Higher-order Unification with Dependent Function Types*

Conal M. Elliott

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Internet: conal@cs.cmu.edu

Abstract

Roughly fifteen years ago, Huet developed a complete semidecision algorithm for unification in the simply typed λ-calculus ($\lambda_{\rightarrow}$). In spite of the undecidability of this problem, his algorithm is quite usable in practice. Since then, many important applications have come about in such areas as theorem proving, type inference, program transformation, and machine learning.

Another development is the discovery that by enriching $\lambda_{\rightarrow}$ to include *dependent function types*, the resulting calculus (λ_{Π}) forms the basis of a very elegant and expressive Logical Framework, encompassing the syntax, rules, and proofs for a wide class of logics.

This paper presents an algorithm in the spirit of Huet's, for unification in λ_{Π}. This algorithm gives us the best of both worlds: the automation previously possible in $\lambda_{\rightarrow}$, and the greatly enriched expressive power of λ_{Π}. It can be used to considerable advantage in many of the current applications of Huet's algorithm, and has important new applications as well. These include automated and semi-automated theorem proving in encoded logics, and automatic type inference in a variety of encoded languages.

1. Introduction

In the past few years, higher-order unification ("HOU$_{\rightarrow}$", *i.e.*, unification in the simply typed λ calculus, "$\lambda_{\rightarrow}$") has found a rapidly increasing number of applications. In spite of the undecidability of the problem [7], experience has shown that Huet's semidecision algorithm [13] is quite usable in practice. The first applications were to theorem proving in higher-order logic, using resolution [12] and later matings [1]. Another was to use $\lambda_{\rightarrow}$ to encode the syntax of programming languages, and in particular the scopes of variable bindings. One can then express program transformation rules without many of the previously required complicated side conditions, and use HOU$_{\rightarrow}$ to apply them [14,22,8,4]. A related area of application is to encode the syntax and inference rules of various logics and use HOU$_{\rightarrow}$ to apply inference rules (in either direction) [19,5]. Quite recently, it has been shown that HOU$_{\rightarrow}$ is exactly what is required for

*This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415, and in part by NSF Grant CCR-8620191.

solving the problem of partial type inference in the ω-order polymorphic λ-calculus [21]. HOU$_{\rightarrow}$ has even been applied to machine learning to extract justified generalizations from training examples [3]. In applications such as these, HOU$_{\rightarrow}$ is usually coupled with some other search mechanism. To meet the needs of such applications, Nadathur and Miller developed a programming language "λProlog", which is an extension of traditional Prolog both in using a richer logic (including explicit quantification and implication), and in using $\lambda_{\rightarrow}$ terms instead of the usual first-order terms (and thus requiring HOU$_{\rightarrow}$) [18,15].

A parallel development is the Edinburgh Logical Framework (LF) [10]. Here, $\lambda_{\rightarrow}$ is enriched with *dependent function types*. The resulting calculus, "λ_{Π}", together with the *judgments as types* principle forms the basis of a very elegant and expressive system encompassing the syntax, rules, and proofs for a wide class of *object-logics*.[1]

This paper presents an algorithm for HOU$_{\Pi}$, *i.e.*, unification in λ_{Π}, that was inspired by Huet's algorithm for HOU$_{\rightarrow}$. Many of the applications of HOU$_{\rightarrow}$ listed above would benefit from the addition of dependent types. For instance, the encoding of a typed object-language in $\lambda_{\rightarrow}$ usually cannot directly capture its typing rules. To account for the object-language's type system generally requires extra nontrivial mechanisms for specifying object-language typing rules and checking or inferring object-language types. In λ_{Π}, one can represent object-language typing rules directly in the language's signature. For some object-languages, HOU$_{\Pi}$ can then do automatic object-language type checking and inference. For object-languages with considerably more complicated type systems, we can still do some of the object-type checking or inference automatically, and return the remaining work as a theorem proving task. It is important to note, though, that the necessary theorem proving can be handled in the logical framework provided by λ_{Π}, (and can therefore be automated or semi-automated).

