces to deciding C-equality between terms whose type is a base type or a type-variable. A weak normal form such a type has a non-redex atom at the head. But two such terms $hM_1 \cdots M_k$ and $h'N_1 \cdots N_{k'}$ are C-equal iff $h \equiv h'$, $k \equiv k'$, and $M_i =_c N_i$ for each i. This suggests that we treat the problem of deciding C-equality between systems of terms, as follows.

Definition 2.1 A *pair* is either a *term-pair* or a *type-pair*, where a term-pair is a two-element multiset of $\mathcal{CL}$-terms and a type-pair is a two-element multiset of types. A pair is *trivial* if its elements are identical, and *valid* if its elements are C-equal (it will be convenient to consider trivial type-pairs to be valid).

A *system* is a multiset of pairs in which no two distinct variables have the same type-erasure; it is *trivial* if each of its pairs is trivial; it is *valid* if each of its pairs is valid. If the symmetric difference of systems Σ and Σ' is trivial, write $\Sigma \cong \Sigma'$.

A consequence of the fact that terms are explicitly typed (as opposed to typable under a type-inference system) is that a pair will not be valid unless its terms have the same type. This restriction is not built into the definition of system since terms of different types may still be unifiable. Type-pairs will play no role until the next section.

The restriction on type-erasures of the variables in a system is designed to avoid the technical complications which would result if distinct variables could become identical after a type-substitution.

Definition 2.2 The set VT consists of the following three reductions.

1. REDUCE

$$\Gamma,\ \langle M,\ N\rangle \longrightarrow \Gamma,\ \langle M',\ N\rangle\,,$$

when M weakly reduces to M'.

2. ADD ARGUMENT

$$\Gamma,\ \langle M,\ N\rangle \longrightarrow \Gamma,\ \langle Md,\ Nd\rangle\,,$$

when M and N have the same type, at least one of M and N is functional, and d is built from the first parameter in $Args$ not occurring in $\langle M,\ N\rangle$, and given the appropriate type.

3. PASSIVE DECOMPOSE

$$\Gamma,\ \langle hM_1 \cdots M_k,\ hN_1 \cdots N_k\rangle \longrightarrow \Gamma,\ \langle M_1,\ N_1\rangle\ \ldots,\ \langle M_k,\ N_k\rangle\,,$$

when h is a non-redex atom.

We adopt the convention that no VT reduction is to be done out of a trivial pair.

The notation for REDUCE exploits the fact that pairs are unordered; we intend of course that either element of a pair may be reduced. A similar remark applies in several places below. The use, in ADD ARGUMENT, of new constants rather than new variables will serve to remind us in unification that the new arguments are not part of the original term and should not be instantiated. The neccessity for the restriction on d in ADD ARGUMENT may be seen by considering the non-valid pair $\langle Kd,\ I\rangle$, which could be reduced to the valid pair $\langle Kdd,\ Id\rangle$ by an improper application of ADD ARGUMENT.

The PASSIVE DECOMPOSE reduction is of course just an application of the standard syntactic unification transformation "Decompose" given in the Appendix, followed by deletion of the trivial pair $\langle h,\ h\rangle$.

We think of VT as a rewriting system for systems of $\mathcal{CL}$ terms, but there are two ways in which this analogy is imperfect. The ADD ARGUMENT reduction may only be applied at the heads of terms, since it changes their type, and PASSIVE DECOMPOSE is not stable under substitution for a head variable. As it happens, though, the facts that rewriting is closed under term-formation and stable under substitution are not relevant to its role in supporting a unification procedure. This will be exploited in the next section. For now we show how VT-reduction can be used to decide C-equality.

Exercise 2.3 Consider the pair

$$\langle SI(SK),\ SI(KI)\rangle\,.$$

An application of ADD ARGUMENT and a weak reduction out of each term in the result yield

$$\langle Id(SKd),\ Id(KId)\rangle\,.$$

More weak reductions result in

$$\langle d(SKd),\ dI\rangle\,,$$

and PASSIVE DECOMPOSE yields

$$\langle SKd,\ I\rangle\,.$$

After an application of ADD ARGUMENT and two weak reductions we have

$$\langle Ke(de),\ e\rangle\,,$$

Finally, weak reductions give the trivial system

$$\langle e,\ e\rangle$$

and (anticipating the next lemma) we conclude that the original terms were C-equal. In fact their $\mathcal{LC}$ translations each $\beta\eta$-reduce to $\lambda x.x(\lambda y.y)$.

Lemma 2.4 *Suppose $\Sigma \longrightarrow \Sigma'$. Then Σ' is valid iff Σ is valid.*

Proof. When the given reduction is REDUCE this is obvious. For an ADD ARGUMENT, invoke extensionality. To see that PASSIVE DECOMPOSE is sound, let M and N be passive, say $M \equiv hM_1\cdots M_k$ and $N \equiv h'N_1\cdots N_{k'}$. Then $\Lambda(M) \equiv h(\Lambda(M_1))\cdots(\Lambda(M_k))$ and $\Lambda(N) \equiv h'(\Lambda(N_1))\cdots(\Lambda(N_{k'}))$. Since $M =_c N$ iff the $\beta\eta$ normal forms of the latter two terms are identical, we see that $M =_c N$ iff $h \equiv h'$, $k \equiv k'$ and for each i, $M_i =_c N_i$. □

Lemma 2.5 *Suppose Σ is* VT*-irreducible. Then Σ is valid iff it is trivial.*

Proof. One direction is immediate. For the other, suppose Σ is valid and irreducible and choose $\langle M,\ N\rangle$ from Σ. Neither M nor N is functional, and since they are in head weak normal form they are both passive. As described in the previous proof, M and N must have identical heads, so PASSIVE DECOMPOSE would apply if M and N were not identical terms. □

Theorem 2.6 *Every sequence of* VT *reductions terminates.*

Proof. Let us say that $M \succ M'$ if either

- M weakly reduces to M',
- $M' \equiv Md$ (for any d of the appropriate type), or
- $M \equiv hM_1 \cdots M_k$, h is a non-redex atom, and and for some i, $M' \equiv M_i$.

Then when a system is identified with the multiset of terms occurring in it, each VT reduction replaces one or two terms by $\succ$-related terms. If we show that the relation $\succ$ is terminating, then the theorem follows by multiset induction.

To show termination of $\succ$ we transfer the problem back into $\mathcal{LC}$. By applying Λ to any sequence of terms obtained by $\succ$ and noting that if $M \longrightarrow wM'$ then $\Lambda(M) \longrightarrow\!\!\!\!\!\rightarrow \beta\Lambda(M')$ in one or more steps, we see that it suffices to show the following relation $>$ to be terminating on $\mathcal{LC}$:

$L > L'$ if either

- $L \longrightarrow \beta L'$,
- $L' \equiv Ld$ (for any d of the appropriate type), or
- $L \equiv hL_1 \cdots L_k$, h is a non-redex atom, and and for some i, $L' \equiv L_i$.

Let the *measure* of an $\mathcal{LC}$-term L be the ordered triple with first element the length of the longest β-reduction out of L, second element the number of symbols in L, and third element the length of the type of L. Order these triples lexicographically and, for sake of contradiction, let L be a term of minimal measure among those admitting an infinite $>$-reduction.

Clearly the first step of an infinite reduction out of L must be an argument-adding step, since the other reductions decrease the measure of a term. Indeed, the reduction must look like some finite number of add-argument steps followed by either a select-argument or a β-reduction.

