UNIVERSITÄT DES SAARLANDES
FACHBEREICH INFORMATIK
Im Stadtwald
W-6600 Saarbrücken 11
Germany

SEKI-REPORT

Unification in an Extensional Lambda Calculus with Ordered Function Sorts and Constant Overloading

Patricia Johann, Michael Kohlhase

SEKI Report SR-93-14

Unification in an Extensional Lambda Calculus with Ordered Function Sorts and Constant Overloading

Patricia Johann* and Michael Kohlhase†
Fachbereich Informatik
Universität des Saarlandes
66123 Saarbrücken, Germany
{*pjohann, kohlhase*}*@cs.uni-sb.de*

Abstract

The introduction of sorts in first-order automatic theorem proving has been accompanied by a considerable gain in computational efficiency via reduced search spaces. This suggests that sort information can be employed in higher-order theorem proving with similar results. This paper develops an order-sorted higher-order calculus suitable for automatic theorem proving applications — by extending the extensional simply typed lambda calculus with a higher-order ordered sort concept and constant overloading — and extends Huet's well-known techniques for unification in the simply typed lambda calculus to arrive at a complete transformation-based unification algorithm for this sorted calculus. Consideration of an order-sorted logic with functional base sorts and arbitrary term declarations was originally proposed by the second author in a 1991 paper; we give here a corrected calculus which supports constant, rather than arbitrary term, declarations, as well as a corrected unification algorithm, and prove in this setting results corresponding to those claimed in that earlier work.

1 Introduction

In the quest for calculi best suited for automating logic, the introduction of sort information has been one of the most promising developments. Sorts, which are intended to capture for automated deduction purposes the kinds of meta-level taxonomic distinctions that humans naturally assume structure the universe, can be employed to syntactically distinguish objects of different classes. The essential idea behind sorted logics is to assign sorts to objects and to restrict the ranges of variables to particular sorts, so that unintended inferences, which then violate the constraints imposed by this sort information, are disallowed. These techniques have been seen to dramatically reduce the search space associated with deduction in first-order systems, so that the resulting sorted calculi are significantly more efficient for such purposes than their unsorted counterparts.

*On leave from the Department of Mathematics and Computer Science, Hobart and William Smith Colleges, Geneva, NY 14456. This material is based upon work supported by the National Science Foundation under Grant No. INT-9224443. It was also supported in part by a grant from the Deutscher Akademischer Austauschdienst and in part by the Deutsche Forschungsgemeinschaft (SFB 314).

†Supported by the Deutsche Forschungsgemeinschaft (SFB 314).

In the context of first-order logic, sort information has been employed with impressive results by Oberschelp ([Obe62]), Walther ([Wal88]), Cohn([Coh89]), Schmidt-Schauß ([Sch89]), and others.

Despite the existence of powerful first-order deduction systems (see, *e.g.*, [OS89], [Lus92]) the inherently higher-order nature of many problems whose solutions one would like to deduce automatically has sparked an increasing interest in higher-order deduction ([ALMP84], [Gor85], [Pau90], [Mil91]). Certainly any system intended for automating real mathematics must concern itself with higher-order logic, as suggested by van Dalen's observation that "analysis is just another word for second-order arithmetic" ([van91]). The behavior of sorted higher-order calculi, which boast both the expressiveness of typed higher-order logic and the efficiency of sorted calculi, is thus a natural topic of investigation — such calculi can be expected to serve as a basis for the development of ever more powerful deduction systems. This paper proposes an extensional order-sorted lambda calculus supporting functional base sorts and constant overloading, and develops for it a complete unification algorithm suitable for use in an automated deduction setting. Although Huet proposed the study of a simple sorted lambda calculus in an appendix to [Hue72], the development of order-sorted higher-order calculi for use in deduction systems has only in recent years been pursued ([Koh92], [NQ92], [Pfe92], [KP93]). There has, however, been considerable interest in order-sorted higher-order logic from the point of view of higher-order algebraic specifications, the theory of functional programming languages, and object-oriented programming ([Car88], [BL90], [Qia90], [CG91], [FP91], [Mit91], [Pie91], [Qia91]).

In unsorted logics, the knowledge that an object is a member of a certain class of objects is expressed using unary predicates. If ι denotes the class of individuals, and o denotes the class of truth values, then, for example, the predicate $N_{\iota\to o}$ in the formula $N2_\iota$, is intended to specify that the individual 2 is a natural number, while in $\neg(N\text{Peter}_\iota)$ it indicates that the individual Peter is not a natural number. This use of predicates leads to a multitude of unit clauses of the form PX in deductions, each of which carries only taxonomic information and contributes to a severe explosion of the deduction search space. In addition, since quantification is unrestricted in such calculi, restricted quantification must be simulated by (typed) formulae like $\forall x_\iota[(Nx_\iota) \Rightarrow (\geq_{\iota\to\iota\to o} x_\iota 0_\iota)]$. But this is certainly unsatisfactory since, *inter alia*, the derivation of nonsensical formulae such as $(N\text{Peter}) \Rightarrow (\geq \text{Peter } 0)$ is permitted even though $(\geq \text{Peter } 0)$ can never be derived if $\neg(N\text{Peter})$ holds (as we expect it would). In sorted logics, on the other hand, the typed predicate $N_{\iota\to o}$ would be replaced by a sort N carrying precisely the same taxonomic information. This eliminates the need for such predicates and unit clauses, and also makes restricted quantification possible. For example, in a sorted logic, the last formula above would read $\forall x_N(\geq_{N\to N\to o} x_N 0_N)$.

Since type information can be regarded as coding very coarse taxonomic distinctions between disjoint classes of objects, the introduction of sorts in the higher-order setting is perhaps even more natural than in first-order logic: in higher-order logics, sort information merely refines an already present structure. Moreover, the fact that humans use class information to structure the universe, and that mathematicians naturally use variables and functions restricted to these classes, can be captured by sorted logics, which are therefore closer to the models mathematicians make of the world than are typed logics. In particular, sorted higher-order logics seem to be considerably more adequate for formalizing analysis, since they support syntactic constructs for representing domains

and codomains of functions, as well as function restriction (as discussed below).

These ideas provide a point of departure for modifying the syntax of the simply typed lambda calculus to incorporate taxonomic information. Sorting the universe of individuals gives rise to new classes of functions, namely functions whose domains and codomains are just (denoted by) the sorts. But in addition to sorting function universes in this essentially first-order manner, classes of functions defined by domains and codomains can themselves be further divided into subclasses, since functions are explicit objects of higher-order logic. Base sorts of functional type, *i.e.,* base sorts that denote classes of functions, are thus introduced. Syntactically, each sort A comes with a type, a codomain sort $\gamma(A)$, and — if of functional type — also with a domain sort $\delta(A)$. But in the presence of functional base sorts, an additional mechanism for inducing subsort information from base sorts is needed: since any function of sort A is indeed a function with domain $\delta(A)$ and codomain $\gamma(A)$, a functional sort A must always be a subsort of the sort $\delta(A) \rightarrow \gamma(A)$.

In the calculus presented here, sort information restricting the ranges of variables to, and assigning constants membership in, certain classes of objects can be declared. Depending on what kind of relationships hold among the various sorts, certain classes of terms built from these atoms then become the objects of study — in practice, it is most natural to require the set of sorts to be partially ordered. Imposing a partial ordering on sorts necessitates specifying a set of subsort declarations which induce the intended partial ordering by covariance in the codomain sort. Subsort declarations restrict the class of models for the calculus, so that terms must meet certain sort conditions to denote meaningful objects, *i.e.,* to be well-sorted. For example, application of the functional term X to the argument Y is allowed only if there exist sorts A and B such that X is of sort A, Y is of sort B, and B is a subsort of $\delta(A)$. The sort of the application term XY is then defined to be $\gamma(A)$.

It is possible to express the concept of function restriction in such a sorted calculus. If X is a term of functional sort A, and B is a subsort of the domain sort of A, then the term $\lambda x_B.Xx$ denotes the restriction of the function (denoted by) X to the domain (specified by) B. Note, however, that in order to properly model extensionality by η-reduction, B must be precisely the maximal domain of X for $\lambda x.Xx$ to η-reduce to X — otherwise X would be equal to a proper restriction of X, which cannot possibly be the same function. In the calculus given here, restrictions are imposed on subsort and constant declarations to ensure that all well-sorted terms representing functions have unique maximal domains.

Currently, sorted deduction systems make only limited use of sorts, employing them mainly to determine which terms may — as governed by sort constraints — be substituted for variables in deduction steps. Accordingly, sort information is manipulated by sorted deduction systems primarily through the use of (pre-)unification algorithms rich enough to accommodate the relevant sorted calculi. Unification in an extensional order-sorted higher-order calculus with functional base sorts was first investigated in [Koh92]. In that work, the second author additionally suggests allowing the sorts of arbitrary terms, rather than just of constants, to be declared, a move which would result in quite an expressive calculus. Unfortunately, the work presented in [Koh92] is flawed in several places and does not adequately address higher-order unification for such a calculus (see the discussion in Section 4 below).

Since our calculus allows only constant, rather than arbitrary term, declarations, and

therefore is somewhat less expressive than that considered in [Koh92], it can be seen as a subcalculus of that one, which has been corrected to properly incorporate extensionality in the presence of partially ordered sorts (in fact, we expect the methods developed here for treating this delicate interaction to be applicable in other calculi). As is proved here, both the subsort relation and sort assignment are decidable for our revised calculus. This is more than just a nice feature of the calculus — for certain of the transformations on which the unification algorithm given here is based, enumerability, at least, of sort assignment is necessary to determine that the transformation indeed applies to a given unification problem.

The transformations, as well as the unification algorithm itself, are given in Section 3, where its soundness and completeness are also proved. The results of this section were obtained by analyzing the impact of properly handling extensionality in the presence of functional base sorts on incrementally building up answer substitutions. An appropriate notion of partial binding is given in Section 3.2, generalizing the Huet-style bindings sufficient for analysis of the simply typed lambda calculus, and it is shown in Section 3.3 how these partial bindings can be used to approximate answer substitutions to unification problems in the calculus under study. This work remedies both the ill-defined unification transformations and the flawed completeness proof given in [Koh92]. For more precise details, the reader is again referred to Section 4 of this paper.

2 Preliminaries

We begin by recalling the basic results concerning the simply typed lambda calculus, and describing the order-sorted higher-order calculus with which we will be concerned. For a detailed discussion of the simply typed lambda calculus, the reader is referred to [HS86] and [Bar84], both excellent sources.

Definition 2.1 The set of *types* $\mathcal{T}$ is obtained by inductively closing a set of *base types* $\mathcal{T}_0$ under function construction, *i.e.*, under the operation $\alpha \to \beta$. The *length* of a type α, denoted $length(\alpha)$, is the number of top-level arrows appearing in it.

Types will be denoted by lower case Greek letters. In theorem proving applications we might have only two base types, o denoting truth-values, and ι denoting the universe of individuals. All other subdivisions of the universe would then be coded into sort distinctions among individuals, as described in the next subsection.

For each type $\alpha \in \mathcal{T}$, fix a countably infinite set of variables of type α and a countably infinite set of constants of type α. Write $x_\alpha, y_\alpha, z_\alpha, \ldots$ for variables of type α and $a_\alpha, b_\alpha, c_\alpha, \ldots$ for constants of type α. The variables and constants for the various $\alpha \in \mathcal{T}$ are collectively called *atoms*; we assume that no two distinct atoms have the same type-erasure.

$\mathcal{LC}$ is the set of explicitly simply typed lambda terms over the atoms. We will write X_α to indicate that X is an $\mathcal{LC}$-term of type α, and omit the type of X when this will not lead to confusion.

On $\mathcal{LC}$, $\beta\eta$-equality is generated by $\beta\eta$-reduction, determined by the rules $(\lambda x.X)Y \xrightarrow{\beta} X[x := Y]$ and $\lambda x.Xx \xrightarrow{\eta} X$. We assume that β-reduction occurs without free variable capture, and that x is not free in X for the η-reduction rule. $\beta\eta$-reduction is terminating and confluent (*i.e., convergent*) on $\mathcal{LC}$-terms, so we can speak of *the* β-

normal form and *the $\beta\eta$-normal form* of an $\mathcal{LC}$-term X. As usual, we denote β-reduction on $\mathcal{LC}$-terms by $\xrightarrow{\beta}$, η-reduction by $\xrightarrow{\eta}$, and $\beta\eta$-reduction by $\xrightarrow{\beta\eta}$.

The reflexive, transitive closure of a reduction relation $\xrightarrow{\nu}$ is denoted $\xrightarrow{\nu}\!\!\!\!\rightarrow$, and we will write $=_\nu$ for the symmetric closure of $\xrightarrow{\nu}\!\!\!\!\rightarrow$, *i.e.,* for the equivalence relation generated by $\xrightarrow{\nu}$. We write $X \equiv Y$ to indicate that two $\mathcal{LC}$-terms X and Y are identical up to renaming of bound variables. As is customary, we will consider $\mathcal{LC}$-terms identical up to renaming of bound variables to be the same.

2.1 Order-sorted Structures

In mathematics, subdividing the universe of individuals gives rise to new classes of functions, namely those whose domains and codomains are among the various subdivisions. But in addition to this essentially first-order way of partitioning function universes, the classes of functions defined by domains and codomains can be further divided into subclasses, since functions are explicit objects of higher-order logic. To reflect this richer structuring of higher-order objects, we introduce into our calculus base sorts of functional type, *i.e.,* base sorts that denote classes of functions, as well as non-functional base sorts. Syntactically, each sort comes with a type, a codomain sort, and — if it is of functional type — also with a domain sort.

Definition 2.2 A *sort system* is a quintuple $(\mathcal{S}_0, \mathcal{S}, \tau, \delta, \gamma)$ such that:

- $\mathcal{S}_0$ is a set of *base sorts* distinct from the set of type symbols. The set of *sorts* obtained by closing $\mathcal{S}_0$ under function construction comprises $\mathcal{S}$.
- The *type function* τ is a mapping $\tau : \mathcal{S}_0 \to \mathcal{T}$. If $\tau(A) \in \mathcal{T}_0$, then A is said to be *non-functional*; A is said to be *functional* otherwise. The set of non-functional (resp., functional) sorts is denoted by $\mathcal{S}^{nf}$ (resp., $\mathcal{S}^f$). For all $A \in \mathcal{S}^f$, we require that $\tau(A) = \tau(\delta(A)) \to \tau(\gamma(A))$, where
  - the *domain sort function* δ is a map $\delta : \mathcal{S}_0^f \to \mathcal{S}$,
  - the *codomain sort function* γ is a map $\gamma : \mathcal{S}_0 \to \mathcal{S}$ with $\gamma \mid_{\mathcal{S}^{nf}}$ the identity map, and
  - the mappings δ and γ are extended to $\mathcal{S}$ by defining $\delta(A) = B$ and $\gamma(A) = C$ for $A \equiv B \to C \in \mathcal{S}$.