As outlined in [10], LF is a "first step towards a general theory of interactive proof checking and proof construction." HOU$_{\Pi}$ can be of significant assistance in two ways. First, it allows us to go beyond purely interactive theorem proving, to do automated (and semi-automated) theorem proving in LF encoded logics. These theorem provers could be expressed, *e.g.*, in a λProlog based on λ_{Π}, or in Elf, a language for logic definition and verified meta-programming [20]. (Implementations of both of these languages are currently under development.) Experience with typed programming languages has shown automatic type inference to be of considerable practical value [9,2,28]. In λ_{Π}, a new problem arises: It is also often possible to infer some of the subterms of a term, in the sense that no other subterms would result in a well-typed term. It is precisely this λ_{Π} *term inference* that provides for object-language type inference. As we will show, HOU$_{\Pi}$ turns the λ_{Π} type checking algorithm into a type checking and term inference algorithm.[2] By a simple extension of HOU$_{\Pi}$ to handle type variables, which we have implemented, one can do automatic type inference as well. (See [4, Chapter 6] for a similar extension to HOU$_{\rightarrow}$.)

The rest of this paper is structured as follows: Section 2 covers some preliminaries, including the language and typing rules of λ_{Π}, the notion of *approximate well-typedness* and some concepts involved in unification. Section 3 discusses the difficulties in adapting Huet's algorithm to λ_{Π}. Section 4 develops our algorithm for HOU$_{\Pi}$ based on a collection of transformations on unification problems. Section 5 contains a proof that the "solved form" unification problems

[1] We refer to an encoded logic as an *object-logic* to stress the difference between the language being represented, and the calculus in which it is represented. The latter is often referred to as the *meta-logic*. When the encoded language is not necessarily a logic, we will often use the more general terms *object-language* and *meta-language*.

[2] Term inference does not subsume theorem proving in object-logics because it may leave free variables in the resulting terms.

constructed by our algorithm are always unifiable. Section 6 presents an algorithm for λ_Π type checking and term inference. Finally, Section 7 discusses related work, and Section 8 summarizes the paper and points to future work.

2. Preliminaries

2.1. The Language and its Typing Rules

The calculus in which we do unification is essentially the one used by LF [10]. However, to simplify the exposition, and because it does not add any expressive power to the calculus, we omit the LF type λ, which explicitly forms functions from terms to types.[3] Letting the meta-variables M and N range over terms, A and B (and sometimes α, σ, and τ) over "types" (including type families), and K over kinds, the language is as follows:

$$M ::= \mathsf{c} \mid v \mid M\,N \mid \lambda v{:}A.\,M$$

$$A ::= \mathsf{c} \mid \Pi v{:}A.\,B \mid A\,M$$

$$K ::= \mathsf{Type} \mid \Pi v{:}A.\,K$$

We will often use the abbreviation "$A{\rightarrow}B$" for "$\Pi v{:}A.\,B$" when v is not free in B, and similarly for kinds.

The typing rules for λ_Π may be found in [10]. Below we reproduce just enough to understand the rest of this paper. A typing judgment involves a *signature* Σ assigning types and kinds to constants, a *context* Γ assigning types to variables. We omit well-formedness considerations for signatures, contexts, and kinds. Constants and variables are looked up in Σ and Γ respectively. The other rules are, for the kind of a type,[4]

$$\frac{\Gamma \vdash_\Sigma A \in \mathsf{Type} \qquad \Gamma, v{:}A \vdash_\Sigma B \in \mathsf{Type}}{\Gamma \vdash_\Sigma \Pi v{:}A.\,B \in \mathsf{Type}}$$

$$\frac{\Gamma \vdash_\Sigma A \in \Pi v{:}B.\,K \qquad \Gamma \vdash_\Sigma M \in B}{\Gamma \vdash_\Sigma A\,M \in [\,M/v\,]K}$$

$$\frac{\Gamma \vdash_\Sigma A \in K \qquad K =_{\beta\eta} K' \qquad \Gamma \vdash_\Sigma K' \text{ kind}}{\Gamma \vdash_\Sigma A \in K'}$$

and, for the type of a term,

$$\frac{\Gamma \vdash_\Sigma A \in \mathsf{Type} \qquad \Gamma, v{:}A \vdash_\Sigma M \in B}{\Gamma \vdash_\Sigma \lambda v{:}A.\,M \in \Pi x{:}A.\,B}$$

$$\frac{\Gamma \vdash_\Sigma M \in \Pi v{:}A.\,B \qquad \Gamma \vdash_\Sigma N \in A}{\Gamma \vdash_\Sigma M\,N \in [\,N/v\,]B}$$

$$\frac{\Gamma \vdash_\Sigma M \in A \qquad A =_{\beta\eta} B \qquad \Gamma \vdash_\Sigma B \in \mathsf{Type}}{\Gamma \vdash_\Sigma M \in B}$$

[3] These λ's never appear in normal form terms. Of course, we do allow constants of functional kind.