In the first case L must be of the form $hL_1 \cdots L_k$ and the reduction must look like

$$L \equiv hL_1 \cdots L_k \gg hL_1 \cdots L_k d_1 \cdots d_n > G > \cdots$$

where G is some L_i, $0 \leq i \leq k$, or some d_j, $1 \leq j \leq n$. But then either L_i or d_j admits an infinite $>$-reduction, and these terms have smaller measure than L, contradicting the minimality of L.

In the second case, either the β-reduction involves subterms from L itself or L is an abstraction and one of the added arguments is the argument in the β-redex. In the first instance, we can clearly do the β-reduction first and so contradict the minimality of L. Otherwise we have:

$$L \equiv \lambda x.A \gg (\lambda x.A)d_1 \cdots d_k > A[x := d_1]d_2 \cdots d_k > \cdots$$

Thus $A[x := d_1]$ admits an infinite $>$-reduction. But observe that $>$-reduction is preserved under the operation of replacing an Arg-constant by a variable. That is, for any C and D, if $C[x := d_1] > D[x := d_1]$ then $C > D$. This implies that A itself admits an infinite $>$-reduction. But the measure of A is less than the measure of $\lambda x.A$, since any β-reduction sequence out of $\lambda x.A$ induces one of the same length out of A, and we again have a contradiction. □

A simple algorithm for deciding C-equality between terms M and N can apply VT reductions in any order to the system originally containing $\langle M,\ N\rangle$. Since every VT sequence terminates, an irreducible system will be obtained; $M =_c N$ iff this system is trivial. Of course, we may halt and report non-equality if we ever generate a pair of passive terms with different heads.

Observe that the proof of Lemma 2.5 above relied only on the assumption that the terms M and N admit no weak *head* reductions. It follows that if we were to restrict REDUCE to applications at the heads of terms then VT would still lead to a decision procedure complete for C-validity. This observation will enable us to correspondingly restrict the search space in our unification procedure.

Although C-equality apparently does not have a presentation as a rewrite system over $\mathcal{CL}$, the technique above could be recast as a reduction relation over an expanded set of terms, by introducing a family of equality operators *eq* to mimic pair-formation, a family of conjunction operators to form system-terms, and a constant *tt* to which identical-pair terms reduce. Then the VT rules together with some bookkeeping rules (such as collapsing conjunctions of *tt*) induce a convergent reduction on system-terms which reduces a term corresponding to a system Σ to *tt* iff Σ is valid.

3 C-Unification

Our C-unification method is simply an elaboration of the point of view suggested in the previous paragraph.

In first-order E-unification (unification relative to a set E of algebraic equations), *narrowing* is a method for generating E-unifiers for a pair of terms $\langle A,\ B\rangle$ which is complete when when E admits a presentation as a convergent rewrite system R. It proceeds as follows: select from R the left-hand side of an equation $S = T$ and from $\langle A,\ B\rangle$ a non-variable subterm, say, A/u, such that A/u and S are syntactically unifiable with most general syntactic unifier σ_0. Apply σ_0 to A, perform the rewrite step using $\sigma_0 S = \sigma_0 T$ and continue constructing and applying such substitutions σ_i until a pair with a most general syntactic unifier σ_n is derived. The composition of the σ_i provides an E-unifier. (When unification is presented in terms of transformations on systems, the answer substitution is automatically built up at each step as part of the modified system.)

Narrowing is sometimes developed in a framework which starts with a convergent rewrite system R for E and introduces a new function symbol eq. Given terms X and Y to be E-unified, we attempt to narrow the term $eq(X,Y)$ to a term $eq(Z,Z')$ in which Z and Z' are syntactically unifiable. A key point is that R is still convergent on the expanded set of terms.

But of course it is not necessary to *start* with a convergent reduction; what matters is the reduction on the paired terms. In our situation, we cannot start with a notion of reduction defined on the terms to be C-unified. But when we pass to the expanded set of terms (more precisely, systems), we then have a reduction relation available. The present section will establish its suitability as a foundation for C-unification.

Definition 3.1 A substitution θ is a *unifier* of a system Σ if $\theta\Sigma$ (obtained by applying θ to each type and term occurring in Σ) is trivial. A *most general unifier* of a system Σ is an idempotent unifier σ such that (i) $D(\sigma_0) \cup D(\sigma_1) \subseteq Supp(\Sigma)$, (ii) the type-erasures of the constants introduced by σ all occur as type-erasures of constants in Σ, and (iii) for all unifiers θ of Σ, $\sigma \leq \theta$.

A substitution θ is a *C-unifier* of a system Σ if $\theta\Sigma$ is valid.

Unifiers will sometimes be referred to as *syntactic* unifiers to emphasize the contrast with C-unifiers.

We write $mgu(\Sigma)$ to stand for any most general (syntactic) unifier of system Σ. In the Appendix we show that syntactically unifiable sytems possess most general unifiers.

Definition 3.2 Let Σ be a system. If $\langle t,\ \alpha\rangle$ is a type-pair in Σ and there are no occurences (in type- or term-pairs) of t in Σ other than the one indicated, then t is *solved* in Σ and $\langle t,\ \alpha\rangle$ is a *solved type-pair*. If $\langle x,\ A\rangle$ is a term-pair in Σ, x and A have the same type, and there are no occurences of x in Σ other than the one indicated, then x is *solved* in Σ and $\langle x,\ A\rangle$ is a *solved term-pair*.

If each non-trivial term- or type-pair in Σ is solved, then Σ is a *solved system*.

If Σ is a solved system its non-trivial pairs determine an idempotent substitution in an obvious way, although a pair consisting of two solved variables requires a choice as to which of them is to be in the domain of the substitution. Similarly, an idempotent substitution can be represented as a solved system (without trivial pairs). If σ is an idempotent substitution, write $[\sigma]$ for the solved system which represents it.

The fundamental connection between solved systems and unifiers is the fact that if $[\sigma]$ is a solved sytem then σ is a most general unifier of $[\sigma]$. This was observed by Martelli and Montanari [MM82] in the context of syntactic unification and is proved in the Appendix for the present situation. Transformation-based unification methods attempt to reduce systems to solved forms, from which solutions may be extracted immediately.

Since substitutions may instantiate types as well as terms, a system may be unifiable even when some pairs consist of terms with different types. By appropriate type-unifications we could insist that the terms in each pair have the same type, without sacrificing completeness of the method, but it seems more efficient to type-unify only when necessary; these type-unifications are embedded in the transformations below.

Definition 3.3 The set **UT** is obtained by adding the following three transformations to the transformations for syntactic unification. (The latter are found in the Appendix.)

1. *Narrow:*

$$\Gamma,\, \langle X,\ Y\rangle \Longrightarrow [\mu],\ \mu\Gamma,\, \langle \mu X^*,\ \mu Y\rangle\,,$$

where there exists a non-variable subterm ocurrence U of X and a combinatory weak reduction rule $L \to R$ with fresh variables such that L and U have most general unifier μ, and X^* is obtained from X by substituting R for U.

2. *Add Argument:*

$$\Gamma,\, \langle X,\ Y\rangle \Longrightarrow [\mu],\ \mu\Gamma,\, \langle (\mu X)d,\ (\mu Y)d\rangle\,,$$

where $\mu \equiv (\pi \to \pi')$ is a most general type-unifier of the set consisting of: the type of X, the type of Y, and (just in case these are each atomic types) the type $(s \to t)$, for fresh type-variables s and t, and where d is built from the first fresh parameter in $Args$, given type π.