Sorts will be denoted by upper case Roman letters from the beginning of the alphabet. If the context is clear, we will abbreviate by $\mathcal{S}$ the sort system $(\mathcal{S}_0, \mathcal{S}, \tau, \delta, \gamma)$. We may suppress references to $\mathcal{S}$ entirely when no confusion will arise. Since we are ultimately interested in sorted terms and their typed counterparts, we will only consider sort systems for which τ is surjective. We will further assume that for each $\alpha \in \mathcal{T}$ there exist only finitely many $A \in \mathcal{S}_0$ such that $\tau(A) = \alpha$, *i.e.,* that sort systems have *finitely many base sorts per type*. The type $\tau(A)$ is called the *type* of the sort A.

It will be useful to have some notational conventions for domain and codomain sorts. For any $A \in \mathcal{S}$, recursively define the following notation: $\delta^0(A) \equiv A$, $\gamma^0(A) \equiv A$, and for $i \geq 1$, $\gamma^i(A) \equiv \gamma(\gamma^{i-1}(A))$, and $\delta^i(A) \equiv \delta(\gamma^{i-1}(A))$. Write $\mathit{length}(A)$ for the *length* of, *i.e.,* the number of top-level arrows in, the sort A.

Example 2.3 Functional base sorts are useful, for example, in the study of elementary analysis, where one can postulate a non-functional base sort R denoting the reals, and a functional base sort C such that $\delta(C) = R$ and $\gamma(C) = R$ denoting the class of continuous real-valued functions on the reals. It is worth noting that it is not possible to syntactically distinguish the continuous real-valued functions on reals solely in terms of their domains and codomains, so that functional base sorts indeed increase the expressiveness of a calculus.

While types represent disjoint classes of objects, certain kinds of orderings on sorts reflect permissible inclusion relations among classes of objects denoted by sorts. The next definition captures a consistency condition we require such orderings to satisfy.

Definition 2.4 Given a sort system $\mathcal{S}$, for each pair of sorts A and B in $\mathcal{S}$ such that $\tau(A) = \tau(B)$, the set $\mathrm{Con}(A, B)$ of *subsort declarations* (for $\mathcal{S}$) is defined to be the set $\{[A \leq B]\}$ if $A, B \in \mathcal{S}^{nf}$, and

$$\mathrm{Con}(\delta(A), \delta(B)) \cup \mathrm{Con}(\delta(B), \delta(A)) \cup \mathrm{Con}(\gamma(A), \gamma(B)) \cup \{[A \leq B]\}$$

if $A, B \in \mathcal{S}^{f}$.

Definition 2.5 Given a sort system $\mathcal{S}$, a *sort structure* (for $\mathcal{S}$) is any set Δ of subsort declarations such that the judgement $\vdash_{ss} \Delta$ is provable by the following calculus:

$$\frac{}{\vdash_{ss} \emptyset} \quad (ss - start)$$

$$\frac{\vdash_{ss} \Delta \qquad \tau(A) = \tau(B)}{\vdash_{ss} \Delta \cup \mathrm{Con}\,(A, B)} \quad (ss - ext)$$

The judgement $\vdash_{ss} \Delta$ is precisely the declaration that Δ is a sort structure. The rule (ss-start) guarantees that the empty set is a sort structure, and (ss-ext) indicates that sort structures may be built inductively by adding to an existing sort structure a set of subsort declarations of the form $\mathrm{Con}(A, B)$. Of course $\tau(A) = \tau(B)$ is entailed in the assumption that $\mathrm{Con}(A, B)$ is defined; nevertheless, we specifically state it as a hypothesis in (ss-ext). Note that since sort structures have finite derivations and since $\mathrm{Con}(A, B)$ is always finite, sort structures are themselves always finite.

The following two useful lemmas are easy to prove.

Lemma 2.6 *Let Δ be a sort structure. If $[A \leq B] \in \Delta$, then $\tau(A) = \tau(B)$.*

Proof. If $[A \leq B] \in \Delta$, then $[A \leq B] \in \mathrm{Con}(C, D)$ for some sorts C and D such that $\tau(C) = \tau(D)$ and $\mathrm{Con}(C, D) \subseteq \Delta$. Induction on the *length*$(\tau(C))$, as in the proof of the next, slightly-more-difficult lemma, yields the conclusion. □

Lemma 2.7 *For a sort structure Δ, $[A \leq B] \in \Delta$ iff $\mathrm{Con}(A, B) \subseteq \Delta$.*

Proof. Necessity follows from Definition 2.4. To prove sufficiency, note that if $[A \leq B] \in \Delta$, then $[A \leq B] \in \mathrm{Con}(C, D)$ for some sorts C and D such that $\tau(C) = \tau(D)$ and $\mathrm{Con}(C, D) \subseteq \Delta$. In particular, then, $[C \leq D] \in \Delta$. We proceed by induction on the length of $\alpha = \tau(C) = \tau(D)$.

- If $length(\alpha) = 1$, then $C, D \in \mathcal{S}^{nf}$, so $[A \leq B] \in \mathrm{Con}(C, D)$ implies $A \equiv C$ and $B \equiv D$, and there is nothing to prove.
- If $length(\alpha) > 1$, then $C, D \in \mathcal{S}^{f}$, and so $\mathrm{Con}(C, D)$ is exactly

$$\mathrm{Con}(\delta(A), \delta(B)) \cup \mathrm{Con}(\delta(B), \delta(A)) \cup \mathrm{Con}(\gamma(A), \gamma(B)) \cup \{[A \leq B]\}.$$

  Again, if $[A \leq B] \equiv [C \leq D]$, then there is nothing to prove. If $[A \leq B] \in \mathrm{Con}(\delta(C), \delta(D))$, then since $\tau(\delta(C)) = \tau(\delta(D))$, and since $length(\tau(\delta(C)))$ is less than $length(\alpha)$, the induction hypothesis guarantees that $\mathrm{Con}(A, B) \subseteq \Delta$ as desired. The result is proved similarly if $[A \leq B] \in \mathrm{Con}(\delta(D), \delta(C))$ or $[A \leq B] \in \mathrm{Con}(\gamma(C), \gamma(D))$.

□

Any sort structure Δ induces an *inclusion ordering* $\leq$ on $\mathcal{S}$, inductively defined by the rules of Definition 2.8. The rule ($\leq$-start) indicates that the inclusion ordering is indeed determined by Δ, ($\leq$-incl) reflects the natural inclusion of function spaces, ($\leq$-cov) insures covariance in the codomain sort, and ($\leq$-refl) and ($\leq$-trans) require that the inclusion ordering determined by Δ be a quasi-ordering. For this ordering, we will write $\leq_\Delta$, or just $\leq$ as above if Δ is clear from the context.

Definition 2.8 For any sort structure Δ, the *inclusion ordering determined by* Δ contains all judgements of the form $\Delta \vdash A \leq B$ which are provable by the following calculus:

$$\frac{[A \leq B] \in \Delta}{\Delta \vdash A \leq B} \quad (\leq\text{-}start)$$

$$\frac{A \in \mathcal{S}^{f} \qquad \vdash_{ss} \Delta}{\Delta \vdash A \leq \delta(A) \to \gamma(A)} \quad (\leq\text{-}incl)$$

$$\frac{\Delta \vdash A \leq B}{\Delta \vdash C \to A \leq C \to B} \quad (\leq\text{-}cov)$$

$$\frac{\vdash_{ss} \Delta}{\Delta \vdash A \leq A} \quad (\leq\text{-}refl)$$

$$\frac{\Delta \vdash A \leq B \qquad \Delta \vdash B \leq C}{\Delta \vdash A \leq C} \quad (\leq\text{-}trans)$$

We will not, however, insist that $\Delta \vdash A \leq B$ hold for any sorts A and B with a common domain sort C and whose codomain sorts satisfy $\Delta \vdash \gamma(A) \leq \gamma(B)$. Letting A denote the class of surjective functions from C to $\gamma(A)$ and B denote the set of surjective functions from C to $\gamma(B)$ demonstrates the undesirability of such a constraint (assuming a standard semantics).

Of course, any judgement $\Delta \vdash A \leq B$ has a finite derivation, and we may induct over such derivations.

Lemma 2.9 *Let Δ be a sort structure for $\mathcal{S}$, and $A, B \in \mathcal{S}^f$. If $\Delta \vdash A \leq B$ then $\Delta \vdash \delta(A) \sim \delta(B)$ and $\Delta \vdash \gamma(A) \leq \gamma(B)$.*

Proof. By induction on the derivation of $\Delta \vdash A \leq B$.

- If $\Delta \vdash A \leq B$ is the conclusion of ($\leq$-start), then $\mathrm{Con}(A, B) \subseteq \Delta$ by Lemma 2.7 so that, in particular, $[\delta(A) \leq \delta(B)] \in \Delta$, $[\delta(B) \leq \delta(A)] \in \Delta$, and $[\gamma(A) \leq \gamma(B)] \in \Delta$. Together these imply that $\Delta \vdash \delta(A) \sim \delta(B)$ and $\Delta \vdash \gamma(A) \leq \gamma(B)$.
- If $\Delta \vdash A \leq B$ is the conclusion of ($\leq$-incl), then there is nothing to prove.
- If $\Delta \vdash A \leq B$ is the conclusion of ($\leq$-cov), then the conclusion of the lemma is precisely the hypothesis of the inference rule.
- If $\Delta \vdash A \leq B$ is the conclusion of ($\leq$-refl), then again there is nothing to prove.
- If $\Delta \vdash A \leq B$ is the conclusion of ($\leq$-trans), then the result follows from the induction hypothesis.

□

As remarked after Definition 2.5, any sort structure Δ contains only finitely many subsort declarations $[A \leq B]$, and this latter fact, together with the assumption that sort systems have only finitely many base sorts per type, implies the decidability of the inclusion ordering $\leq$ determined by Δ. Although only semi-decidability of $\leq$ will be necessary for ensuring computability of sort assignment (see Lemma 2.22 below) — and thus determining applicability of our algorithm — in fact decidability is not hard to prove. The proof uses the next two results.

Lemma 2.10 *For a sort structure Δ, if $\Delta \vdash A \leq B$, then $\tau(A) = \tau(B)$.*

Proof. The proof is a routine induction on the derivation of $\Delta \vdash A \leq B$, using Lemma 2.6. □

Corollary 2.11 *A sort system $\mathcal{S}$ is the disjoint union of infinitely many subsets $\mathcal{S}_\alpha = \{A \in \mathcal{S} \mid \tau(A) = \alpha\}$ of sorts which are mutually incomparable. That is, if $A \in \mathcal{S}_\alpha$ and $B \in \mathcal{S}_\beta$ with $\alpha \not\equiv \beta$, then A and B are incomparable with respect to $\leq$. Moreover, since $\mathcal{S}$ has only finitely many base sorts per type, the subsets $\mathcal{S}_\alpha$ are finite,* i.e., *any sort system $\mathcal{S}$ has finitely many sorts per type.*

Theorem 2.12 *For any type $\alpha \in \mathcal{T}$ and any sort structure Δ, if $\leq$ is the inclusion ordering determined by Δ, then the restriction $\leq_\alpha$ of $\leq$ to sorts of type α is effectively computable.*

Proof. By induction on the $length(\alpha)$.

- If $length(\alpha) = 1$, then the set of subsort declarations in Δ restricted to type α, together with the conclusions of the rule ($\leq$-refl) are closed under ($\leq$-start) and ($\leq$-refl). The relation we are interested in is therefore just the transitive closure of this finite set, and the conclusion of the lemma follows from the well-known fact that the transitive closure of a finite relation is effectively computable.

- If $length(\alpha) > 1$, then the relation we are interested in is the closure under ($\leq$-cov) and ($\leq$-trans) of the finite set containing all conclusions of ($\leq$-start) restricted to sorts of type α, ($\leq$-incl), and ($\leq$-refl). This closure can be computed by alternately iterating closures under ($\leq$-trans) and ($\leq$-cov) until saturation. Since sort systems have only finitely sorts per type, Corollary 2.11 implies that $\leq_\alpha$ must be finite, and therefore that saturation is indeed achieved after only finitely many iterations. As above, closures under ($\leq$-trans) are computable, and the induction hypothesis implies computability of closures under ($\leq$-cov).

□

Corollary 2.13 *For any sort structure Δ, the inclusion ordering determined by Δ is decidable.*

Proof. Let $\leq$ denote the ordering determined by Δ, and suppose sorts A and B are given. We want to decide whether or not $\Delta \vdash A \leq B$.

If $\tau(A) \neq \tau(B)$, then $\Delta \vdash A \leq B$ is not possible by Lemma 2.10. Otherwise, for $\alpha = \tau(A) = \tau(B)$, we can effectively compute $\leq_\alpha$ and then check whether or not A and B are in this relation by inspection. □

To define the signatures over which our well-sorted terms will be built, we require a final preliminary notion. It will turn out to be important that signatures "respect function domains," in the sense that for any term X and any sorts A and B such that X has sort A and also sort B, $\delta(A) \sim \delta(B)$ holds. The proof that signatures indeed satisfy this property (see Lemma 2.24) will depend in part on the consistency conditions embodied in Definition 2.4 and in part on the fact that constant declarations meet the sort condition of the fifth clause of Definition 2.16 below, given in terms of the relation Rdom, which we now define.

Definition 2.14 Given a sort structure Δ, the binary relation Rdom_Δ is defined by

$$\frac{A, B \in \mathcal{S}^{nf} \qquad \tau(A) = \tau(B)}{\mathrm{Rdom}_\Delta(A, B)}$$

$$\frac{A, B \in \mathcal{S}^{f} \quad \Delta \vdash \delta(A) \sim \delta(B) \quad \mathrm{Rdom}_\Delta(\gamma(A), \gamma(B))}{\mathrm{Rdom}_\Delta(A, B)}$$

We will write Rdom for Rdom_Δ when Δ can be discerned from the context, and A Rdom B in place of $\mathrm{Rdom}(A, B)$.

We collect some easy but important facts about the relation Rdom.

Lemma 2.15 *For any sort structure Δ, the following statements hold:*

1. *If A Rdom B, then $\tau(A) = \tau(B)$.*
2. *Rdom is an equivalence relation.*
3. *If $\Delta \vdash A \leq B$, then A Rdom B.*

Proof. (1) is proved by induction on the derivation of A Rdom B, using Lemma 2.10. (2) is proved by induction on the length of the types of the terms involved in showing reflexivity and transitivity, and by observing that the definition of Rdom is symmetric. To prove (3), induction in a manner similar to that used in the proof of Lemma 2.7 is employed together with Lemma 2.6 to first show that if $[A \leq B] \in \Delta$, then A Rdom B. The more general result is obtained by induction on the derivation of $\Delta \vdash A \leq B$ using this fact and (2). □

We are at last in a position to describe the signatures which are suitable for our purposes.