3. *Split:*

$$\Gamma,\, \langle xX_1 \cdots X_n,\ hZ_1 \cdots Z_m Y_1 \cdots Y_n\rangle \Longrightarrow$$
$$[\mu],\ \mu\Gamma,\, \langle z_1,\ \mu Z_1\rangle, \ldots, \langle z_m,\ \mu Z_m\rangle\,,\, \langle \mu X_1,\ \mu Y_1\rangle, \ldots, \langle \mu X_n,\ \mu Y_n\rangle\,,$$

where $m, n \geq 0$, $x \in Vars$, h is a pure atom, each z_i is a fresh indeterminate given the same type as Z_i, $1 \leq i \leq m$, and μ is a most general unifier of x and $hz_1 \cdots z_m$

It is important to note that in transformations *Narrow* and *Split*, the computation of the unifiers μ implicitly involves some type-unification.

We adopt the convention that no **UT** transformation is to be done out of a solved or trivial pair. This respects the intuition that the solved part of a system is merely a record of an answer substitution being constructed.

An implementation of **UT** would presumably not treat *Add Argument* as a separate transformation, but would rather incorporate it into a more generous version of *Narrow* which supplies arguments as needed. It is easier to analyze the transformations separately, though, and we want to emphasize the fact that the **UT** transformations are immediately derived from the VT reductions.

After establishing some basic properties of the transformations we will show that it is possible to impose a certain discipline on applications of the rules without sacrificing completeness.

We will need to be careful about the set of variables occurring in a system. If $\Sigma \Longrightarrow \Sigma'$ then $Supp(\Sigma) \subseteq Supp(\Sigma')$ (see Remark 5.5). In addition, solved variables remain solved after a transformation, that is, if $\Sigma \Longrightarrow \Sigma'$ then $\{x | x \text{ is solved in } \Sigma\} \subseteq \{x | x \text{ is solved in } \Sigma'\}$. This is easily checked; it relies on the conventions that transformations are not performed on solved pairs, and that distinct terms do not have the same type erasure (the latter ensures that distinct variables are not identified after application of a type-substitution).

Theorem 3.4 (Soundness) *If $\Sigma \Longrightarrow \Sigma'$ and $\theta\Sigma'$ is valid, then $\theta\Sigma$ is valid.*

Proof. Use the notation of Definition 3.3. Our hypothesis entails that $\theta[\mu]$ is valid, so $\mu \leq_c \theta$ and $\theta\mu =_c \theta$. Thus $\theta\mu\Gamma =_c \theta\Gamma$, and so we need only show that θ C-unifies the "redex pair" of the transformation.

When the transformation is *Narrow*, we observe that $\mu X =_c \mu X^*$. Thus $\theta X =_c \theta\mu X =_c \theta\mu X^* =_c \theta\mu Y =_c \theta Y$, as desired.

When the transformation is *Add Argument* we want to see that $\theta X =_c \theta Y$. But $\theta\mu(Xd) =_c \theta\mu(Yd)$, that is, $(\theta X)(\theta d) =_c (\theta Y)(\theta d)$ and we may invoke the extensionality rule since θd is guaranteed to be new to $\langle \theta X,\ \theta Y \rangle$.

In the case of *Split*, the fact that $\theta X_i =_c \theta\mu X_i =_c \theta\mu Y_i =_c \theta Y_i$ for $1 \leq i \leq n$ implies that we need only argue that $\theta \langle x,\ hZ_1 \cdots Z_m \rangle$ is valid. We compute: $\mu x \equiv \mu h z_1 \cdots z_m$ by definition of μ, so $\theta x =_c \theta\mu x \equiv \theta\mu(hz_1 \cdots z_m) =_c \theta(hz_1 \cdots z_m)$, but our hypothesis implies that for each i, $\theta z_i =_c \theta Z_i$. □

We now address completeness.

3.1 The main lemma

The Lifting Lemma below is the key step in showing that for any Σ, **UT** can enumerate a complete set of C-unifiers for Σ. It is convenient to isolate a notion of C-unifier involving certain technical conditions. First, in order to enforce the idea that constants from $Args$ are not part of our unification problems but are introduced only as dummy arguments, we focus on answer substitutions θ such that each θx is a pure term (call these *pure substitutions*). This means that must confine attention to pure problems, but of course any problem can be considered a pure one by suitably defining $Args$, if necessary, to be the erasures of those constants not occurring in the input. Second, we slightly weaken the customary requirement that substitutions map each variable to a normal form.

Definition 3.5 A $\mathcal{CL}$ term is a *strong normal form* [CF58] if it is $\mathcal{H}(L)$ for a $\mathcal{LC}$ term L in $\beta\eta$-normal form. In order to maintain a consistent notation we will refer to terms in strong normal form as being *C-normal*.

A pure idempotent substitution θ is a *normalized C-unifier of* Σ if

1. $D(\theta_0) \cup D(\theta_1) \subseteq Supp(\Sigma)$,
2. $\theta\Sigma$ is valid, and
3. for each variable x not solved in Σ, θx is C-normal.

Write NCU(Σ) for the set of normalized C-unifiers of Σ.

If we say only that θ is a "C-unifier" of system Σ, we mean only that $\theta\Sigma$ is valid.

It is clear that each $\mathcal{CL}$-term is C-equal to a unique C-normal term, and that C-normal terms are irreducible with respect to weak reduction. It also true that a subterm of a C-normal form is C-normal. This last seems difficult to prove directly, but we may appeal to classical results on $\mathcal{CL}$. Curry [CF58] defined a notion of *strong reduction* on $\mathcal{CL}$-terms and it was proved ([CF58], [Ler67]) that the C-normal forms are precisely the terms which are irreducible with respect to this reduction (see also [HL70]). Since strong reduction is a congruence with respect to the term-forming operations, the class of irreducibles under strong reduction (i.e., the class of C-normal forms) is closed under subterm.

Lemma 3.6 (Lifting Lemma) *Let $\theta \in NCU(\Sigma)$ and let $\langle X,\ Y \rangle$ be an unsolved pair in Σ. If*

$$\theta\Sigma \longrightarrow \Delta$$

is a VT *step out of $\langle \theta X,\ \theta Y \rangle$, then there exists a Σ' and θ' with*

$$\Sigma \Longrightarrow \Sigma',$$

such that

1. $\theta' \equiv \theta[\mathrm{Supp}(\Sigma)]$,

2. $\theta'\Sigma' \cong \Delta$,

3. $\theta' \in NCU(\Sigma')$.

Proof. Write Σ as $\Gamma, \langle X, Y\rangle$. Since $\langle X, Y\rangle$ is not solved, θ is C-normal on the variables of X and Y.

In case Δ is obtained by REDUCE, we have

$$\theta\Sigma \equiv \theta\Gamma,\ \theta\langle X, Y\rangle \longrightarrow \theta\Gamma,\ \langle(\theta X)', \theta Y\rangle \equiv \Delta.$$

Suppose that $(\theta X)'$ is obtained from θX by a combinatory weak reduction rule $L \to R$ with fresh variables, replacing, in (θX), subterm $A \equiv \delta L$ by δR.

A is of the form θU for a subterm occurrence U of X; since θ is pointwise weakly normal on the variables of X, U is not a variable. Letting μ be a most general unifier of U and L and constructing X^* by substituting R for U, the following is a *Narrow* step, defining Σ':

$$\Gamma, \langle X, Y\rangle \Longrightarrow [\mu],\ \mu\Gamma,\ \langle \mu X^*,\ \mu Y\rangle.$$

Take θ' to be $\theta \cup \delta$. Since the variables of L are fresh, $\theta' \equiv \theta[Supp(\Sigma)]$.

To check that $\theta'\Sigma' \cong \Delta$, observe that since θ' unifies U and L, $\mu \leq \theta'$, so that $\theta'[\mu]$ is trivial and $\theta'\mu \equiv \theta'$. We need only show that $\theta'\mu X^* \equiv (\theta X)'$. But X^* is X with U replaced by R, so $\theta'\mu X^*$ is $\theta'\mu X$ with $\theta'\mu U$ replaced by $\theta'\mu R$. That is, $\theta'\mu X^*$ is θX with θU replaced by δR, which is indeed $(\theta X)'$.

To verify that $\theta' \in NCU(\Sigma')$, first note that $\theta'\Sigma'$ is valid since $\theta'\Sigma' \cong \Delta$ and VT reductions preserve validity. Since θ' is a most general unifier of pure terms, it is pure. Now let z be an unsolved variable of Σ'; we show that $\theta' z$ is normal. Such a z is either a variable from Σ or is introduced by μ. If z is from Σ then z was unsolved there, and θ' agrees with θ on z. Suppose z is introduced by μ. Then z is a variable in μU, that is, for some x in U, z is in μx. This implies that $\theta' z$ is a subterm of $\theta'\mu x$. But $\theta'\mu x \equiv \theta x$, which is normal, and subterms of normal terms are normal.

In case Δ is obtained by ADD ARGUMENT, we have:

$$\theta\Sigma \equiv \theta\Gamma,\ \langle \theta X, \theta Y\rangle \longrightarrow \theta\Gamma,\ \langle (\theta X)(e),\ (\theta Y)(e)\rangle \equiv \Delta;$$

Writing the type of θX as $(\alpha \to \beta)$, let the type of X be τ_1 and the type of Y be τ_2, and in case these last two are each atomic types let $(s \to t)$ be the type introduced as in the definition of *Add Argument*. An application of *Add Argument* yields Σ':

$$\Gamma,\ \langle X, Y\rangle \Longrightarrow [\mu],\ \mu\Gamma,\ \langle (\mu X)(d),\ (\mu Y)(d)\rangle;$$

Choose θ' to be $\theta \cup \delta$, where $\delta_0 \equiv \{s := \alpha, t := \beta\}$ and δ_1 is the identity. Clearly $\theta' \equiv \theta[Supp(\Sigma)]$.

To verify that $\theta'\Sigma' \cong \Delta$, first observe that θ'_0 unifies τ_1, τ_2, and, if applicable, $(s \to t)$, since it maps each of these to $(\alpha \to \beta)$. So $\mu \leq \theta'_0$, and $\theta'[\mu]$ is trivial. Furthermore $\theta'\mu \equiv \theta'$, so that $\theta'\mu$ and θ agree on Γ, X, and Y. Finally, we argue that $\theta' d \equiv e$. First note that the type of e is α, and since $\theta'\mu X \equiv \theta X$ has type $(\alpha \to \beta)$, $\theta' d$ will also have type α. Furthermore, since θ is pure we know that Σ and $\theta\Sigma$ involve precisely the same *Args*-parameters, and so d has the same type-erasure as e (they were each chosen to be the first fresh *Args*-parameter available).

To see that $\theta' \in NCU(\Sigma')$, first note that $\theta'\Sigma'$ is valid as before, then observe that θ' is appropriately normal since no new unsolved term-variables appear in Σ'. It is clear that θ' is pure.

In case Δ is obtained by PASSIVE DECOMPOSE, we have two subcases. If θX and θY have the same constant at the head, then X and Y also have these constants at the head, and we may obtain Σ' by applying the syntactic unification transformation Decompose to $\langle X,\ Y\rangle$, and take θ' to be θ. Otherwise, we can describe $\langle X,\ Y\rangle$ and $\langle \theta X,\ \theta Y\rangle$ as follows.

$$\langle X,\ Y\rangle \equiv \langle xX_1 \cdots X_n,\ hZ_1 \cdots Z_m Y_1 \cdots Y_n\rangle\,,$$

where $m, n \geq 0$, $x \in Vars$, and h is a pure atom, while

$$\langle \theta X,\ \theta Y\rangle \equiv \langle aA_1 \cdots A_k B_1 \cdots B_m M_1 \cdots M_n,\ aA_1 \cdots A_k C_1 \cdots C_m N_1 \cdots N_n\rangle\,,$$

for some $k \geq 0$, with

$$aA_1 \cdots A_k B_1 \cdots B_m \equiv \theta x,$$

$$aA_1 \cdots A_k \equiv \theta h,$$

$$C_i \equiv \theta Z_i,\ 1 \leq i \leq m,$$

$$M_i \equiv \theta X_i,\ 1 \leq i \leq n,\ \text{and}$$

$$N_i \equiv \theta Y_i,\ 1 \leq i \leq n.$$

The repetition of the A_i is justified by the facts that θ is C-normal on the variables of X and Y and C-normal terms are unique in their C-equivalence class, and the assertion that h cannot be a constant from $Args$ follows from the fact that θ is a pure substitution.

We obtain Σ' by applying *Split*:

$$\Gamma,\ \langle xX_1 \cdots X_n,\ hZ_1 \cdots Z_m Y_1 \cdots Y_n\rangle \Longrightarrow$$

$$[\mu],\ \mu\Gamma,\ \langle z_1,\ \mu Z_1\rangle, \ldots, \langle z_m,\ \mu Z_m\rangle,\ \langle \mu X_1,\ \mu Y_1\rangle, \ldots, \langle \mu X_n,\ \mu Y_n\rangle\,,$$

where μ is the most general unifier of x and $hz_1 \cdots z_m$. Take θ' to be $\theta \cup \delta$, where δ_0 is the identity and $\delta_1 \equiv \{z_1 := B_1, \ldots, z_m := B_m\}$. As before, $\theta' \equiv \theta[Supp(\Sigma)]$.

To check that $\theta'\Sigma' \cong \Delta$, we first see that θ' unifies x with $hz_1 \cdots z_m$, since applying θ' to each yields $aA_1 \cdots A_k B_1 \cdots B_m$. So $\theta'\mu \equiv \theta'$ and the pairs of $\theta'\Sigma'$ match the pairs of Δ, except that the trivial sub-system $\theta[\mu]$ does not appear in Δ and, when $k > 0$, Σ' will not include pairs corresponding to the $\langle A_i,\ A_i\rangle$ in Δ.

As usual, $\theta'\Sigma'$ is valid, and the fact that the B_i are pure and C-normal yield purity and C-normality for θ'; hence $\theta' \in NCU(\Sigma')$. □

Using the Lifting Lemma, we may show that **UT** transformations can enumerate a complete set of C-unifiers for any system. But we would like to constrain the nondeterminism inherent in such a method as much as possible, so we explore some refinements of the process before giving a completeness proof.

3.2 Refinements

We begin by observing that certain VT reductions preserve the set of C-unifiers of a system. Call an application of PASSIVE DECOMPOSE out of a pair of rigid terms a *rigid/rigid* step.

Definition 3.7 A system is *simple* if each term in the system is a passive weak normal form, and there is no pair of rigid terms with identical heads.

Lemma 3.8 *Any sequence of* REDUCE, ADD ARGUMENT, *and rigid/rigid* PASSIVE DECOMPOSE *steps applied to a system will terminate in a simple system with the same C-unifiers.*

Proof. It is easy to see that if $\Sigma \longrightarrow \Sigma'$ by a REDUCE, ADD ARGUMENT, or rigid/rigid PASSIVE DECOMPOSE step then Σ and Σ' have the same C-unifiers. The fact that VT is terminating completes the proof. □

Simple systems are those which are irreducible with respect to the VT-reductions of Lemma 3.8, and we need only apply **UT** transformations to simple systems. This is the sense in which the method to be presented is a "normalized narrowing" algorithm.

Since VT reductions are not to be done out of trivial pairs (indeed, such pairs may be deleted from a system with no consequences for validity or C-unification), it will be more efficient to perform rigid/rigid PASSIVE DECOMPOSE steps as soon as they become possible.