Definition 2.16 A *(sorted) signature* Σ comprises

- a sort system $\mathcal{S} = (\mathcal{S}_0, \mathcal{S}, \delta, \gamma, \tau)$,
- a sort structure Δ (for $\mathcal{S}$),
- a countably infinite set $Vars_A$ of *(sorted) variables* $x_A, y_A, z_A, \ldots$ for each $A \in \mathcal{S}$,
- a set $\mathcal{C}$ of typed (but unsorted) constant symbols, and
- a set of *constant declarations* of the form $[c_\alpha :: A]$ for $c \in \mathcal{C}$ such that $\tau(A) = \alpha$. We assume that if $[c :: A]$ and $[c :: B]$ are constant declarations, then A Rdom B.

In a theorem proving context, any signature would have, for each $\alpha \in \mathcal{T}$, only finitely many constant declarations involving constants of type α, *i.e., only finitely many constant declarations per type.* We will therefore assume this restriction on signatures in what follows.

Any sorted variable can be regarded as a typed variable in a natural way by "forgetting" its sort information and retaining only its type information. If we denote the forgetful functor by $\overline{\ }$, then we may regard the sorted variable x_A as the typed variable $\overline{x_A}$, *i.e.,* as $x_{\tau(A)}$. By prudently naming the variables, we may arrange that the forgetful functor is bijective on variables, thereby avoiding the merely technical complications that could otherwise arise.

The requirement that $\tau(A) = \alpha$ for a constant declaration $[c_\alpha :: A]$ insures that sort assignments respect the types of constants. According to Definition 2.16, signatures permit constant overloading of a restricted nature, consistent with this requirement.

2.2 Term Structure

We now define and explore properties of the calculus with which we will be concerned.

Definition 2.17 The set of *well-sorted $\mathcal{LC}$-terms* (for Σ) is determined inductively by the following inference rules:

$$\frac{x \in Vars_A}{\Sigma \vdash x : A} \quad (var)$$

$$\frac{[c :: A] \in \Sigma}{\Sigma \vdash c : A} \quad (const)$$

$$\frac{\Sigma \vdash X : A \quad \Sigma \vdash Y : B \quad \Delta \vdash B \sim \delta(A)}{\Sigma \vdash XY : \gamma(A)} \quad (app)$$

$$\frac{x \in Vars_B \qquad \Sigma \vdash X : A}{\Sigma \vdash \lambda x.X : B \to A} \quad (abs)$$

$$\frac{\Sigma \vdash X : A \qquad \Delta \vdash \delta(A) \sim B}{\Sigma \vdash \lambda x_B.Xx : A} \quad (\eta)$$

$$\frac{\Sigma \vdash X : B \qquad \Delta \vdash B \leq A}{\Sigma \vdash X : A} \quad (weaken)$$

Let $\mathcal{LC}_A(\Sigma) = \{X \mid \Sigma \vdash X : A\}$ and $\mathcal{LC}(\Sigma) = \bigcup_{A \in \mathcal{S}} \mathcal{LC}_A(\Sigma)$. For any $X \in \mathcal{LC}(\Sigma)$ write $\mathcal{S}_\Sigma(X)$ for $\{A \in \mathcal{S} \mid X \in \mathcal{LC}_A(\Sigma)\}$. If $A \in \mathcal{S}_\Sigma(X)$ we say that X *has sort* A.

The first four clauses of Definition 2.17 give an inductive assignment of a sort to every well-sorted $\mathcal{LC}$-term for the signature Σ (henceforth called $\mathcal{LC}(\Sigma)$-*terms*). Without loss of generality, we may assume that we never follow one application of the rule (weaken) by another in constructing any such derivation, since the inclusion ordering $\leq$ determined by Δ is transitive.

We consider terms which are identical up to renaming of (sorted) variables to be the same. As with types, we will omit the sort of terms whenever possible.

If Σ is a signature with sort system $\mathcal{S}$ and sort structure Δ, and if $\sim$ is the equivalence relation determined by Δ, then by the rule (weaken), $\mathcal{LC}_A(\Sigma) = \mathcal{LC}_B(\Sigma)$ whenever $A \sim B$. By passing to the quotient signature Σ' with respect to $\sim$, *i.e.*, to the signature with sort system $\mathcal{S}'$ equal to $\mathcal{S}/\sim$ obtained by replacing all sorts in $\mathcal{S}$ by canonical $\sim$-equivalence class representatives, we arrive at a signature whose equivalence relation is trivial and such that $\mathcal{LC}_A(\Sigma') = \mathcal{LC}_A(\Sigma)$ for all sorts A. We may, and will therefore, assume without loss of generality that $\leq$ is a partial ordering for all signatures in the remainder of this paper. We will also assume that we have ridded our sort structures of redundant subsort declarations of the form $[A \leq A]$, and that whenever $\Delta \vdash A \leq B$ for some sort structure Δ, $length(A) \leq length(B)$ holds. The latter assumption is without loss of generality under a standard semantics.

A signature is said to be *subterm closed* if each subterm of a well-sorted term is again well-sorted. It is natural in the context of mathematics to expect signatures to be subterm closed, since it does not make sense to allow ill-formed subexpressions in well-sorted expressions (a situation that may be different in, for example, field of natural language processing). That signatures are subterm closed is not difficult to prove.

Lemma 2.18 *If Σ is a signature, $X \in \mathcal{LC}(\Sigma)$, and Y is a subterm of X, then $Y \in \mathcal{LC}(\Sigma)$. That is, any signature Σ is subterm closed.*

Proof. Since $X \in \mathcal{LC}(\Sigma)$, X has sort A for some sort A. The proof is by induction on the derivation of $\Sigma \vdash X : A$.

- If $\Sigma \vdash X : A$ is the conclusion of (var) or (const), then there is nothing to prove.

- If $\Sigma \vdash X \equiv UV : A$ by an application of (app) with $\Sigma \vdash U : A$, and $\Sigma \vdash V : \delta(A)$, then $\Sigma \vdash X : \gamma(A)$. Any other proper subterm Y of X is a subterm of either U or V, so that $Y \in \mathcal{LC}(\Sigma)$ by the induction hypothesis.
- If $\Sigma \vdash X \equiv \lambda x.U : B \to C \equiv A$ is the conclusion of an application of (abs) with $\Sigma \vdash U : A$ and $x \in Vars_B$, then any proper subterm Y of X is a subterm of U and so is in $\mathcal{LC}(\Sigma)$ by the induction hypothesis.
- If $\Sigma \vdash X \equiv \lambda x.Ux : A$ is the conclusion of an application of (η) with $\Sigma \vdash U : A$ and $\Delta \vdash \delta(A) = B$, then $\Sigma \vdash Ux : \gamma(A)$ and any other proper subterm Y of X is in $\mathcal{LC}(\Sigma)$ by the induction hypothesis.
- If $\Sigma \vdash X : A$ is the conclusion of an application of (weaken) with $\Sigma \vdash X : B$ and $\Sigma \vdash B \leq A$, then the result follows immediately from the induction hypothesis.

□

Although Lemma 2.18 guarantees that signatures are subterm closed, it is not necessarily true that if Y is a subterm of a term $X \in \mathcal{LC}_A(\Sigma)$, then given a sort B such that Y has sort B, there is a derivation of $\Sigma \vdash Y : B$ which is a subderivation of any given one for $\Sigma \vdash X : A$. This is because the rules (η) and (abs) provide different ways of sorting certain abstraction terms. Thus, if $\Sigma \vdash X : A$ and we would like to prove a result concerning the sort of a subterm Y of X, we must look at the derivation of $\Sigma \vdash X : A$, rather than only at the structure of X itself. On the other hand, if we do not need information about the sort of the subterm Y, then Lemma 2.18 shows that the structure of X itself may provide sufficient information for proof purposes. This observation will have both important and severe consequences for our unification algorithm (see the discussion and example in Section 3.2 below).

In any signature, variables have unique least sorts:

Lemma 2.19 *If Σ is a signature with sort structure Δ and $x \in Vars_A$, then x has least sort A in Σ,* i.e., *for all $B \in \mathcal{S}_\Sigma(x)$, $\Delta \vdash A \leq B$.*

Proof. According to Definition 2.17, if $\Sigma \vdash x : B$ for any sort $B \not\equiv A$, then this fact must be the conclusion of an application of (weaken). The result is thus immediate. □

But due to the possibility of constant overloading, it is not necessarily true that every term will have a unique least sort, *i.e.,* not every signature is a *regular* signature. Nevertheless, for arbitrary terms, there does exist some relation among the various sorts a term may have:

Lemma 2.20 *If $\Sigma \vdash X : A$ and $\Sigma \vdash X : B$, then $\tau(A) = \tau(B)$.*

Proof. By induction on the derivations of $\Sigma \vdash X : A$ and $\Sigma \vdash X : B$. If $A \leq B$ or $B \leq A$, then the lemma holds by Lemma 2.10.We may therefore assume without loss of generality that no (weaken) steps appear in the derivations of $\Sigma \vdash X : A$ and $\Sigma \vdash X : B$.

- If $\Sigma \vdash X : A$ by (var), then $\Sigma \vdash X : B$ is also the result of (var), and so $A \equiv B$.
- If $\Sigma \vdash X : A$ by (const), then $\Sigma \vdash X : B$ is also the result of (const). Thus A Rdom B by Definition 2.16, and so by the first part of Lemma 2.15, $\tau(A) = \tau(B)$.

- If $\Sigma \vdash X : A$ by (app), then $X \equiv UV$ for some U and V, and $\Sigma \vdash X : B$ is also the result of (app). The result follows immediately by applying the induction hypothesis to U.
- If $\Sigma \vdash X : A$ by (abs), then $\Sigma \vdash X : B$ is either the result of (abs) or (η). In the first case, $X \equiv \lambda x.U$ and the conclusion follows by applying the induction hypothesis to U. In the second case, $X \equiv \lambda x_E.Ux$, where $E = \delta(B)$ and U has some sort C such that $E \to \gamma(C) = A$. By the induction hypothesis, $\tau(C) = \tau(B)$. But $\tau(A) = \tau(E) \to \tau(\gamma(C)) = \tau(\delta(B)) \to \tau(\gamma(B)) = \tau(B)$.
- If $\Sigma \vdash X : A$ by (η), then $\Sigma \vdash X : B$ is the result of either (abs) or (η). The first case has, by symmetry, already been considered. In the second, $X \equiv \lambda x.Ux$, and the conclusion follows immediately by applying the induction hypothesis to U.

□

As a corollary, we observe that for every $X \in \mathcal{LC}(\Sigma)$, the set $\mathcal{S}_\Sigma(X)$ is finite, since Σ has only finitely many sorts per type.

As a further consequence of Lemmas 2.20 and 2.18, we see that if we consider the forgetful functor to be the identity on typed constants, then it can be extended to well-sorted terms by induction on the derivations proving the terms well-sorted. This extension gives an injection from $\mathcal{LC}(\Sigma)$ into $\mathcal{LC}$. The forgetful functor is not, however, bijective in general.

If Σ is a signature with exactly one sort A such that $\tau(A) = \alpha$ for each $\alpha \in \mathcal{T}_0$, and such that Δ is the empty sort structure, then Lemma 2.10 implies that the sort system $\mathcal{S}$ of Σ is isomorphic to $\mathcal{T}$ via the type assignment τ. Moreover, the set of constant declarations contains at most one declaration $[c :: A]$ per constant $c \in \mathcal{C}$, since constant declarations must respect the typing of the constants, and so $\mathcal{LC}(\Sigma)$ is isomorphic to the fragment of $\mathcal{LC}$ containing only the finitely many constants per type appearing in constant declarations in Σ.

In order to prove computability of sort assignment for $\mathcal{LC}(\Sigma)$, we extend the function $\mathcal{S}_\Sigma(\cdot)$ on $\mathcal{LC}(\Sigma)$ to all of $\mathcal{LC}$ via the forgetful functor (which, recall, provides an injection of the former into the latter).

Definition 2.21 For $X \in \mathcal{LC}$ and Σ a signature, define

$$\mathcal{S}_\Sigma(X) = \{\mathcal{S}_\Sigma(Y) \mid Y \in \mathcal{LC}(\Sigma) \text{ and } \overline{Y} \equiv X\}$$

According to this definition, $X \in \mathcal{LC} \setminus \mathcal{LC}(\Sigma)$ iff $\mathcal{S}_\Sigma(X) = \emptyset$, *i.e.*, iff there exists no $Y \in \mathcal{LC}(\Sigma)$ such that $\overline{Y} \equiv X$. If such a Y exists, it is unique; in this case, we abuse terminology and say that $X \in \mathcal{LC}$ is *well-sorted* with respect to Σ.

Theorem 2.22 *If $X \in \mathcal{LC}$ and Σ is a signature, then $\mathcal{S}_\Sigma(X)$ is effectively computable.*

Proof. We will see later (in Theorem 2.32) that η-reduction on $\mathcal{LC}(\Sigma)$ is sort-preserving, and, assuming this, we take X to be in η-normal form. Therefore, no derivation of $X \in \mathcal{LC}(\Sigma)$ can contain an application of the inference rule (η), and we need only consider derivations in the calculus for $\mathcal{LC}(\Sigma)$-term formation which do not invoke the (η) rule. We proceed by induction on the structure of X.

- If $X \equiv \overline{x_A}$, then $\mathcal{S}_\Sigma(X) = \{B \mid \Delta \vdash A \leq B\}$, which is computable by Corollary 2.13.

- If $X = c_\alpha$, then $\mathcal{S}_\Sigma(X) = \{B \mid \Delta \vdash A \leq B \text{ for some } A \in \mathcal{S} \text{ with } [c :: A] \in \Delta\}$. This set is also computable by Corollary 2.13 and the fact that signatures have finitely many constant declarations per type.

- If $X \equiv UV$, then

  $$\mathcal{S}_\Sigma(X) = \{B \mid \Delta \vdash \gamma(A) \leq B \text{ for some } A \in \mathcal{S} \text{ with } A \in \mathcal{S}_\Sigma(U) \text{ and } \delta(A) \in \mathcal{S}_\Sigma(V)\}.$$

  This set is computable by the induction hypothesis and Corollary 2.13.

- If $X \equiv \lambda\overline{x_A}.U$ with U not of the form Vx, then

  $$\mathcal{S}_\Sigma(X) = \{B \mid \Delta \vdash A \to C \leq B \text{ for some } C \in \mathcal{S} \text{ with } C \in \mathcal{S}_\Sigma(U)\}.$$

  If $U \equiv Vx$ for some V, then

  $$\begin{array}{rcl} \mathcal{S}_\Sigma(X) & = & \{B \mid \Delta \vdash A \to C \leq B \text{ for some } C \in \mathcal{S} \text{ with } C \in \mathcal{S}_\Sigma(U)\} \\ & \cup & \{B \mid B \in \mathcal{S}_\Sigma(V)\}. \end{array}$$

  These sets are both computable by the induction hypothesis and Corollary 2.13.