As observed in the previous section, one can confine applications of VT reductions to the heads of terms. This suggests that one can similarly restrict applications of *Narrow*.

Definition 3.9 The set of transformations **HUT** consists of *Head-Narrow*, *Split*, and *Add Argument*.

Here, a *Head-Narrow* transformation is a *Narrow* transformation corresponding to a weak reduction at the head of a term.

There are five possible patterns for *Head-Narrow*, which we indicate as follows; we discuss the types of terms in a moment.

Head-Narrow:

$$\Gamma,\ \langle xU\vec{Z},\ Y\rangle \Longrightarrow [\mu], \mu\Gamma, \mu\,\langle U\vec{Z},\ Y\rangle\,, \text{ where } \mu \text{ unifies } x \text{ and } I.$$

$$\Gamma,\ \langle xUV\vec{Z},\ Y\rangle \Longrightarrow [\mu], \mu\Gamma, \mu\,\langle U\vec{Z},\ Y\rangle\,, \text{ where } \mu \text{ unifies } x \text{ and } K.$$

$$\Gamma,\ \langle xV\vec{Z},\ Y\rangle \Longrightarrow [\mu], \mu\Gamma, \mu\,\langle z\vec{Z},\ Y\rangle\,, \text{ where } \mu \text{ unifies } x \text{ and } Kz.$$

$$\Gamma,\ \langle xUVW\vec{Z},\ Y\rangle \Longrightarrow [\mu], \mu\Gamma, \mu\,\langle UW(VW)\vec{Z},\ Y\rangle\,, \text{ where } \mu \text{ unifies } x \text{ and } S.$$

$$\Gamma,\ \langle xVW\vec{Z},\ Y\rangle \Longrightarrow [\mu], \mu\Gamma, \mu\,\langle zW(VW)\vec{Z},\ Y\rangle\,, \text{ where } \mu \text{ unifies } x \text{ and } Sz.$$

$$\Gamma,\ \langle xW\vec{Z},\ Y\rangle \Longrightarrow [\mu], \mu\Gamma, \mu\,\langle z_1W(z_2W)\vec{Z},\ Y\rangle\,, \text{ where } \mu \text{ unifies } x \text{ and } Sz_1z_2.$$

Of course, the possibilities for transformations are limited more than the notation suggests, since they are constrained by typing considerations. For example, the first pattern for *Head-Narrow* above can only be executed if the type of U is such that U can go at the head of the sequence $\vec{Z}$. Similar remarks hold for the other patterns.

It is interesting here to consider the "classical" higher-order unification situation, in which types do not have variables. Syntactic unification and narrowing then make reference to term-variables only, and of course all of our results apply to this situation. But in the transformation where x is bound to Sz_1z_2 (and in this transformation only!) the type of the new variables z_1 and z_2 are not uniquely determined by the type of x, and in fact there are infinitely many possibilities for these types. This implies that the search space of our procedure is infinite-branching when types must be fully specified. I am indebted to Ullrich Hustadt for this observation on an early version of this paper.

But since our substitutions are allowed to act on type variables we can postpone our commitment to the types of z_1 and z_2: letting the type of x be $\alpha \to \gamma$, the unification of x with Sz_1z_2 results in type $\alpha \to t \to \gamma$ for z_1 and type $\alpha \to t$ for z_2, where t is a fresh type-variable. The type variable t might become instantiated later, in the course of the type-unification inherent in the transformations.

Exercise 3.10 We illustrate the use of the *Head-Narrow* transformations. Suppose that s is a type-variable and 0 is a base type; we construct some of the (infinitely many) C-unifiers of the following pair:

$$\langle fg,\ b\rangle\,,$$

in which f is a term-variable with type $(s \to 0)$, g is a term-variable with type s, and b is a constant with type 0.

If we bind f to Kz we arrive at:

$$\langle f,\ Kz\rangle\,,\ \langle z,\ b\rangle\,,$$

where z has type 0; syntactic unification yields the solution binding f to Kb. Note that neither s nor g is constrained in this solution.

Returning to the original system: in looking for more solutions we are blocked from attempting to bind f to K since the respective types do not unify. Applying the *Head-Narrow* transformation in which f is bound to I has the effect of binding s to 0 and leads immediately to a solved system:

$$\langle s,\ 0\rangle\,,\ \langle f,\ I\rangle\,,\ \langle g,\ b\rangle\,.$$

Again returning to the original system: typing considerations forbid binding f to S or to Sz, but binding f to Sz_1z_2 is available:

$$\langle f,\ Sz_1z_2\rangle\,,\ \langle z_1g(z_2g),\ b\rangle\,.$$

Here, z_1 has type $(s \to t \to 0)$ and z_2 has type $(s \to t)$ for a fresh type-variable t. At the next step we can bind z_1 to I, forcing s to be $(t \to 0)$, yielding:

$$\langle f,\ SIz_2\rangle\,,\ \langle g(z_2g),\ b\rangle\ \langle s,\ (t \to 0)\rangle\,.$$

Finally, binding g to Kb results in the solved system whose non-trivial part is:

$$\langle f,\ SIz_2\rangle\,,\ \langle g,\ Kb\rangle\ \langle s,\ (t \to 0)\rangle\,.$$

3.3 The algorithm

Definition 3.11 The non-deterministic algorithm $\mathcal{U}$ is the following process:
Repeatedly:

1. Reduce the system to a simple system then apply some **HUT** transformation out of an unsolved pair.

2. If at any point the system is syntactically unifiable by a pure substitution then *optionally* return a most general unifier of the system.

In contrast to syntactic unification, a semantic unification procedure cannot necessarily simply transform systems to syntactically unifiable ones, since some semantic unifiers may be more general than the most general syntactic unifier. For C-unification, an example is provided by the pair $\langle Kax,\ Kay\rangle$, in which the identity substitution is a C-unifier but not a syntactic unifier. This explains the non-determinism in step 2 of the algorithm below. (On the other hand, it is true that if Σ is a *solved* system the substitution associated with Σ may be returned, as shown by Lemma 5.3.)

Observe that if at any point there is a pair of passive terms whose heads are constants with different type-erasures or with types which do not unify, then the current system is not C-unifiable.

It follows from Theorem 3.4 and Lemma 3.8 that if Algorithm $\mathcal{U}$ is run with initial system Σ and returns substitution θ then θ is a C-unifier of Σ. The main result of this paper, Theorem 3.13, is a converse.

We first isolate a technical Lemma justifying the restriction to unsolved pairs when applying transformations. Note that any system Σ can be written as $\Gamma, [\sigma]$ where $[\sigma]$ is the set of solved pairs in Σ; we refer to $[\sigma]$ as the solved part of Σ.

Lemma 3.12 *Suppose that Σ is syntactically unifiable. If θ is a C-unifier of Σ and a syntactic unifier of the unsolved part of Σ, then $mgu(\Sigma) \leq_c \theta$.*

Proof. Let the solved and unsolved parts of Σ be $[\sigma]$ and Γ respectively. We first claim that if γ is $mgu(\Gamma)$, then $\gamma\sigma$ is $mgu(\Sigma)$. Certainly $\gamma\sigma[\sigma]$ is trivial, and the fact that $\gamma\sigma\Gamma$ is trivial follows from the fact that σ is the identity on Γ. So $\gamma\sigma$ is a unifier. To see that $\gamma\sigma$ is most general, let δ be any unifier of $[\sigma], \Gamma$; it suffices to show that $\delta\gamma\sigma \equiv \delta$. But $\delta\gamma \equiv \delta$ since δ unifies Γ, and $\delta\sigma \equiv \delta$ since δ unifies $[\sigma]$.