□

Corollary 2.23 *For any signature Σ and $X \in \mathcal{LC}$, it is decidable whether or not X is well-sorted with respect to Σ.*

We now prove that signatures respect function domains, in the sense that for every term X of functional sort and any sorts $A, B \in \mathcal{S}_\Sigma(X)$, we must have $\delta(A) = \delta(B)$. This unique domain sort is called the *supporting sort* of X and is denoted $supp(X)$. At first glance, requiring signatures to respect function domains appears to be a grave restriction on the expressiveness of a calculus. But functional extensionality itself relies heavily on the notion of explicitly specified domains of functions, which unique supporting sorts are intended to syntactically capture. Indeed, in mathematics, functions are assumed to have unique (explicitly specified) domains, and must therefore be distinguished from restrictions to subdomains. For example, the addition function on the reals must be distinguished from the addition function on the natural numbers, and in general functions f and g should only be considered the same if $fa = ga$ for all a in the common (explicitly specified) domain of f and g. Observing these distinctions is necessary for a correct treatment of extensional higher-order calculi, and they must be reflected in the syntax of any such calculus.

Lemma 2.24 *If $\Sigma \vdash X : A$ and $\Sigma \vdash X : B$, then A Rdom B. That is, any signature Σ respects function domains.*

Proof. By induction on the derivations of $\Sigma \vdash X : A$ and $\Sigma \vdash X : B$. If $A \leq B$ or $B \leq A$, then the lemma holds by the third part of Lemma 2.15. We may therefore assume without loss of generality that no (weaken) steps appear in the derivations $\Sigma \vdash X : A$ and $\Sigma \vdash X : B$.

- If $\Sigma \vdash X : A$ by (var), then $\Sigma \vdash X : B$ is also the result of (var), and so $A \equiv B$.

- If $\Sigma \vdash X : A$ by (const), then $\Sigma \vdash X : B$ is also the result of (const). Thus A Rdom B by Definition 2.16.
- If $\Sigma \vdash X : A$ by (app), then $X \equiv UV$ for some U, V, and $\Sigma \vdash X : B$ is also the result of (app). The result follows immediately by applying the induction hypothesis to U.
- If $\Sigma \vdash X : A$ by (abs), then $\Sigma \vdash X : B$ is either the result of (abs) or (η). In the first case, $X \equiv \lambda x.U$ and the conclusion follows by applying the induction hypothesis to U and using ($\leq$-cov). In the second case, $X \equiv \lambda x_E.Ux$, where $E \equiv \delta(B)$ and U has some sort C such that $E \to \gamma(C) = A$. By the induction hypothesis, C Rdom B, and since B and C are functional sorts, $\gamma(C)$ Rdom $\gamma(B)$. Then $A = E \to \gamma(C)$ implies that $\delta(A) = \delta(B) = E$, and $\gamma(A) = \gamma(C)$ Rdom $\gamma(B)$. Thus A Rdom B as desired.
- If $\Sigma \vdash X : A$ by (η), then $\Sigma \vdash X : B$ is the result of either (abs) or (η). The first case has, by symmetry, already been considered. In the second, $X \equiv \lambda x.Ux$, and the conclusion follows immediately from the induction hypothesis applied to U.

□

2.3 Order-sorted Reduction

We now fix an arbitrary signature Σ for use throughout the remainder of this paper.

As per the discussion immediately preceding Lemma 2.24, η-expansion of the term X_A to $\lambda x_B.Xx$, which corresponds to restricting the function denoted by X to the sort denoted by B, should only yield the original function again if B represents the (explicitly specified) domain of the function denoted by X. This restriction is embodied in the order-sorted η-rule below.

Definition 2.25 The following order-sorted reductions are defined for $\mathcal{LC}(\Sigma)$-terms:

- $(\lambda x.X)Y \xrightarrow{\beta} X[x := Y]$.
- $\lambda x_B.Xx \xrightarrow{\eta} X$ if $x_B \notin FV(X)$ and $B \equiv supp(X)$.

The first rule above, which we assume to happen without free variable capture, is called *order-sorted β-reduction* and the second is called *order-sorted η-reduction.* Hereafter, these relations will be referred to simply as "β-reduction" and "η-reduction" when the context is clear. Of course there are restrictions on the sorts of the terms implicit in the rules for $\mathcal{LC}(\Sigma)$-term formation. Observe, for example, that we must have $B \leq supp(X)$ in the rule for order-sorted η-reduction in order to ensure that $\lambda x.Xx \in \mathcal{LC}(\Sigma)$. But in fact we require the stronger condition that B actually be equivalent to $supp(X)$ for the sake of properly handling extensionality.

It is possible to define order-sorted β-reduction with reference to typed β-reduction by

$$X \xrightarrow{\beta} Y \text{ iff } \overline{X} \xrightarrow{\beta} \overline{Y},$$

an equivalence of which we will make much use in what follows. But in the interest of being self-contained, we prefer instead to *define* β-reduction wholly in terms of the order-sorted calculus.

Since order-sorted $\beta\eta$-reduction generalizes ordinary typed $\beta\eta$-reduction, we will write $\xrightarrow{\beta\eta}$ for order-sorted $\beta\eta$-reduction as well as for the typed version. We will similarly abuse notation in denoting the transitive, as well as the reflexive, symmetric, and transitive, closure of $\xrightarrow{\beta\eta}$, since the typed relations are subsumed by their order-sorted versions.

It is important to our program that the fundamental operations of our calculus do not allow the formation of ill-sorted terms from well-sorted ones. This will ensure that our unification algorithm never has to handle ill-sorted terms, even intermediately. In fact we show, using the next sequence of lemmas, that if $X \xrightarrow{\beta\eta}\!\!\!\!\rightarrow Y$, then $\mathcal{S}_\Sigma(X) \subseteq \mathcal{S}_\Sigma(Y)$. To see that the reverse inclusion need not hold in general, we need only consider an $\mathcal{LC}(\Sigma)$-term of the form $X \equiv (\lambda x_B.x)Y$ such that Y has sort A for some A strictly less than B. Then $X \xrightarrow{\beta} Y$, but there is no derivation proving that X has sort A.

Lemma 2.26 *If $\Sigma \vdash X : A$ and $\Sigma \vdash Y : E$, then $\Sigma \vdash X[x_E := Y] : A$.*

Proof. It is easy to see that for any (free) variable x_E occurring in X we can replace in the derivation of $\Sigma \vdash X : A$ all occurrences of the derivation hypothesis $\Sigma \vdash x : E$ with the derivation of $\Sigma \vdash Y : E$. The resulting derivation shows that $\Sigma \vdash X[x := Y] : A$. □

Lemma 2.27 *If $(\lambda x.X)Y \xrightarrow{\beta} X[x := Y]$, then $\mathcal{S}_\Sigma((\lambda x.X)Y) \subseteq \mathcal{S}_\Sigma(X[x := Y])$.*

Proof. We must show that if $\Sigma \vdash (\lambda x.X)Y : A$, then $\Sigma \vdash X[x := Y] : A$. The proof is by induction on the derivation of $\Sigma \vdash (\lambda x.X)Y : A$. Note that $\Sigma \vdash (\lambda x.X)Y : A$ must be the result of an application of (app) or (weaken). In the former case, we must have $\Sigma \vdash \lambda x.X : B$, $\Sigma \vdash Y : \delta(B)$, and $A = \gamma(B)$ for some B such that $\Sigma \vdash X : \gamma(B)$. Then by Lemma 2.26, $\Sigma \vdash X[x := Y] : \gamma(B) = A$. If $\Sigma \vdash (\lambda x.X)Y : A$ is the conclusion of (weaken), then the result follows immediately from the induction hypothesis and an application of (weaken). □

Corollary 2.28 *If $X \xrightarrow{\beta} Y$, then $\mathcal{S}_\Sigma(X) \subseteq \mathcal{S}_\Sigma(Y)$.*

Corollary 2.29 *If $X \xrightarrow{\beta}\!\!\!\!\rightarrow Y$, then $\mathcal{S}_\Sigma(X) \subseteq \mathcal{S}_\Sigma(Y)$.*

Similar although slightly stronger results hold for order-sorted η-reduction:

Lemma 2.30 *If $\lambda x_E.Xx \xrightarrow{\eta} X$, then $S_\Sigma(\lambda x.Xx) = S_\Sigma(X)$.*

Proof. If $\lambda x_E.Xx \xrightarrow{\eta} X$, then $E = \delta(A)$ for any sort A such that $\Sigma \vdash X : A$. Then by (η), $\Sigma \vdash \lambda x_E.Xx : A$ as well. Conversely, if $\Sigma \vdash \lambda x_E.Xx : A$, then we induct on the derivation of $\Sigma \vdash \lambda x.Xx : A$.

- If $\Sigma \vdash \lambda x.Xx : A$ is the conclusion of an application of (abs), then $x \in Vars_E$, $\Sigma \vdash Xx : B$, and $\Sigma \vdash \lambda x.Xx : E \to B \equiv A$ for some sort B. Then $\Sigma \vdash Xx : B$ is either the consequence of an application of (app) or of (weaken).
  - In the first case, $\Sigma \vdash X : C$ for some sort C such that $B = \gamma(C)$ and $\Sigma \vdash x : \delta(C)$. Then since signatures respect function domains, the fact that X has sort C implies that $\lambda x.Xx$ has sort $\delta(C) \to \gamma(C)$, so that $\delta(C) = E$. Thus $\Delta \vdash C \leq \delta(C) \to \gamma(C) = E \to B \equiv A$, so that $\Sigma \vdash X : A$ by (weaken).

- In the second case, we must have $\Sigma \vdash Xx : D$ for some sort D such that $\Delta \vdash D \leq B$. Since we assume without loss of generality that applications of (weaken) are consolidated, $\Sigma \vdash Xx : D$ must itself be the consequence of (app). By the previously considered subcase, $\Sigma \vdash X : A$.

- If $\Sigma \vdash \lambda x.Xx : A$ is the conclusion of an application of (η), then clearly $\Sigma \vdash X : A$.
- If $\Sigma \vdash \lambda x.Xx : A$ is the conclusion of an application of (weaken), then the result follows immediately from the induction hypothesis and (weaken).

□

Corollary 2.31 *If* $X \xrightarrow{\eta} Y$, *then* $\mathcal{S}_\Sigma(X) = \mathcal{S}_\Sigma(Y)$.

Corollary 2.32 *If* $X =_\eta Y$, *then* $S_\Sigma(X) = S_\Sigma(Y)$.

We can combine these results into

Theorem 2.33 *If* $X \xrightarrow{\beta\eta} Y$ *or* $X \xrightarrow{\beta\eta^{-1}} Y$, *then* $\mathcal{S}_\Sigma(X) \subseteq \mathcal{S}_\Sigma(Y)$.

Order-sorted $\beta\eta$-reduction satisfies the usual properties associated with typed $\beta\eta$-reduction, particularly convergence. Termination is a direct consequence of the corresponding well-known result for the simply typed lambda calculus:

Theorem 2.34 *Order-sorted $\beta\eta$-reduction on $\mathcal{LC}(\Sigma)$ is terminating.*

Proof. Any sequence of order-sorted $\beta\eta$-reductions induces a corresponding typed $\beta\eta$-reduction sequence of the same length on the $\mathcal{LC}$-terms which are the images under the forgetful functor of the $\mathcal{LC}(\Sigma)$-terms in the given order-sorted $\beta\eta$-reduction sequence. The result follows from the fact that $\beta\eta$-reduction is terminating on $\mathcal{LC}$. □

That order-sorted $\beta\eta$-reduction is convergent is only slightly harder to prove.

Theorem 2.35 *Order-sorted $\beta\eta$-reduction on $\mathcal{LC}(\Sigma)$ is convergent.*

Proof. Since order-sorted $\beta\eta$-reduction is terminating, it suffices to see that it is weakly confluent. This follows from the corresponding result for $\beta\eta$-reduction on $\mathcal{LC}$ and the fact, a consequence of Theorem 2.33, that if $X \xrightarrow{\beta\eta} Y$ then $supp(X) \equiv supp(Y)$. □

In light of Lemma 2.35, it makes sense to speak of *the* order-sorted $\beta\eta$-normal form of an $\mathcal{LC}(\Sigma)$-term; we denote it by $\beta\eta nf(X)$. It is also sensible to refer to *the* order-sorted long β-normal form of X. By this we mean the term obtained by computing the order-sorted β-normal form of X and then performing (if needed) some order-sorted η^{-1}-reductions, as in [Bre88]. Theorem 2.33 guarantees that for all $X \in \mathcal{LC}(\Sigma)$, $\mathcal{S}_\Sigma(X) \subseteq \mathcal{S}_\Sigma(\beta\eta nf(X))$ and $\mathcal{S}_\Sigma(X) \subseteq \mathcal{S}_\Sigma(l\beta nf(X))$.

3 Order-sorted Higher-order Unification

When considering unification in the simply typed lambda calculus, it is customary to work modulo η-equality (see, for example, the presentation in [Sny91]). Although it is also possible to do so in our setting, we will explicitly keep track of order-sorted η-equality, since we have seen that the interaction between extensionality and sorts can be unexpectedly subtle.

3.1 Systems and Substitutions

We begin with the basic notions required in any discussion of unification. In the following, we represent unification problems by equational systems comprising the pairs of $\mathcal{LC}(\Sigma)$-terms to be simultaneously unified, and use transformations of such systems as our main tool for solving the unification problems they represent.

Definition 3.1 A *pair* is a two-element multiset of $\mathcal{LC}(\Sigma)$-terms. A *system* is a finite set Γ of pairs. A pair is *η-trivial* (or simply *trivial*) if its elements are η-equal, and *Σ-valid* if its elements are $\beta\eta$-equal; a system is *Σ-valid* if each of its pairs is Σ-valid.

As usual, we write $\Gamma, \langle X, Y\rangle$ instead of $\Gamma \cup \{\langle X, Y\rangle\}$. But since Γ may or may not also contain $\langle X, Y\rangle$, such a decomposition is ambiguous. We will use the notation $\Gamma; \langle X, Y\rangle$ to abbreviate $\Gamma \cup \{\langle X, Y\rangle\}$ when $\langle X, Y\rangle$ is *not* a pair in Γ.

Definition 3.2 A pair $\langle X, Y\rangle$ is *solved in* Γ if it is either trivial or for some $x \in Vars_A$, $X \xrightarrow{\eta\!\!\!\!\rightarrow} x$, $A \in \mathcal{S}_\Sigma(Y)$ and there are no occurrences of x in Γ other than the one indicated. In this case, x is said to be *solved in* Γ. If each pair in Γ is solved in Γ, then Γ is a *solved system.*

Definition 3.3 A *substitution* is a finitely supported map from variables to $\mathcal{LC}(\Sigma)$; a substitution θ induces a mapping on terms, which we will also denote by θ.

We will write substitution application as juxtaposition, so that θX is the application of the substitution θ to the term X. By $D(\theta)$ and $I(\theta)$ we will denote the set of variables in the domain of θ and the set of variables introduced by θ, respectively.

Definition 3.4 A substitution θ is *well-sorted* (or a *well-sorted substitution*) if for every $x \in Vars_A$, $A \in \mathcal{S}_\Sigma(\theta x)$.

It follows that if $X \in \mathcal{LC}_A(\Sigma)$ and θ is well-sorted, then $\theta X \in \mathcal{LC}_A(\Sigma)$ as well. That the set of well-sorted substitutions is closed under composition is not hard to prove. We assume the standard results about ordinary (not necessarily well-sorted) substitutions.

We can extend equalities on $\mathcal{LC}(\Sigma)$ to (well-sorted) substitutions in the usual manner:

Definition 3.5 Let $=_*$ be an equational theory on $\mathcal{LC}(\Sigma)$, W be a set of variables, and θ and θ' be substitutions. Then $\theta =_* \theta'[W]$ means that for every variable in $x \in W$, $\theta x =_* \theta' x$. Define the subsumption relation $\theta' \leq_* \theta[W]$ to hold provided there exists a substitution ρ such that $\theta =_* \rho\theta'[W]$.

If W is the set of all variables, we drop the notation "$[W]$." We will be primarily concerned with the cases when $=_*$ is $=_{\beta\eta}$ or the empty equational theory. In the latter case, we write simply "$\equiv$" and "$\leq$" for the induced equality and subsumption ordering on substitutions.

We can extend substitutions on $\mathcal{LC}(\Sigma)$ to mappings on systems $\Gamma \equiv \{\langle X_i, Y_i\rangle \mid i \leq n\}$ by defining $\sigma\Gamma$ to be the system $\{\langle \sigma X_i, \sigma Y_i\rangle \mid i \leq n\}$. The normal form $l\beta nf(\Gamma)$, all of whose unsolved pairs comprise terms in long β-normal form, is defined similarly. If all terms in the unsolved pairs of Γ are in long β-normal form, we say that Γ is *in long β-normal form.*

We will write $FV(X)$ for the set of free variables occurring in the $\mathcal{LC}(\Sigma)$-term X and $FV(\Gamma)$ for the free variables occurring in any term in the system Γ.

Definition 3.6 A well-sorted substitution θ is a Σ-*unifier* of a system Γ if $\theta\Gamma$ is Σ-valid. If σ is a Σ-unifier of Γ with the properties that $D(\sigma) \subseteq FV(\Gamma)$ and that for any Σ-unifier θ of Γ, $\sigma \leq_{\beta\eta} \theta$, then σ is said to be a *most general* Σ-*unifier* of Γ. A system Γ is Σ-*unifiable* if there exists some Σ-unifier of Γ.

For technical reasons, given a system Γ, we will make extensive use of substitutions satisfying the conditions of the next definition. Note that we relax the standard requirement that substitutions map all variables to normal forms, and allow solved variables to be bound arbitrarily. This is justified in Lemma 3.9 below.

Definition 3.7 An idempotent well-sorted substitution θ is a *normalized* Σ-*unifier* of a system Γ if

- $D(\theta) \subseteq FV(\Gamma)$,
- θ is a Σ-unifier of Γ, and
- for all unsolved variables x in Γ, θx is in long β-normal form.

Write $U_\Sigma(\Gamma)$ for the set of all normalized Σ-unifiers of Γ.

It is clear that every well-sorted substitution θ is $\beta\eta$-equal to a well-sorted substitution θ' with $D(\theta) = D(\theta')$ and $\theta' x$ in long β-normal form for each $x \in D(\theta)$. Such a substitution θ' is said to be *in long* β-*normal form*. Thus for any Σ-unifier θ of a system Γ, there exists a $\theta' \in U_\Sigma(\Gamma)$ such that $\theta' =_{\beta\eta} \theta[FV(\Gamma)]$. In particular, every Σ-unifiable system has a normalized Σ-unifier.

The remainder of this section explores the relationship between systems and their unifiers.

If Γ is a solved system whose non-trivial pairs are $\langle X_1, Y_1\rangle, ..., \langle X_n, Y_n\rangle$ with $X_i \xrightarrow{\eta\!\!\!\rightarrow} x_i$ for $i = 1, ..., n$, then these pairs determine an idempotent well-sorted substitution $\sigma_\Gamma = \{x_1 \mapsto Y_1, ..., x_n \mapsto Y_n\}$. Note, however, that such a pair $\langle X, Y\rangle$ with $X \xrightarrow{\eta\!\!\!\rightarrow} x \in Vars_A$ and $Y \xrightarrow{\eta\!\!\!\rightarrow} y \in Vars_A$ requires a choice as to which of x and y is to be in the domain of the substitution. We will assume that a uniform way exists for making such a choice, and so will refer to *the* well-sorted substitution determined by a solved system. On the other hand, idempotent well-sorted substitutions can be represented by solved systems without trivial pairs. If σ is such a substitution, write $[\sigma]$ for any solved system which represents it.

Note that any system Γ can be written as $\Gamma'; [\sigma]$ where σ is the set of solved pairs in Γ. Call $[\sigma]$ the *solved part* of Γ.

Transformation-based unification methods attempt to reduce systems to be unified to solved systems which represent their unifiers. The fundamental connection between solved systems and Σ-unifiers is the following fact, which shows that solved systems indeed represent their own solutions:

Lemma 3.8 *If* $\Gamma = \{\langle X_1, Y_1\rangle, ..., \langle X_n, Y_n\rangle\}$ *is a solved system, then* σ_Γ *is a most general* Σ-*unifier for* Γ. *In fact, for any* Σ-*unifier* θ *of* Γ, $\theta =_{\beta\eta} \theta\sigma_\Gamma$.

Proof. Clearly σ_Γ is a Σ-unifier of Γ. Suppose that the non-trivial pairs of Γ are $\langle X_{i_j}, Y_{i_j}\rangle$, where $X_{i_j} \xrightarrow{\eta\!\!\!\rightarrow} x_{i_j}$ for $j = 1, ..., k$. If θ is any Σ-unifier of Γ, then $\theta\sigma_\Gamma x_{i_j} \equiv \theta Y_{i_j} =_{\beta\eta} \theta x_{i_j}$ for $j = 1, ..., k$, and $\theta x \equiv \theta\sigma_\Gamma x$ for $x \notin D(\sigma_\Gamma)$, so that indeed $\theta =_{\beta\eta} \theta\sigma_\Gamma$. □

In general, however, a system Γ will not have a single most general Σ-unifier, and may not even have a finite *complete set of Σ-unifiers, i.e.,* a finite set U of Σ-unifiers such that for every Σ-unifier θ of Γ there exists a substitution $\sigma \in U$ such that $\sigma \leq_{\beta\eta} \theta[FV(\Gamma)]$. This quarrelsome behaviour has nothing to do with sorts, however — it is inherited from the unsorted calculus ([Gou66]).

The next lemma will be used to show that we need not be concerned with solved pairs when computing Σ-unifiers, and is therefore consistent with the intuition that the solved part of a system is merely a record of an answer substitution being constructed.

Lemma 3.9 *Suppose Γ is a Σ-unifiable system with solved part $[\sigma]$ and unsolved part Γ'. If θ is a Σ-unifier of Γ, then for every Σ-unifier ρ of Γ' such that $D(\rho) \subseteq FV(\Gamma')$ and $\rho \leq_{\beta\eta} \theta[FV(\Gamma')]$, $\rho\sigma$ is a Σ-unifier of Γ and $\rho\sigma \leq_{\beta\eta} \theta[FV(\Gamma)]$.*

Proof. Let θ be a Σ-unifier of Γ, and let ρ be a Σ-unifier of Γ' such that $D(\rho) \subseteq FV(\Gamma')$ and $\rho \leq \theta[FV(\Gamma')]$. Then $\rho\sigma$ Σ-unifies Γ since $\rho\sigma[\sigma]$ and $\rho\sigma\Gamma' \equiv \rho\Gamma'$ are Σ-valid. Since θ Σ-unifies $[\sigma]$ and σ is idempotent, we have $\theta\sigma =_{\beta\eta} \theta$. Finally, $FV(\Gamma') \subseteq FV(\Gamma)$ implies that $\rho\sigma \leq_{\beta\eta} \theta\sigma =_{\beta\eta} \theta[FV(\Gamma)]$. □

Finally, observe that terms having different sorts can be unifiable, since sorts can be altered by substitutions via constant declarations. Unlike the case for higher-order unification in the absence of constant declarations, we cannot insist that each pair in a unification problem have precisely the same sorts without sacrificing the completeness of our method — the possibility of pairs of terms not having the same sorts does not, of course, preclude unifiability of the terms. Nevertheless, Σ-unifiable terms must have the same types.

3.2 The Unification Algorithm

One of the key steps for unification in the presence of sorts is solving the following problem: given a term $X \equiv \lambda x_1 ... x_k . h U_1 ... U_n \in \mathcal{LC}_A(\Sigma)$ in long β-normal form, find a term $G \in \mathcal{LC}_A(\Sigma)$ with head h which can be instantiated to yield X. This is a generalization of a similar problem in $\mathcal{LC}$, and in [Hue75], Huet gives a set of *partial bindings* capable of approximating any $\mathcal{LC}$-term in the sense just described. Even in the presence of functional base sorts, Huet-style partial bindings indeed suffice to approximate arbitrary $\mathcal{LC}(\Sigma)$-terms, although not necessarily with terms of the appropriate sorts. This is because, by contrast with the assignment of types to terms in $\mathcal{LC}$ based on term structure, sort assignment in $\mathcal{LC}(\Sigma)$ is not structural. That is, the *derivation* that a given term has a given sort, rather than just the structure of the term itself, must be consulted when deducing sort information.

Partial bindings will emerge as our main analytical tool for investigating completeness properties of our unification algorithm. But they are also necessary for defining the transformations on which the algorithm will be based. Below, a variable will be called *fresh* if it does not appear in any term in the current context.

Definition 3.10 If h is an atom such that either $h \in Vars_C$ or $[h :: C]$ is a constant declaration in Σ, then a *partial binding of sort A for head h* is any term of the form

$$G \equiv \lambda y_1 ... y_l . h V_1 ... V_m$$

where

- $l = length(A)$,
- $m = l + length(\tau(C)) - length(\tau(A)) \geq 0$,
- $y_j \in Vars_{\delta^j(A)}$ for $j = 1, ..., l$,
- $V_i \equiv z_i y_1 ... y_l$, for $1 \leq i \leq m$, where $z_i \in Vars_{\delta^1(A) \to ... \to \delta^l(A) \to \delta^i(C)}$ is fresh, and
- $\Delta \vdash \gamma^m(C) \leq \gamma^l(A)$.

Note that partial bindings for a given sort A and head h need not exist because of the second and the last conditions of Definition 3.10, but because signatures respect function domains, when they do exist, they are unique up to renaming of the variables z_i. If Σ is a signature without functional base sorts, then the partial bindings are η-expanded; in particular, if Σ is a signature with exactly one sort per (base) type, then the partial bindings are precisely those obtained for $\mathcal{LC}$ (modulo the isomorphism between $\mathcal{T}$ and $\mathcal{S}$ discussed after Lemma 2.20).

Call a partial binding $G \equiv \lambda y_1 ... y_l . h V_1 ... V_m$ a j^{th} *projection binding* if $h \equiv y_j$ and an *imitation binding* if $h \in FV(G) \cup \mathcal{C}$. Write $\mathcal{G}^h_A(\Sigma)$ for the set of partial bindings of sort A for head h.

It turns out that we cannot insist that partial bindings be η-expanded, as is traditional, without sacrificing completeness of our algorithm (see Example 3.15). That our partial bindings are η-expandable, in general, is the primary difference between them and the Huet-style bindings of [Sny91], but this liberalization suffices to recover completeness in the presence of functional base sorts.

The following are examples of partial bindings.

Example 3.11 Let A, B, and C be base sorts, with C functional such that $\delta(C) = A$ and $\gamma(C) = B$. Let Σ be a signature with no subsort declarations and the single constant declaration $[c :: C]$. Then

- $\mathcal{G}^c_C(\Sigma) = \{c\}$,
- $\mathcal{G}^c_B(\Sigma) = \{cz \mid z \in Vars_A \text{ is fresh}\}$, and
- $\mathcal{G}^c_{A \to B}(\Sigma) = \{\lambda x . c(zx) \mid z \in Vars_{A \to A} \text{ is fresh and } x \in Vars_A\}$.

The following lemma justifies the terminology for partial bindings.

Lemma 3.12 *If* $G \in \mathcal{G}^h_A(\Sigma)$, *then* $\Sigma \vdash G : A$.

Proof. Let $G \equiv \lambda y_1 ... y_l . h V_1 ... V_m$ as in Definition 3.10. Then there is some $C \in \mathcal{S}$ such that either $h \in Vars_C$ or $[h :: C]$ is a constant declaration in Σ. It follows that $\Sigma \vdash V_i : \delta^i(C)$ for $i = 1, ..., m$, so that by successively applying (app), we see that $\Sigma \vdash h V_1 ... V_m : \gamma^m(C)$. An application of (weaken) shows that $\Sigma \vdash h V_1 ... V_m : \gamma^l(A)$, and we conclude via a series of l applications of (abs) that $\Sigma \vdash \lambda y_1 ... y_l . h V_1 ... V_m : \delta^1(A) \to ... \to \delta^l(A) \to \gamma^l(A)$. But $\delta^1(A) \to ... \to \delta^l(A) \to \gamma^l(A) \equiv A$ since $l = length(A)$, so that indeed $\Sigma \vdash G : A$ as desired. □

At the end of this subsection we present an algorithm $\Sigma\mathcal{U}$ which is a *complete* Σ-unification method, *i.e.*, which is such that, for any system Γ and a Σ-unifier θ of Γ, there exists a computation of Algorithm $\Sigma\mathcal{U}$ on input system Γ yielding a Σ-unifier σ of Γ with $\sigma \leq_{\beta\eta} \theta[FV(\Gamma)]$ (see Theorem 3.29). That is, for any system Γ, Algorithm $\Sigma\mathcal{U}$ can produce a Σ-unifier of Γ which is more general than any given Σ-unifier of Γ. The following transformations on which Algorithm $\Sigma\mathcal{U}$ is based are adapted from those of [Sny91].

Definition 3.13 The set $\Sigma\mathcal{T}$ comprises the following transformations on systems in long β-normal form.