Next, since θ unifies Γ, $\gamma \leq \theta$, and so $\gamma\sigma \leq \theta\sigma$. But since θ C-unifies $[\sigma]$ and σ is idempotent, $\theta\sigma =_c \theta$. □

Theorem 3.13 (Completeness) *Let θ be a pure C-unifier of Σ. Then there is a computation of Algorithm $\mathcal{U}$ on Σ producing a pure C-unifier δ of Σ with $\delta \leq_c \theta[\mathrm{Supp}(\Sigma)]$.*

Proof. Since every pure C-unifier of Σ is pointwise C-equal to a normalized C-unifier of Σ, we may prove the theorem under the additional hypothesis that $\theta \in NCU(\Sigma)$.

Let the *degree* of a system be the maximum length of a VT sequence out of it. The proof is by induction on the degree of $\theta\Sigma$.

If θ is a unifier of the unsolved part of Σ, then Σ is unifiable and Algorithm $\mathcal{U}$ can return a most general unifier δ. Lemma 3.12 assures us that $\delta \leq_c \theta$. This situation obtains if the degree of $\theta\Sigma$ is 0.

Otherwise we define a system Σ' and a substitution θ' as follows.

1. If Σ is not simple, apply a VT step to obtain Σ' and let θ' be θ.

2. Otherwise there exists an unsolved $\langle X,\ Y\rangle$ from Σ so that $\theta X \not\equiv \theta Y$ and a VT-step out of $\langle \theta X,\ \theta Y\rangle$ (at the head, if it is to be a weak reduction) yielding Δ. The Lifting Lemma applies, yielding Σ' and θ'.

In each case the action performed is a $\mathcal{U}$ step, $\theta' \in NCU(\Sigma')$, and the degree of $\theta'\Sigma'$ is less than the degree of $\theta\Sigma$ (using the facts that $\theta'\Sigma' \cong \Delta$ in case 2 and that no VT steps are ever done out of trivial pairs).

By induction, there is a computation of Algorithm $\mathcal{U}$ on Σ' producing a C-unifier δ of Σ' with $\delta \leq_c \theta'[Supp(\Sigma')]$. By Theorem 3.4, δ is a C-unifier of Σ. Since $Supp(\Sigma) \subseteq Supp(\Sigma')$, $\delta \leq_c \theta'[Supp(\Sigma)]$. But since $\theta' \equiv \theta[Supp(\Sigma)]$, $\delta \leq_c \theta[Supp(\Sigma)]$ as desired. □

3.4 Comparison with Huet's algorithm

Huet [Hue76] calls an $\mathcal{LC}$ term $\lambda x_1, \ldots, x_n.hL_1 \cdots L_k$ *flexible* if h is a variable not among $x_1, \ldots, x_n$, and *rigid* otherwise. Our use of these terms is consistent with his: it is easy to check that a $\mathcal{LC}$ term L is flexible [respectively, rigid] iff $\mathcal{H}(L)$ reduces to a flexible [rigid] term by weak reductions and argument-adding steps.

Under this association our simple systems correspond to the disagreement sets which are produced by Huet's SIMPL algorithm. So there is a sense in which the SIMPL phase of Huet's algorithm is the normalization phase of a narrowing algorithm. In any event, the correspondence between Huet's notion of flexible pair and ours suggests a correspondence between the two notions of *pre-unification*. But the close correspondence is rather between *pre-solved systems*. The respective processes of pre-unification are quite different, and their relationship deserves further study.

This invites a comparison between Huet's MATCH algorithm and our use of the **HUT** transformations. One important difference is that there are finitely many **HUT** transformations possible out of any system (even one with pairs of flexible terms), essentially because a finite set of combinators is powerful enough to simulate the behavior of arbitrary $\mathcal{LC}$-terms.

It is interesting to see how Huet's algorithm, in particular, the step there called "Projection", fares when type-variables are present. Consider a pair $\langle xM_1 \cdots M_r,\ aN_1 \cdots N_p\rangle$ of terms of the same type τ. When some M_i has type $(\tau_1 \to \cdots \to \tau_k \to \tau)$, the i^{th} Projection step introduces the partial substitution $x := \lambda\vec{w}.w_i(h_1\vec{w}) \cdots (h_k\vec{w})$, where $\vec{w}$ is a sequence of new variables $w_1, \ldots, w_r$ corresponding to $M_1, \ldots, M_r$, and the h_j are new variables. But in our present setting the type of M_i may look like $(\tau_1 \to \cdots \to \tau_k \to s)$ where s is a type-variable. In this case there are infinitely many instantiations of the type of M_i corresponding to functions with result-type τ, and it is not clear how to account for the infinitely many relevant Projection steps . Put simply, we cannot know how many h_j to use in the binding for x. This point is made in [Nip90], where an (incomplete) extension to Huet's algorithm is presented. The classical algorithm makes essential use of the "shapes" of types, information which is not available when types are incompletely determined. In contrast, the approach presented here is driven by the shapes of terms.

Example 3.10 addresses precisely this situation. In fact the solutions developed there other than the one binding f to Kb are those which would be derived by Huet's algorithm

on the corresponding lambda terms by instances of Projection when (in the notation of the previous paragraph) $k = 0$ and $k = 1$.

Huet showed that any enumeration of complete sets of higher-order unifiers must be redundant. That is, there are pairs of $\mathcal{LC}$ terms such that any complete set of $\mathcal{LC}$-unifiers of the pair must contain distinct substitutions σ_1 and σ_2 with $\sigma_1 \leq_{\beta\eta} \sigma_2$. Of course C-unification inherits this property, and the current version of our Algorithm $\mathcal{U}$ can return redundant unifiers. For instance, in Example 3.10, if after binding f to Sz_1z_2 we had bound z_1 to Kx and then bound x to Kb, we would derive the solution binding f to $S(K(Kb))z_2$, which is C-equal to Kb.

Huet's method of preunification, which does not attempt to be complete, is irredundant.

4 Directions for further research

- Refining our algorithm to alleviate redundancy among the solutions generated seems to be an important goal if the method is to be used in practice. In addition, the relationship between our notion of preunification and Huet's deserves careful examination.

- The use of additional combinators (e.g., B and C, where $Bfgx = f(gx)$ and $Cfxy = fyx$) allows very convenient optimizations when compiling lambda expressions into combinators [Tur79], but it is not clear how their introduction would affect the performance of the method based on I, K, and S. Note that the addition of more primitives increases the number of possible *Narrow* steps out of any term, but each such step is an abbreviation for several I, K, S steps; in a sense this represents a flattening of the search tree.

- The ability to work with type-substitutions in our explicitly typed setting suggests that our approach might be generalized to deal with richer explicitly typed systems.

- It is natural to ask: what happens if we pass to type-inference rather than explicit typing? Higher-order unification in this setting seems much harder than in the explicitly typed problem, essentially because type inference does not interact well with $\beta\eta$-conversion (as opposed to $\beta\eta$-reduction). The following example, adapted from [Hin69], gives some insight into the difficulty; it applies to both $\mathcal{LC}$ and $\mathcal{CL}$.

  It is proved in [Hin69] that if we can infer that X has type α in the inference system for simple types, then there is an X' such that X' reduces to X and X' has principal type α. Now let τ_1 and τ_2 be types which do not unify and set $\alpha_i \equiv (\tau_i \to \tau_i)$, $i = 1, 2$. Since we can infer that I has each of the types α_1 and α_2, there exist weakly equivalent I_1 and I_2 with types α_1 and α_2. Now consider the unification problem $\langle x_1,\ x_2\rangle$ in the context $x_1 : \alpha_1, x_2 : \alpha_2$. This system has the solution $x_i := I_i, i = 1, 2$, but since the α_i do not unify it is difficult to see how this system could be transformed into a representation of *any* solution.