- DECOMPOSE: For any atom h,

$$\Gamma; \langle \lambda x_1 ... x_k . h X_1 ... X_n, \lambda x_1 ... x_k . h U_1 ... U_n \rangle \Longrightarrow$$
$$\Gamma, \langle \lambda x_1 ... x_k . X_1, \lambda x_1 ... x_k . U_1 \rangle, ..., \langle \lambda x_1 ... x_k . X_n, \lambda x_1 ... x_k . U_n \rangle.$$

- ELIMINATE: If $x \in Vars_A$, $x \notin \{x_1, ..., x_k\}$, $x \notin FV(\lambda x_1 ... x_k . X)$, and $\sigma = \{x \mapsto \lambda x_1 ... x_k . X\}$ is well-sorted, then

$$\Gamma; \langle \lambda x_1 ... x_k . x x_1 ... x_k, \lambda x_1 ... x_k . X \rangle \Longrightarrow \langle x, \lambda x_1 ... x_k . X \rangle, \sigma\Gamma.$$

- IMITATE: If $x \in Vars_A$, $h \in \mathcal{C}$ or $h \in FV(\lambda x_1 ... x_k . h U_1 ... U_m)$, $h \not\equiv x$, and $G \in \mathcal{G}^h_A(\Sigma)$ is an imitation binding, then

$$\Gamma; \langle \lambda x_1 ... x_k . x X_1 ... X_n, \lambda x_1 ... x_k . h U_1 ... U_m \rangle \Longrightarrow$$
$$\Gamma, \langle x, G \rangle, \langle \lambda x_1 ... x_k . x X_1 ... X_n, \lambda x_1 ... x_k . h U_1 ... U_m \rangle.$$

- j-PROJECT: If $x \in Vars_A$, h is a (possibly bound) atom and $G \in \mathcal{G}^h_A(\Sigma)$ is a j^{th} projection binding for some $j \in \{1, ..., n\}$ such that $head(X_j) \in \mathcal{C}$ implies $head(X_j) \equiv h$, then

$$\Gamma; \langle \lambda x_1 ... x_k . x X_1 ... X_n, \lambda x_1 ... x_k . h U_1 ... U_m \rangle \Longrightarrow$$
$$\Gamma, \langle x, G \rangle, \langle \lambda x_1 ... x_k . x X_1 ... X_n, \lambda x_1 ... x_k . h U_1 ... U_m \rangle.$$

- GUESS: If h is any atom, and x and y are free variables in $Vars_A$ and $Vars_B$, respectively, both distinct from h, and $G \in \mathcal{G}^h_A(\Sigma)$, then

$$\Gamma; \langle \lambda x_1 ... x_k . x X_1 ... X_n, \lambda x_1 ... x_k . y U_1 ... U_m \rangle \Longrightarrow$$
$$\Gamma, \langle x, G \rangle, \langle \lambda x_1 ... x_k . x X_1 ... X_n, \lambda x_1 ... x_k . y U_1 ... U_m \rangle.$$

As part of the transformations IMITATE, j-PROJECT, and GUESS, we immediately apply ELIMINATE to the new pair $\langle x, G \rangle$. This has the effect of applying the well-sorted substitution $\{x \mapsto G\}$ to the rest of the obtained system.

We write $\Gamma \Longrightarrow \Gamma'$ if Γ reduces to Γ' by one application of a transformation in $\Sigma\mathcal{T}$ and implicitly assume that applications of the transformations are such that all terms involved are well-sorted.

Our notation exploits the fact that pairs are unordered. We adopt the convention that no transformations may be done out of solved or trivial pairs. This accords with the intuition that the solved pairs in a system are merely recording an answer substitution as it is incrementally built up. Notice that if $\Gamma \Longrightarrow \Gamma'$, then $\{x \mid x \text{ is solved in } \Gamma\} \subseteq \{x \mid x \text{ is solved in } \Gamma'\}$, so that solved variables remain solved after application of a transformation from $\Sigma\mathcal{T}$.

We emphasize that there is no deletion of trivial pairs in this presentation, and that this design choice simplifies certain arguments. For example, this guarantees that if $\Gamma \Longrightarrow \Gamma'$, then $FV(\Gamma) \subseteq FV(\Gamma')$, so that when a fresh variable is chosen during a computation it is guaranteed to be new to the entire computation. As a consequence, we need not manipulate the "protected sets of variables" typically found in completeness proofs in the literature. This convention also eliminates complications in proving the soundness of resolution procedures based on our unification algorithm, and respects the fundamental idea behind the use of transformations for describing algorithms, namely that the logic of the problem being considered can be abstracted from implementational issues such as choice of data structures and flow of control.

We can now define our order-sorted higher-order unification algorithm.

Definition 3.14 The non-deterministic algorithm $\Sigma\mathcal{U}$ is the process of repeatedly

1. reducing all terms of the unsolved pairs in the system to long β-normal form and then applying some transformation in $\Sigma\mathcal{T}$ to a non-trivial unsolved pair, and
2. returning a most general Σ-unifier if at any point in the computation the system becomes solved.

Observe that the choice of pair upon which Algorithm $\Sigma\mathcal{U}$ is to act is non-deterministic, as is the choice of rule from $\Sigma\mathcal{T}$ to be applied.

We illustrate use of the Algorithm $\Sigma\mathcal{U}$:

Example 3.15 Let $[b :: \delta(A)]$ and $[c :: A]$ be constant declarations in a signature Σ with a functional base sort A. Let $f \in Vars_A$, $x \in Vars_{\delta(A)}$, and $w \in Vars_{A \to \delta(A)}$, and consider the Σ-unifiable long β-normal form system

$$\Gamma \equiv \langle fx, cb\rangle, \langle wc, b\rangle.$$

Applying IMITATE with partial binding c to the first pair of Γ yields

$$\langle f, c\rangle, \langle cx, cb\rangle, \langle wc, b\rangle.$$

An application of DECOMPOSE results in

$$\langle f, c\rangle, \langle x, b\rangle, \langle wc, b\rangle,$$

and an application of IMITATE with binding $\lambda y.b$ for $y \in Vars_A$ to the third pair, followed by some β-reductions give the solved system

$$\Gamma' \equiv \langle f, c\rangle, \langle x, b\rangle, \langle w, \lambda y.b\rangle, \langle b, b\rangle,$$

from which we extract the well-sorted substitution $\sigma = \{f \mapsto c, x \mapsto b, w \mapsto \lambda y.b\}$. Anticipating Theorem 3.21, we conclude that σ is a Σ-unifier of Γ' and hence of Γ.

Note that if we instead allow only η-expanded partial bindings, then the only possible IMITATE step binds f to $\lambda y.c(zy)$ for a variable y and a fresh variable z of appropriate sorts. But then ELIMINATE cannot be performed on the pair $\langle f, \lambda y.c(zy)\rangle$ (as is required to complete the IMITATE step) without creating ill-sorted terms in the remainder of Γ, since $\Sigma \vdash \lambda y.c(zy) : A$ does not hold, and so in particular, $w(\lambda y.c(zy))$ is not well-sorted.

Yet while unification in $\mathcal{LC}(\Sigma)$ is apparently more delicate than unification in $\mathcal{LC}$, the extra care pays off when sort information disallows certain undesirable unifications that would be possible in an unsorted calculus. That sorts can serve as a unification search space filter is indeed the primary motivation for their introduction.

Example 3.16 Let Σ be a signature with base sorts D, I, and R, where the non-functional sort R is intended to denote the real numbers, and the functional sorts D and I denote the strictly decreasing and strictly increasing functions on the reals, respectively. Suppose further that $\delta(D) = \delta(I) = R$ and $\gamma(D) = \gamma(I) = R$. Finally, let $[n :: D \rightarrow I]$ and $[4 :: R]$ comprise the set of constant declarations of Σ, where n is intended to denote the "negation functor" mapping each function F to $-F$, and 4 denotes the real number four.

Let $x \in Vars_R$, $f \in Vars_I$, and $g \in Vars_D$, and consider the unification problem given by the pairs

$$\Gamma \equiv \langle f4, ngx\rangle, \langle gx, 4\rangle.$$

Note that 1-PROJECT does not apply to $\langle f4, ngx\rangle$ because of the condition on the head of the projection term in that rule, and that because of the requirement that $m \geq 0$ for partial bindings, an application of IMITATE to that pair is the only possibility for computation on the system. Letting z be fresh from $Vars_D$, we see that $nz \in \mathcal{G}^n_I(\Sigma)$, and so apply IMITATE with this binding for f to get

$$\langle f, nz\rangle, \langle nz4, ngx\rangle, \langle gx, 4\rangle.$$

By reasoning similar to that for the original system we conclude that only DECOMPOSE applies here, resulting in

$$\langle f, nz\rangle, \langle z, g\rangle, \langle x, 4\rangle, \langle gx, 4\rangle.$$

Two applications of ELIMINATE yield

$$\langle f, ng\rangle, \langle z, g\rangle, \langle x, 4\rangle, \langle g4, 4\rangle,$$

all of whose pairs, save the last — unsolvable — one, are solved. The only alternative to eliminating z above is applying GUESS to $\langle z, g\rangle$ in the second system, but such a step makes no progress toward a solution. Anticipating Theorem 3.29, we conclude that the original system is unsolvable, in accordance with the facts that neither the identity function nor the function which is constantly four is strictly decreasing.

Of course, if we were to interpret D as denoting the (not strictly) decreasing real-valued functions on the reals, then we would want to be able to compute the function whose value is constantly four as a binding for g. So while considering the original system Γ as a typed system permits too many bindings for certain applications, a system supporting only constant declarations may permit too few. A calculus allowing arbitray term declarations finds a middle road: in a signature with a term declaration assigning the term $\lambda y.4$ to be of sort D when $y \in Vars_R$, Γ would have precisely the desired solutions.

3.3 Soundness and Completeness of the Algorithm

The proof that our transformations are sound is not appreciably different from the proof for the corresponding transformations for unification in $\mathcal{LC}$ (as, presented, for example, in [Sny91]). We therefore only outline the proof.

Lemma 3.17 *If* $\Gamma \Longrightarrow \Gamma'$ *using* ELIMINATE, *then for any well-sorted substitution* θ, θ *is a* Σ*-unifier of* Γ *iff it is a* Σ*-unifier of* Γ'.

Lemma 3.18 *If* $\Gamma \Longrightarrow \Gamma'$ *using* DECOMPOSE, *where the pair in* Γ *which is transformed is* $\langle X, Y\rangle \equiv \langle \lambda x_1 ... x_k.hX_1...X_n, \lambda x_1...x_k.hY_1...Y_n\rangle$, *then for any well-sorted substitution* θ,

1. *if* $h \in \mathcal{C}$, h *is bound in* X *or* Y, *or* $h \in FV(X) \cup FV(Y)$ *but* $h \notin D(\theta)$, *then* θ *is a* Σ*-unifier of* Γ *iff it is a* Σ*-unifier of* Γ'.
2. *if* $h \in D(\theta)$, *then* θ *is a* Σ*-unifier of* Γ *if it is a* Σ*-unifier of* Γ'.

We do not have θ a Σ-unifier of Γ implying that it is a Σ-unifier of Γ' in general, as the following example from [Sny91] shows.

Example 3.19 Applying DECOMPOSE to a system may lose Σ-unifiers: the system $\langle fab, fcd\rangle$, where $f \in Vars_A$, $A \notin \mathcal{S}_0^f$, and $a, b, c, d \in \mathcal{C}$, has infinitely many Σ-unifiers, but the system $\langle a, c\rangle, \langle b, d\rangle$ obtained by applying DECOMPOSE has none.

Moreover, in applying IMITATE, j-PROJECT, and GUESS, we effectively commit ourselves to a particular approximation of a solution, and so cannot reasonably expect any Σ-unifier of an original system Γ to be a Σ-unifier of the obtained system Γ' as well.

Lemma 3.20 *If* $\Gamma \Longrightarrow \Gamma'$ *by* DECOMPOSE, IMITATE, j-PROJECT, *or* GUESS, *then* θ *is a* Σ*-unifier of* Γ *if it is a* Σ*-unifier of* Γ'.

Theorem 3.21 *(Soundness) If* $\Gamma \Longrightarrow \Gamma'$, *then for any well-sorted substitution* θ, θ *is a* Σ*-unifier of* Γ *if it is a* Σ*-unifier of* Γ'.

So if algorithm $\Sigma\mathcal{U}$ is run on initial system Γ and returns a well-sorted substitution θ, then θ is indeed a Σ-unifier of Γ. The main result of this section and of this paper is a converse, namely that given an initial system Γ and Σ-unifier θ, $\Sigma\mathcal{U}$ can compute a Σ-unifier σ of Γ which is more general than θ.

We require a few technical lemmas.

Lemma 3.22 *If* $Y \equiv \lambda x_1...x_p.hU_1...U_q \in \mathcal{LC}_A(\Sigma)$ *is in* $\beta\eta$*-normal form, then* $p \leq length(A)$.

Proof. By induction on the derivation of $\Sigma \vdash Y : A$.

- If $\Sigma \vdash Y : A$ is the conclusion of an application of (var), (const), or (app), then $p = 0$ and so there is nothing to prove.
- If $\Sigma \vdash Y : A$ is the conclusion of an application of (weaken), then the result follows immediately from the induction hypothesis and the fact that if $B \leq A$ then $length(B) \leq length(A)$.

- If $\Sigma \vdash Y : A$ is the conclusion of an application of (abs), then

$$\frac{x_1 \in Vars_B \qquad \Sigma \vdash \lambda x_2 ... x_p.hU_1...U_q : D}{\Sigma \vdash \lambda x_1 ... x_p.hU_1...U_q : B \to D \equiv A}$$

  Since $\lambda x_2...x_p.hU_1...U_q$ is in $\beta\eta$-normal form, the induction hypothesis guarantees that $p - 1 \leq length(D) = length(A) - 1$. That $p \leq length(A)$ is therefore immediate.

□

Lemma 3.23 *(Structure Lemma) If $Y \equiv \lambda x_1...x_p.hU_1...U_q \in \mathcal{LC}_A(\Sigma)$ is in $\beta\eta$-normal form, then either $h \in Vars_C$ or $[h :: C]$ is a constant declaration in Σ for some sort C such that $length(A) + length(\tau(C)) - length(\tau(A)) \geq 0$ and $\Delta \vdash \gamma^q(C) \leq \gamma^p(A)$.*

Proof. By induction on the derivation of $\Sigma \vdash Y : A$.