- Another line of inquiry involves higher-order unification in the presence of equations between algebraic terms. Adding a set E of equations to the axioms for $\beta\eta$-convertibility defines $\beta\eta E$-equality, and determines a corresponding notion of unification, *higher-order E-unification.* Breazu-Tannen showed [Bre88] that the

combination of algebra and typed λ-calculus is well-behaved (see also [BG89], [Dou91] [Bar90]); a complete set of transformations for higher-order E-unification has been defined by Snyder in [Sny90].

We have seen that by using combinators we can cast higher-order unification problems in the same mold as algebraic unification, and so this setting is a congenial one for the combined problem. In an obvious way one can define CE-equality and CE-unification, and observe that a solution to the CE-unification problem yields a solution to the higher-order E-unification problem.

In [Joh91] a method for CE-unification based on the techniques of this paper is developed for the case when E has a presentation as a confluent and terminating rewrite system.

5 Appendix

We first verify the claim in the Introduction that the translations between $\mathcal{LC}$ and $\mathcal{CL}$ are well-behaved with respect to substitutions.

Lemma 5.1 *1. For any $\mathcal{LC}$-term G, and substitution σ,*

$$\mathcal{H}(\sigma G) \equiv (\mathcal{H} \circ \sigma)\mathcal{H}(G).$$

2. For any $\mathcal{CL}$-term Y, and substitution θ,

$$\Lambda(\theta Y) \equiv (\Lambda \circ \theta)\Lambda(Y).$$

Proof.

1. The proof relies on the following sub-lemma, proved by an easy induction: For any type-substitution σ_0,
$$\mathcal{H}(\sigma_0 G) \equiv \sigma_0 \mathcal{H}(G).$$
Now, a classical fact about $\mathcal{H}$ is that for ordinary term-substitutions σ_1 and $\mathcal{LC}$-terms F,
$$\mathcal{H}(\sigma_1 F) \equiv (\mathcal{H} \circ \sigma_1)\mathcal{H}(F).$$
The result follows by setting F to be $\theta_0 G$.

2. The proof is similar to the above, using the facts that
$$\Lambda(\theta_0 Y) \equiv \theta_0 \Lambda(Y)$$
and (for any Z)
$$\mathcal{H}(\theta_1 Z) \equiv (\mathcal{H} \circ \theta_1)\mathcal{H}(Z).$$

□

Lemma 5.2 *1. A substitution σ is idempotent iff both σ_0 and σ_1 are idempotent.*

2. Suppose σ is idempotent. For any θ, $\sigma \leq \theta$ iff $\theta\sigma \equiv \theta$, and $\sigma \leq_c \theta$ iff $\theta\sigma =_c \theta$.

3. If $\sigma \leq \theta$ then $\sigma_0 \leq \theta_0$.

Proof.

1. This is a tedious but routine calculation.

2. For the non-trivial direction, first suppose that $\sigma \leq \theta$ and let η be such that $\eta\sigma \equiv \theta$ Then $\theta\sigma \equiv \eta\sigma\sigma \equiv \eta\sigma \equiv \theta$.

   If we suppose that $\sigma \leq_c \theta$ a similar calculation shows that $\theta\sigma =_c \theta$

3. Let η be such that $\eta\sigma \equiv \theta$, we show that $\eta_0\sigma_0 \equiv \theta_0$. Let t be any type-variable. Choose x of type t, and observe that $\eta\sigma x$ has type $\eta_0\sigma_0 t$ and that θx has type $\theta_0 t$. Since the terms are identical by hypothesis, so are their types.

□

It is not hard to construct an example in which $\sigma \leq \theta$, but $\sigma_1 \leq \theta_1$ fails.

A consequence of the first part of Lemma 5.2 is that when $[\sigma]$ is a solved system, σ is an idempotent substitution. This follows from the usual characterization of idempotent type- and term-substitutions as those whose domains are disjoint from the variables they introduce.

Lemma 5.3 *Let $[\sigma]$ be solved.*

1. If θ unifies $[\sigma]$ then $\theta\sigma \equiv \theta$. If θ C-unifies $[\sigma]$ then $\theta\sigma =_c \theta$.

2. If $\sigma \leq \theta$ then θ unifies $[\sigma]$. If $\sigma \leq_c \theta$ then θ C-unifies $[\sigma]$.

Proof.

1. Suppose θ unifies $[\sigma]$ (respectively, suppose that θ C-unifies $[\sigma]$). Choose any term- or type-variable v. We claim that $\theta\sigma v \equiv \theta\sigma_0 v$ (respectively, that $\theta\sigma v =_c \theta\sigma_0 v$). When v is a type-variable this follows from the Lemma 5.2, so suppose v is a term-variable. The claim is immediate if $\sigma_0 v \notin D(\sigma_1)$; but if $\sigma_0 v \in D(\sigma_1)$ we use the fact that θ unifies $[\sigma_1]$ (respectively, that θ C-unifies $[\sigma_1]$).

   So it suffices to show that under either hypothesis $\theta\sigma_0 v \equiv \theta v$. This follows from the fact that either hypothesis implies that θ_0 unifies the solved system corresponding to σ_0, so that $\theta_0\sigma_0 \equiv \theta_0$ and hence $\theta\sigma_0 \equiv \theta$.

2. The system $\theta[\sigma]$ is

$$\theta_0[\sigma_0],\ \theta[\sigma_1].$$

   By Lemma 5.2, $\sigma_0 \leq \theta_0$, and so $\theta_0[\sigma_0]$ is trivial. To show that $\theta[\sigma_1]$ is trivial, let $x \in D(\sigma_1)$; we need to see that $\theta x \equiv \theta\sigma_1 x$ in the first case, and that $\theta x =_c \theta\sigma_1 x$ in the second. Now, for any $x \in D(\sigma_1)$, $\sigma_0 x \equiv x$, since $[\sigma]$ is solved, so $\sigma x \equiv \sigma_1 x$. Therefore $\theta\sigma_1 x \equiv \theta\sigma x$, and the results follow from the facts that $\theta \equiv \theta\sigma$ and $\theta =_c \theta\sigma$.

□

It remains to see that the theory of syntactic unification proceeds smoothly in the presence of type-variables. In fact a simple variation on the standard transformations for syntactic unification of algebraic terms [MM82] leads to an algorithm for syntactic unification in our setting (we use a presentation inspired by [GS89a]).

In order to unify a system of terms, we may need to unifiy the types of the terms appearing there. If Σ is any system, say that the *derived system* of Σ is the system of type-pairs obtained by replacing each term by its type.

Definition 5.4 The set ST consists of the following transformations.

1. Decompose:

$$\Gamma,\ \langle (M_1 \cdots M_k),\ (N_1 \cdots N_k) \rangle \longrightarrow \Gamma,\ \langle M_1,\ N_1 \rangle, \ldots \langle M_k,\ N_k \rangle$$

2. Eliminate:

$$\Gamma,\ \langle x,\ A \rangle \longrightarrow \mu\Gamma,\ \langle x,\ A \rangle$$

   when x and A have the same type, and where μ is the substitution whose type-part is the identity and whose term-part is $\{x := A\}$.

3. Type-Unify:

$$\Sigma \longrightarrow [\mu],\ \mu\Sigma$$

   if the derived system of Σ is not already trivial, and where μ is the substitution whose type-part is the most general unifier of the derived system of Σ and whose term-part is the identity.

Remark 5.5 There is no deletion of trivial pairs in this presentation. This implies that no variables are lost when a system is transformed, which simplifies certain arguments (for example, when a fresh variable is chosen during a computation, that variable is guaranteed to be new to the entire computation. This ensures that if $\Sigma \Longrightarrow \Sigma'$ under VT or ST then $Supp(\Sigma) \subseteq Supp(\Sigma')$, and in Theorem 3.13, eliminates the manipulation of "protected" sets of variables typically found in completeness proofs in the literature.