- If $\Sigma \vdash Y : A$ is the conclusion of an application of (var) or (const), then $p = 0$, $q = 0$, and C can be taken to be A.
- If $\Sigma \vdash Y : A$ is the conclusion of an application of (app), then $p = 0$ and

$$\frac{\Sigma \vdash hU_1...U_{q-1} : B \qquad \Sigma \vdash U_q : \delta(B)}{\Sigma \vdash hU_1...U_q : \gamma(B) = A}$$

  for some sort B. Since $hU_1...U_{q-1}$ is in $\beta\eta$-normal form, the induction hypothesis guarantees that either $h \in Vars(C)$ or there is a constant declaration $[h :: C]$ in Σ for some sort C such that $length(B) + length(\tau(C)) - length(\tau(B)) \geq 0$ and $\Delta \vdash \gamma^{q-1}(C) \leq B$. Since $\Delta \vdash B \leq \delta(B) \to A$, we have $length(B) \leq 1 + length(A)$ and $length(\tau(B)) = 1 + length(\tau(A))$, so that $length(A) + length(\tau(C)) - length(\tau(A)) \geq 0$. By Lemma 2.9, $\Delta \vdash \gamma^q(C) \leq \gamma(B) \equiv A$ as well.
- If $\Sigma \vdash Y : A$ is the conclusion of an application of (abs), we must have

$$\frac{x_1 \in Vars_B \qquad \Sigma \vdash \lambda x_2 ... x_p.hU_1...U_q : D}{\Sigma \vdash \lambda x_1 ... x_p.hU_1...U_q : B \to D \equiv A}$$

  Since $\lambda x_2...x_p.hU_1...U_q$ is in $\beta\eta$-normal form, the induction hypothesis guarantees that either $h \in Vars(C)$ or there is a constant declaration $[h :: C]$ in Σ for some sort C such that $length(D) + length(\tau(C)) - length(\tau(D)) \geq 0$ and $\Delta \vdash \gamma^q(C) \leq \gamma^{p-1}(D) \equiv \gamma^p(A)$. But since $length(A) = length(D) + 1$ and $length(\tau(A)) = length(\tau(D)) + 1$, we have $length(A) + length(\tau(C)) - length(\tau(A)) \geq 0$ as desired.
- If $\Sigma \vdash Y : A$ is the conclusion of (weaken), then

$$\frac{\Sigma \vdash Y : B \qquad \Delta \vdash B \leq A}{\Sigma \vdash Y : A}$$

  By the induction hypothesis, either $h \in Vars(C)$ or there is a constant declaration $[h :: C]$ in Σ for some sort C such that $length(B)+length(\tau(C))-length(\tau(B)) \geq 0$ and $\Delta \vdash \gamma^q(C) \leq \gamma^p(B)$. But since $\Delta \vdash B \leq A$ implies $length(B) \leq length(A)$, and because $\tau(A) = \tau(B)$, we have $length(A) + length(\tau(C)) - length(\tau(A)) \geq 0$. Finally, since $\Delta \vdash B \leq A$, iterating Lemma 2.9 yields $\Delta \vdash \gamma^p(B) \leq \gamma^p(A)$, and so by transitivity of $\leq$, $\Delta \vdash \gamma^q(C) \leq \gamma^p(A)$.

□

Lemma 3.24 *(Partial Binding Lemma) If* $X \equiv \lambda x_1...x_k.hU_1...U_n \in \mathcal{LC}_A(\Sigma)$ *is in long* β*-normal form, then there exist a partial binding* $G \in \mathcal{G}^h_A(\Sigma)$ *and a well-sorted substitution* ρ *in long* β*-normal form such that*

1. $D(\rho)$ *is precisely the set of fresh variables in* G,
2. ρz *has smaller depth than* X *for each* $z \in D(\rho)$, *and*
3. $\rho G =_{\beta\eta} X$.

Proof. Let $Y \equiv \lambda x_1...x_p.hU'_1...U'_q$ be the $\beta\eta$-normal form of X, where $U_i \xrightarrow{\eta\twoheadrightarrow} U'_i$ for $i = 1,...,q$, $p \leq k$, and $n = q+(k-p)$. Let $C \in \mathcal{S}$ be a sort such that either $h \in Vars_C$ or $[h :: C]$ is a constant declaration in Σ, $m = length(A)+length(\tau(C))-length(\tau(A)) \geq 0$, and $\Delta \vdash \gamma^q(C) \leq \gamma^p(A)$, whose existence is guaranteed by the Structure Lemma. Furthermore, let $G \equiv \lambda x_1...x_l.hV_1...V_m \in \mathcal{G}^h_A(\Sigma)$, where $V_i = z_i x_1...x_l$ for fresh variables z_i, $i = 1,...,m$. Observe that $l \leq length(\tau(A)) = k$ and $n = length(\tau(C))$, so that $m = l+n-k = l+q-p$. Since $\Sigma \vdash Y : A$ by Corollary 2.32, Lemma 3.22 ensures that $p \leq l \leq k$. Lemma 2.9 implies that the substitution ρ whose components are

$$\begin{array}{lcl} z_1 & \mapsto & \lambda x_1...x_l.U_1 \\ & \vdots & \\ z_q & \mapsto & \lambda x_1...x_l.U_q \\ z_{q+1} & \mapsto & \lambda x_1...x_l.x_{p+1} \\ & \vdots & \\ z_m & \mapsto & \lambda x_1...x_l.x_l \end{array}$$

is well-sorted, has domain consisting precisely of the set of fresh variables in G, and has the property that ρz has smaller depth than X for each $z \in D(\rho)$. It is well-defined because $m-q = l-p \geq 0$. Finally,

$$\begin{array}{lll} \rho(G) & \equiv & \rho(\lambda x_1...x_l.hV_1...V_m) \\ & =_\beta & \lambda x_1...x_l.hU_1...U_q x_{p+1}...x_l \\ & =_\eta & \lambda x_1...x_p.hU'_1...U'_q \\ & =_\eta & X \end{array}$$

as desired. □

Note that with the Huet-style partial bindings, it would not necessarily be possible to find G of sort A and a substitution ρ as required:

Example 3.25 If Σ is a signature with a constant declaration $[c :: A]$ for a functional base sort A, then the only derivation of $\Sigma \vdash \lambda x.cx : A$ is

$$\dfrac{\dfrac{[c :: A] \in \Sigma}{\Sigma \vdash c : A}}{\Sigma \vdash \lambda x.cx : A}$$

Any Huet-style partial binding that might approximate the long β-normal form $\lambda x.cx$ must be of the form $\lambda x.c(zx)$ where z is a fresh variable of an appropriate sort. But

there is no derivation of $\Sigma \vdash \lambda x.c(zx) : A$ — in particular, such a derivation cannot be obtained by applying the rule (η) by way of mimicking the above derivation. Under our definition, however, $G \equiv c$ is itself a partial binding of sort A for head h, and ρ can be taken to be the identity substitution.

The following measure will provide the basis for proving termination of Algorithm $\Sigma\mathcal{U}$.

Definition 3.26 The measure μ is defined for all systems Γ and substitutions θ by

$$\mu(\Gamma, \theta) = \langle \mu_1(\Gamma, \theta), \mu_2(\Gamma) \rangle,$$

where $\mu_1(\Gamma, \theta)$ is the multiset of the depths of the θ-bindings of unsolved variables in Γ which are also in $D(\theta)$, and $\mu_2(\Gamma)$ is the multiset of depths of terms in Γ.

We require a final consolidating lemma to prove completeness of Algorithm $\Sigma\mathcal{U}$.

Lemma 3.27 *Let $\theta \in U_\Sigma(\Gamma)$ and let $\langle X, Y \rangle$ be a non-trivial unsolved pair in a system Γ in long β-normal form. Then there exist a system Γ' and a substitution θ' such that*

$$\Gamma \Longrightarrow \Gamma'$$

and

1. $\theta \equiv \theta'[FV(\Gamma)]$,
2. $\theta' \in U_\Sigma(\Gamma')$, *and*
3. $\mu(\Gamma', \theta') < \mu(\Gamma, \theta)$.

Proof. If $head(X) \equiv head(Y) \notin D(\theta)$, then since $\langle X, Y \rangle$ is not trivial, it is not hard to see that DECOMPOSE applies. By Lemma 3.18, $\theta \in U_\Sigma(\Gamma')$, and $\mu(\Gamma', \theta) < \mu(\Gamma, \theta)$ since $\mu_1(\Gamma', \theta)$ is no larger than $\mu_1(\Gamma, \theta)$ and $\mu_2(\Gamma') < \mu_2(\Gamma)$.

Otherwise, either $head(X) \not\equiv head(Y)$ or else $head(X) \equiv head(Y) \in D(\theta)$. In either of these cases, one of X and Y has an unsolved variable $x \in D(\theta) \cap Vars_A$ of Γ as its head (since X and Y are Σ-unifiable); without loss of generality, assume X does. Then since θ is well-sorted, $\Sigma \vdash \theta x : A$, and θx is in long β-normal form since θ is normalized. Suppose $\theta x \equiv \lambda x_1 ... x_k.hU_1 ... U_n$. Then by Lemma 3.24, there exist $G \in \mathcal{G}^h_A(\Sigma)$ and a well-sorted substitution ρ in long β-normal form satisfying the conclusions of that lemma. Therefore,

- if $head(Y) \notin D(\theta)$ and $h \equiv head(Y)$, then IMITATE applies.
- if $head(Y) \notin D(\theta)$ and $h \not\equiv head(Y)$, then j-PROJECT applies for some j.
- if $head(Y) \in D(\theta)$, then GUESS applies.

Let $\theta' = \theta \cup \rho$. Clearly $\theta \equiv \theta'[FV(\Gamma)]$, $\theta \in U_\Sigma(\Gamma')$ since $\theta \in U_\Sigma(\Gamma)$ and ρ is in long β-normal form, and $D(\rho)$ is exactly the set of fresh variables in G. Moreover, $\mu_1(\Gamma', \theta') < \mu_1(\Gamma, \theta)$: x is removed from the set of unsolved variables in Γ which appear in $D(\theta)$, and is replaced by the set of fresh variables of G, but for each such variable z, $\theta' z \equiv \rho z$ is smaller than θx. Thus $\mu(\Gamma', \theta') < \mu(\Gamma, \theta)$.

Observe also that in case $head(X) \equiv head(Y) \notin D(\theta)$ does not hold, but $X \xrightarrow{\eta} x \in Vars_A$ and x is not free in Y and $\Sigma \vdash Y : A$, then ELIMINATE applies. In this case, we can take θ' to be θ, by noting that $\mu_1(\Gamma', \theta) < \mu_1(\Gamma, \theta)$, which implies that $\mu(\Gamma', \theta) < \mu(\Gamma, \theta)$. □

The proof of Lemma 3.27 shows that it is possible to restrict DECOMPOSE to apply only when $head(X) \equiv head(Y) \notin D(\theta)$, but there is no way of encoding this restriction into the transformations since θ cannot be mentioned there.

If we call a transformation prescribed by Lemma 3.27 a *μ-prescribed* transformation, then each application of a μ-prescribed transformation actually decreases the well-founded measure μ. Thus any sequence of μ-prescribed transformations must terminate. The previous lemma also guarantees that any system obtained by repeatedly applying μ-prescribed transformations must be solved. That is, any sequence of μ-prescribed transformations must terminate in a solved system.

Corollary 3.28 *If Γ is a Σ-unifiable system in long β-normal form to which no μ-prescribed transformation in $\Sigma\mathcal{T}$ applies, then Γ is solved.*

Proof. If Γ is not solved, then there is a non-trivial unsolved pair $\langle X, Y\rangle$ in Γ. Since Γ is Σ-unifiable, there exists a substitution $\theta \in U_\Sigma(\Gamma)$. By the last lemma, $\langle X, Y\rangle$ admits an application of some μ-prescribed transformation from $\Sigma\mathcal{T}$. □

Theorem 3.29 *(Completeness) Let θ be a Σ-unifier of Γ. Then there exists a computation of Algorithm $\Sigma\mathcal{U}$ on Γ producing a Σ-unifier σ of Γ such that $\sigma \leq_{\beta\eta} \theta[FV(\Gamma)]$.*

Proof. Since every Σ-unifier of Γ is pointwise $\beta\eta$-equal on $FV(\Gamma)$ to some $\theta' \in U_\Sigma(\Gamma)$, we may prove the theorem under the additional hypothesis that $\theta \in U_\Sigma(\Gamma)$.

If Γ is not in long β-normal form, then perform reductions until a system in long β-normal form results. Note that if θ Σ-unifies Γ, then θ also Σ-unifies $l\beta nf(\Gamma)$, and that this reduction is a $\Sigma\mathcal{U}$ step. We may therefore assume without loss of generality in the remainder of this proof that Γ is in long β-normal form. We induct on the length of the longest sequence of μ-prescribed sequence of transformations available out of Γ.

If no μ-prescribed transformation from $\Sigma\mathcal{T}$ applies to Γ, then by Corollary 3.28, Γ is solved so we may return a most general Σ-unifier σ of Γ whose existence is guaranteed by Lemma 3.8. This action is a step of Algorithm $\Sigma\mathcal{U}$, and indeed $\sigma \leq_{\beta\eta} \theta$.

If some μ-prescribed transformation from $\Sigma\mathcal{T}$ applies to Γ, yielding a system Γ' and a substitution θ' satisfying the conclusion of Lemma 3.27, then applying this transformation is a $\Sigma\mathcal{U}$ step. By the induction hypothesis, there is a computation of $\Sigma\mathcal{U}$ on Γ' producing a Σ-unifier δ of Γ' such that $\delta \leq_{\beta\eta} \theta'[FV(\Gamma')]$. It follows from Lemma 3.21 that δ is a Σ-unifier of Γ, and since $FV(\Gamma) \subseteq FV(\Gamma')$, $\delta \leq_{\beta\eta} \theta'[FV(\Gamma)]$. But $\theta' \equiv \theta[FV(\Gamma)]$, so that $\delta \leq_{\beta\eta} \theta[FV(\Gamma)]$ as desired. □

Since we have not made any assumption about the order in which transformations from $\Sigma\mathcal{T}$ are performed, and since any application of ELIMINATE to a system reduces the measure μ, we infer that the strategy of eager variable elimination is complete for unification in our calculus, just as it is for unification in the simply typed lambda calculus. It is not, however, known to be true that eager variable elimination is complete for an arbitrary calculus and equational theory, even if both are first-order.

3.4 Pre-unification

As with unification in the simply typed lambda calculus, the rule GUESS (whose analogue in [Sny91] is called FLEX-FLEX for reasons explained momentarily) gives rise to a serious explosion of the search space for unifiers. But unfortunately, the "guessing" of partial

bindings engendered by GUESS cannot be avoided without sacrificing completeness of Algorithm $\Sigma\mathcal{U}$. Huet solved this problem in the simply typed lambda calculus by redefining the higher-order unification problem to a form sufficient for refutation purposes: *flex-flex pairs*, *i.e.*, pairs of terms in long β-normal form both of which have variables at the head, are considered *pre-unified*, or already solved. We conjecture that it is possible to define an appropriate notion of pre-unification in our setting as well, but sound the following warnings against a naive adaptation of the standard methods.

- Pre-unification only makes sense in regular signatures, as can be seen by considering the non-regular signature Σ with only two incomparable sorts A and B, constant declarations $[c :: A]$ and $[c :: B]$, and the pair $\langle x_A, y_B \rangle$. This pair has Σ-unifier $\sigma = \{x \mapsto c, y \mapsto c\}$, but σ can only be found by applying the rule GUESS.

- The standard way (in the simply typed lambda calculus) to generate trivial unifiers for flex-flex pairs is to

  - define, for every type $\alpha \equiv \alpha_1 \rightarrow ... \rightarrow \alpha_n \rightarrow \alpha_0$, $n \geq 0$, a term $X_\alpha \equiv \lambda x_1 ... x_n . v$ where $\tau(x_i) = \alpha_i$ for $i = 1, ..., n$, and $v \in Vars_{\alpha_0}$ is a fresh variable which will never be used in any other term, and then to
  - define an (infinite) set of bindings $\psi = \{x \mapsto X_\alpha \mid x \in Vars_\alpha\}$, which is then restricted to the free variables of the flex-flex pairs under consideration to yield a unifying substitution.