Lemma 5.6 *1. If $\Sigma \longrightarrow \Sigma'$ by an ST transformation then Σ and Σ' have the same unifiers.*

2. Suppose Σ is ST-irreducible. Then Σ is unifiable iff Σ is solved.

3. Every sequence of ST-reductions terminates.

Proof.

1. We have cases according to the transformation performed; we use the notation of Definition 5.4. If the ST step in question is Decompose, the result is clear. For the case of Eliminate, if θ is a unifier of either the left- or right-hand sides then θ is a unifier of $[\mu]$. Then $\theta\mu \equiv \theta$, so that $\theta\Gamma$ and $\theta\mu\Gamma$ are identical. For the case of Type-Unify, first suppose that θ unifies the left-hand side Σ. Then θ_0 unifies the derived system of Σ, and so $\mu \leq \theta_0$, and $\theta[\mu]$ is trivial since $\theta_0[\mu]$ is. But then $\theta\mu \equiv \theta$, so that $\theta\mu\Sigma$ is trivial. Finally suppose that θ unifies the right-hand system $[\mu],\ \mu\Sigma$. Then $\mu \leq \theta$, so $\theta\mu \equiv \theta$, and $\theta\Sigma$ is trivial.

2. The fact that irreducible unsolved systems are not unifiable is clear, the converse follows from Lemma 5.3.

3. Define the following well-founded order on systems: compare the number of unsolved variables, then, if necessary, compare the sum of the sizes of the terms. It is easy to check that Decompose and Eliminate decrease this measure. But Type-Unify leaves this measure unchanged (this uses the fact that systems do not have distinct variables with the same type-erasure) and cannot be applied more than once consecutively.

□

Corollary 5.7 *Every unifiable system has a most general unifer.*

Proof. If Σ is syntactically unifiable then Σ reduces to a solved $[\sigma]$ by ST reductions. The substitution σ unifies Σ by part (1) of Lemma 5.6 and part (2) of Lemma 5.3 and is as general as other unifier by part (1) of Lemma 5.3; it is readily seen to be idempotent and to introduce no new variables or constants by the definition of ST. □

Acknowledgements.

Discussions with Wayne Snyder and with Frank Pfenning provided insight and encouragement and suggestions for improvement on an earlier draft. Ullrich Hustadt's observation about typing in the *Narrow* transformation was crucial.

References

[Bar90] F. Barbanera. Adding algebraic rewriting to the Calculus of Constructions: strong normalization preserved. *Extended Abstracts, Second Int'l Workshop on Conditional and Typed Rewriting Systems*, Center for Pattern Recognition and Machine Intelligence, Concordia University, Montreal, 1990.

[BG89] V. Breazu-Tannen and J. Gallier. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *Proc.* 16^{th} *ICALP*, Springer-Verlag, 1989. Also, to appear in *Theoretical Computer Science.*

[Bre88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. Third Annual Symp. on Logic in Computer Science*, IEEE, pp. 82-90, June 1988.

[CF58] H. B. Curry, R. Feys. *Combinatory Logic, Vol. I*, North-Holland, 1958.

[CHS72] H. B. Curry, J. R. Hindley, J. P. Seldin. *Combinatory Logic, Vol. II*, North-Holland, 1972.

[Dou91] D. J. Dougherty. Adding algebra to the untyped lambda calculus. In *Proc. Fourth Int'l. Conf. on Rewriting Techniques and Applications*, Springer-Verlag LNCS 488, pp. 37-48, 1991. To appear, *Information and Computation.*

[DJ90] D. J. Dougherty and P. Johann. An improved general E-unification method. In *Proc. Tenth Conf. on Automated Deduction*, Springer-Verlag Lec. Notes in AI 449, pp. 261-75, 1990. To appear, *Journal of Symbolic Computation.*

[Ell89] C. Elliott. Higher-order unification with dependent function types/ In *Proc. Third Int'l. Conf. on Rewriting Techniques and Applications*, Springer-Verlag Lec. Notes in CS 355, pp. 121–136, 1989.

[Fay79] M. Fay. First order unification in an equational theory. *Proc. Fourth Workshop on Automated Deduction*, Austin, Texas, 1979.

[Gol81] D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science* 13, pp. 225-230, 1981.

[GS89a] J. H. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science* 67:2,3, pp. 203-260, 1989.

[GS89b] J. H. Gallier and W. Snyder. Higher-order unification revisited: complete sets of transformations. *Journal of Symbolic Computation* 8:1&2, pp. 101-140, 1989.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. of the American Math. Soc.* 146, December 1969, pp. 29-60.

[HL70] R. Hindley, B. Lercher. A short proof of Curry's normal form theorem. *Proc. American Math. Soc.* 24, pp. 808-10.

[Hue73] G. Huet. The undecidability of unification in third-order logic. *Information and Control* 22, pp. 257-267, 1973.

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

[Hue76] G. Huet. *Résolution d'equations dans les langages d'ordre* $1, 2, \ldots, \omega$. Thèse d'Etat, Université de Paris VII, 1976.

[Hug82] R. J. M. Hughes. Super-Combinators. In *Proc. ACM Conference on LISP and Functional Programming*, 1982.

[Hug84] R. J. M. Hughes. The design and implementation of programming laguages. Dissertation, Programming Research Group, Oxford University, 1984.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*, Cambridge University Press. 1986.

[Joh91] P. Johann. *Complete Sets of Transformations for Unification Problems.* Dissertation, Wesleyan University, 1991.

[JP73] D. Jensen and T. Pietrzykowski. Mechanizing ω-order type theory through unification. Report CS-73-16, Dept. of Applied Analysis and Computer Science, University of Waterloo, 1973.

[Ler67] B. Lercher. Strong reduction and normal form in combinatory logic. *The Journal of Symbolic Logic*, 2 (1967), pp. 213-23.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. on Prog. Languages and Systems 4:2*, pp. 258-282, 1982.

[Nip90] T. Nipkow. Higher-order unification, polymorphism, and subsorts. *Extended Abstracts, Second Int'l. Workshop on Conditional and Typed Rewriting Systems*, Center for Pattern Recognition and Machine Intelligence, Concordia University, Montreal, 1990.

[Nad87] G. Nadathur. A higher-order logic as the basis for logic programming. Dissertation, University of Pennsylvania, 1987.

[Pau90] L. C. Paulson. Isabelle: The Next 700 Theorem Provers, in P. Odifreddi (ed.), *Logic and Computer Science*, Academic Press, 1990.

[Pey87] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Pie71] T. Pietrzykowski. A complete mechanization of second order logic. *ACM Journal* 20:2, pp. 333-364, 1971.

[Ret87] P. Réty. Improving basic narrowing techniques. In *Proc. Second Int'l. Conf. on Rewriting Techniques and Applications*, 1987.

[Sch86] M. Scheevel. Norma: a graph reduction processor. In *Proc. ACM Conference on LISP and Functional Programming*, pp. 212-219, 1986.

[Sny90] W. Snyder. Higher-order E-unification. In *Proc. Tenth Conference on Automated Deduction*, Springer-Verlag Lec. Notes in AI 449, 573–587, 1990.

[Sta86] R. Statman. On translating lambda terms into combinators: the basis problem. *Proc. Symp. on Logic in Computer Science*, pp. 378-382, 1986.

[Sto85] W. R. Stoye. The implementation of functional languages using custom hardware. Dissertation, Computer Lab, University of Cambridge, 1985.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience* 9, pp. 31-49, 1979.