  But in the presence of functional base sorts we cannot even get off the ground with this program. We can, of course, define for each $A \in \mathcal{S}$ a term $X_A = \lambda x_1 ... x_n . v$ where $x_i \in Vars_{\delta^i(A)}$ for $i = 1, ..., n$, and $v \in Vars_{\gamma^n(A)}$ is a fresh variable, and $\gamma^n(A)$ is non-functional. We can also construct an analogue ψ' of ψ, and restrict it to an actual substitution. But this induced substitution need not be well-sorted — indeed any component $\{x_A \mapsto X_A\}$, where A is a functional base sort, *will not* be well-sorted. Moreover, for a functional base sort A, no term X of sort A capable of absorbing arbitrary arguments to $x \in Vars_A$ can be given in general. Of course, if A is an arrow sort composed solely of non-functional base sorts, then X_A suffices as for the simply typed lambda calculus.

- A non-trivial flex-flex pair $\Gamma \equiv \langle \lambda x_1 ... x_k . h U_1 ... U_n, \lambda x_1 ... x_k . h' V_1 ... V_n \rangle$ in long β-normal form with $h, h' \in Vars_A$ for a functional base sort A is Σ-unifiable iff the arguments to h and h' can be made identical under some well-sorted substitution, *i.e.*, iff application of DECOMPOSE out of such a pair preserves solutions. This is because, as remarked above, we cannot bind h or h' to abstraction terms capable of absorbing arbitrary arguments U_i and V_i without losing well-sortedness of the Σ-unifier being constructed.

- The existence of unifiers for flex-flex pairs depends heavily on the lattice structure of the inclusion ordering of the signature Σ under which unification is being considered. For example, if $[a :: \delta(A)]$ is the only constant declaration in Σ, then the flex-flex pair $\langle xa, ya \rangle$ with $x \in Vars_A$, $y \in Vars_B$, and A and B functional base sorts such that $\delta(A) = \delta(B)$, is Σ-unifiable iff there exists a sort D such that $D \leq A$ and $D \leq B$. In that case, if $z \in Vars_D$, then $\{x \mapsto z, y \mapsto z\}$ is a Σ-unifier of $\langle xa, ya \rangle$.

And there may yet be other issues to consider.

4 Conclusion, Related Work, and Future Directions

We have developed an order-sorted lambda calculus with functional base sorts and constant overloading which is suitable for use in automatic theorem proving applications, and proved sound and complete a transformation-based unification algorithm for this calculus. Our calculus can be seen as a subcalculus of the one proposed in [Koh92], but corrected to be well-defined (see the problematic clauses 4 and 5 of Definition 2.5 there) and provably subterm closed, to have effectively enumerable (indeed effectively computable) sort assignment, and to correctly incorporate the principle of extensionality, whose fundamental importance to any calculus intended to automate real mathematics is attested to both in Remark 2.10 there and in Section 2.2 above. With regard to the latter, we specifically introduce the (η) rule of Definition 2.17 to guarantee that sorts are invariant under η-equality — in the absence of such a provision, a term X with functional base sort A is guaranteed to have the sort $\delta(A) \rightarrow \gamma(A)$ of the η-equivalent term $\lambda x_{\delta(A)}.Xx$, but the latter will not in general have sort A (indeed this situation obtains in the calculus proposed in [Koh92]). In particular, this means that contrary to stated intention, X and $\lambda x.Xx$ cannot possibly represent the same (mathematical) function. Care about extensionality, as the reader is reminded in Section 2.3, is also required in defining order-sorted η-reduction in the presence of partially ordered sorts.

Once we have a well-defined order-sorted calculus permitting constant overloading in hand, we give transformations inducing a suitable (non-deterministic) unification algorithm. Our transformations are generalizations of those of Huet ([Hue75]), modified to accommodate the subtleties arising from the introduction of functional base sorts. In particular, because one of the consequences of extensionality under such a system is that sort assignment is not based solely on term structure (as it is in the simply typed lambda calculus), we are forced to consider partial bindings more general than those considered by Huet to insure that our unification transformations are well-defined. Indeed, in the IMITATE, j-PROJECT, and GUESS transformations, it is necessary (in the notation of Definition 3.13) to bind the variable $x \in Vars_A$ to a partial binding G *of sort* A in order that the application of ELIMINATE which is part of those transformations does not create ill-sorted terms in the resulting system. This point is overlooked in [Koh92], where for signatures all of whose term declarations are constant declarations, the partial bindings have exactly the same structure as Huet's general bindings.

Our analysis then proceeds essentially by analogy with that of the simply typed calculus, except that by contrast with the situation there, in the presence of functional base sorts it is not immediate that, given a term X of sort A, we can always approximate X by a partial binding of sort A. For our more liberal notion of partial binding this is in fact true (see Lemma 3.24), but fails under the Huet-like definition (see Examples 3.15 and 3.25). Allowing partial bindings which are not necessarily η-expanded remedies this difficulty completely.

4.1 Related Work

Work related to that presented here can be roughly classified into three major areas. A brief discussion of each will serve to place the current investigation in a proper context.

Sorted first-order systems: The primary impetus for the study of order-sorted higher-order logics is the dramatic reduction in the search space associated with deduction achieved by incorporating sort information into first-order calculi. These

successes are detailed in, for example, [Wal88], [Coh89], and [Sch89]. Our results generalize unification results for the pure calculus considered by Walther, as well as for certain of the more expressive calculi considered by Schmidt-Schauß. But while these authors give in addition full refutation calculi for the logics they study, we have not yet begun to undertake that task (see Section 4.2). Decidability of the unification problems considered, complexity of the proposed unification algorithms, and sizes of complete sets of unifiers are also studied in the first-order setting, yet such investigations are not meaningful for order-sorted higher-order calculi since the unification problem even for unsorted (*i.e.*, simply typed) higher-order logic is known to be undecidable ([Hue73], [Gol81]), and minimal complete sets of higher-order unifiers need not exist in general ([Hue76]).

Refinement sorts: The type structure of the simply typed lambda calculus can be refined in ways other than that presented here, depending on which semantic features the resulting calculi are intended to syntactically capture. But for all calculi with refinement sorts, care is taken to ensure that sorts respect the underlying type structure of the calculus, a feature which is essential for the effective computability of sort assignment.

Nipkow and Qian ([NQ92]) consider a collection of sort systems parameterized by rules for contravariance in the domain sort and present a unification algorithm for the resulting sorted calculi. Functional terms in these calculi do not have unique supporting sorts in general, and the consequent difficulties with extensionality are solved by studying unification under sorted equalities which have been restricted to appropriate domain sorts — such restrictions enable specification of a well-defined η-rule. Constant overloading and functional base sorts are not present in these calculi.

In [KP93], Frank Pfenning and the second author consider a calculus with intersection sorts, *i.e.*, which has, for any sorts A and B that refine the same type, a sort $A\&B$ denoting the intersection of the sets denoted by A and B. This calculus also supports contravariance in the domain sort and constant overloading. Permitting intersection sorts makes it possible to define a minimal sort for every term, so that all signatures are regular. In this setting, problems with extensionality are alleviated by allowing only typed abstractions and by defining a term $\lambda x_\alpha.X$ to have the sort $A \rightarrow B$ iff X has sort B whenever x has sort A. η-equality is then a typed relation which preserves the sorts of terms. This calculus has been generalized by Pfenning ([Pfe92]) to a lambda calculus with dependent types, which will be used as a logical framework extending LF in the ELF programming language ([Pfe91]).

The calculi mentioned here allow only for sorting the universe of individuals, and so are not directly comparable, in terms of expressive power, with ours. Indeed, these calculi represent a principally different approach to deduction which appears to call for a semantics where functions are total, rather than partial, functions on the denotations of types, and where sorted functions are restrictions of these.

Polymorphic sorts: The works of Cardelli ([Car88]), Bruce and Longo ([BL90]), Curien and Ghelli ([CG91]), and Pierce ([Pie91]) treat variants of the system $F_{\leq}$ which encompass polymorphic intersection types (*i.e.*, intersection types whose variables are explicitly quantified), and the interaction between these types

and various subsort relations. These calculi serve as computational models for functional programming languages and are much more expressive than those studied here, but since they are not intended for deduction purposes, their unification problems have not been addressed. In such calculi, subsort declarations are not required to respect the functional structure of types, rendering the decidability of sort assignment a very complex issue.

4.2 Future Directions

Of course, the unification algorithm described here is but a small contribution to the development of calculi suitable for mechanizing real mathematics. Extracting from this work an appropriate pre-unification algorithm is in fact crucial to any effort to implement refutation calculi for the extensional lambda calculus with ordered function sorts and constant overloading presented here. As discussed in Section 3.4, such extraction is not apparently straightforward, and is currently under investigation by the second author, as is an extension to an order-sorted higher-order calculus with arbitrary term declarations and functional base sorts.

Although for automated deduction purposes higher-order logic is typically presented in terms of the lambda calculus, recent successes in developing combinatory logic based unification algorithms ([Dou93], [DJ92], [Vit92]) suggest that this alternate, algebraic formulation of higher-order logic can provide a computational framework for the mechanization of pure higher-order logic and its more expressive extensions. Transporting to a combinatory logic setting the features of the lambda calculi described in this paper is work in progress by the first author.

The results reported in this paper, as well as subsequent related work, will be implemented in the prototypical order-sorted higher-order resolution theorem prover currently under construction at the Universität des Saarlandes in Saarbrücken.

References

[ALMP84] P. B. Andrews, E. Longini-Cohen, D. Miller, and F. Pfenning. Automating Higher-order Logics. *Contemporary Mathematics* 29, pp. 169 – 192, 1984.

[Bar84] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, revised edition. North-Holland, Amsterdam, 1984.

[BL90] K. B. Bruce and G. Longo. A Modest Model of Records, Inheritance, and Bounded Quantification. *Information and Computation* 87, pp. 196 – 240, 1990.

[Bre88] V. Breazu-Tannen. Combining Algebra and Higher-Order Types. In *Proceedings of the Third Annual Symposium on Logic and Computer Science*, IEEE, pp. 82 – 90, 1988.

[Car88] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation* 76, pp. 138 – 164, 1988.

[CG91] P.-L. Curien and G. Ghelli. Subtyping + Extensionality: Confluence of $\beta\eta$top reduction in $F_{\leq}$. In Springer-Verlag LNCS 526, pp. 731 – 749, 1991.

[Coh89] A. G. Cohn. Taxonomic Reasoning with Many-sorted Logics. *Artifical Intelligence Review* 3, pp. 89 – 128, 1989.

[DJ92] D. J. Dougherty and P. Johann. A Combinatory Logic Approach to Higher-order E-unification. In Springer-Verlag LNAI 607, pp. 79 – 93, 1992. Expanded version submitted, *Theoretical Computer Science.*

[Dou93] D. J. Dougherty. Higher-order Unification via Combinators. *Theoretical Computer Science* 114, pp. 273 – 298, 1993.

[FP91] T. Freeman and F. Pfenning. Refinement Types for ML. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, ACM, pp. 268 – 277, 1991.

[Gol81] W. Goldfarb. The Undecidability of the Second-order Unification Problem. *Theoretical Computer Science* 13, pp. 225 – 230, 1981.

[Gor85] M. Gordon. HOL: A Machine Oriented Formulation of Higher-order Logic. University of Cambridge, Computer Laboratory, Report 68, 1985.

[Gou66] W. E. Gould. A Matching Procedure for Omega-Order Logic. Dissertation, Princeton University, 1966.

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

[Hue72] G. Huet. Constrained Resolution: A Complete Method for Higher Order Logic. Dissertation, Case Western Reserve University, 1972.

[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 Languages d'Ordre $1, 2, ...\omega$. Thèse d'Etat, Universitè de Paris VII, 1976.

[Koh92] M. Kohlhase. An Order-sorted Version of Type Theory. In Springer-Verlag LNAI 624, pp. 421 – 432, 1992. An expanded version of this paper appears as SEKI-Report SR-91-18 (SFB), Universität des Saarlandes, Saarbrücken, Germany.

[KP93] M. Kohlhase and F. Pfenning. Unification in a λ-calculus with Intersection Types. To appear in *Proceedings of the International Logic Programming Symposium*, 1993.

[Lus92] E. L. Lusk. Controlling Redundancy in Large Search Spaces: Argonne-style Theorem Proving Through the Years. In Springer-Varlag LNAI 624, pp. 96 – 106, 1992.

[Mil91] D. Miller. A Logic Programming Language with Lambda Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation* 2, pp. 497 – 536, 1986.

[Mit90] J. C. Mitchell. Type Inference with Simple Types. *Journal of Functional Programming* 1, pp. 245 – 285, 1991.

[NQ92] T. Nipkow and Z. Qian. Reduction and Unification in Lambda Calculi with Subtypes. In Springer-Verlag LNAI 607, pp. 66 – 78, 1992.

[Obe62] A. Oberschelp. Untersuchung zur Mehrsortigen Quantorenlogik. *Mathematische Annalen* 145, pp. 297 – 333, 1962.

[OS89] H.-J. Ohlbach and J. Siekmann. The Markgraph Karl Resolution Procedure. In *Computational Logic — Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, eds., MIT Press, pp. 41 – 112, 1989.

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

[Pfe91] F. Pfenning. Logic Programming in the LF Logical Framework. In *Logical Frameworks*, G. Huet and G. D. Plotkin, eds., Cambridge University Press, 1991.

[Pfe92] F. Pfenning. Intersection Types for a Logical Framework. POP-Report, Carnegie Mellon University, 1992.

[Pie91] B. C. Pierce. Programming with Intersection Types and Bounded Polymorphism. Dissertation, Carnegie Mellon University, 1991.

[Qia90] Z. Qian. Higher-order Order-sorted Algebras. In Springer-Verlag LNCS 463, pp. 86 – 100, 1990.

[Qia91] Z. Qian. Extensions of Order-sorted Algebraic Specifications: Parameterization, Higher-order Functions and Polymorphism. Dissertation, Universität Bremen, 1991.

[Sch89] M. Schmidt-Schauß. Computational Aspects of an Order-sorted Logic with Term Declarations. Springer-Verlag LNAI 395, 1989.

[Sny91] W. Snyder. A Proof Theory for General Unification. Birkhäuser Boston, 1991.

[van91] D. van Dalen. Summer School on Logic, Languages, and Information, Saarbrücken, 1991.

[Vit92] M. Vittek. A Combinatory Logic Rewriting Relation which Supports Narrowing. Presented at the Sixth International Workshop on Unification, Dagstuhl, 1992.

[Wal88] C. Walther. Many-sorted Unification. *Journal of the ACM* 35, pp. 1 – 17, 1988.