[4] We write "$\Gamma, v{:}A$ for the result of extending the context Γ by the assignment of the type A to the variable v and, for simplicity, we will assume that v is not already assigned a type in Γ. (This can always be enforced by α-conversion.) Also, in these rules, the $=_{\beta\eta}$ relation refers to convertability between types or kinds using the β or η conversion rules on the terms contained in these types or kinds. We take α-conversion for granted.

2.2. Approximate Typing and Normalization

Huet's algorithm depends on normalizability and the Church-Rosser property, and both may fail with ill-typed terms.[5] However, as we will see, in HOU_Π we are forced to deal with ill-typed terms. A key factor in the design of our algorithm is how we deal with ill-typedness, and to that end, we define a notion of *approximate well-typedness*, and state some of its properties.

We begin by defining two approximation functions, one that maps a term M into a simply typed term $\overline{M}$, and another that maps a type A into a simple type $\overline{A}$:[6]

Definition 1 *The* approximation *of a term or type is given by*

$$\begin{aligned} \overline{\mathsf{c}} &= \mathsf{c} \\ \overline{v} &= v \\ \overline{M\,N} &= \overline{M}\,\overline{N} \\ \overline{\lambda x{:}A.\,M} &= \lambda x{:}\overline{A}.\,\overline{M} \end{aligned}$$

and

$$\begin{aligned} \overline{\mathsf{c}} &= \mathsf{c} \\ \overline{\Pi x{:}A.\,B} &= \overline{A}{\to}\overline{B} \\ \overline{A\,M} &= \overline{A} \end{aligned}$$

Approximation is extended to kinds by setting $\overline{K} = \mathsf{Type}$ *for all* K*, and to contexts and signatures in the obvious way.*

Definition 2 *A term* M has the approximate type A in Γ *iff* $\overline{\Gamma} \vdash_{\overline{\Sigma}} \overline{M} \in A$. *(Note then that* A *is a simple type.) In this case, we say that* M *is* approximately well-typed *in* Γ.

Even approximately well-typed terms do not have the strong normalization or Church-Rosser properties, but they do have weaker normalization and uniqueness properties, which will suffice for our algorithm.

Definition 3 *A term is in* long $\beta\eta$ head normal form *(*LHNF *for short) if it is*

$$\lambda x_1{:}\sigma_1.\ \cdots \lambda x_k{:}\sigma_k.\ a\,M_1 \cdots M_m$$

for some $k \geq 0$ *and* $m \geq 0$*, where* a *is a variable or constant, and* $a\,M_1 \cdots M_m$ *is not of approximate function type. Given a term in LHNF as above, its* head *is* a*, and its* heading *is* $\lambda x_1.\ \cdots \lambda x_k.\ a$. *(Note that if the term is approximately well-typed, the heading determines* m*, but not* $\sigma_1, \ldots, \sigma_k$ *or* $M_1, \ldots, M_m$.*)*

We can now state our normalization and uniqueness properties:

Theorem 4 *Every approximately well-typed term has a LHNF. Furthermore, every LHNF of an approximately well-typed term has the same heading (modulo* α*-conversion).*

[5] For non-normalizability, consider $(\lambda x{:}\sigma.\ x\,x)\,(\lambda x{:}\sigma.\ x\,x)$. This is ill-typed for any choice of σ, and has no normal form. For loss of Church-Rosser, consider $(\lambda x{:}\sigma.\,(\lambda y{:}\tau.\,y)\,x)$, which β-reduces to $(\lambda x{:}\sigma.\,x)$, but η-reduces to $(\lambda y{:}\tau.\,y)$.

[6] This translation is similar to the translation in [10] of λ_Π terms into untyped terms.

Proof sketch: The properties follow from the normalizability and Church-Rosser properties of well-typed terms in $\lambda_{\rightarrow}$ [11,23], since given an approximately well-typed term M, any reduction sequence on $\overline{M}$ (which is well-typed in $\lambda_{\rightarrow}$) can be paralleled on M, and noting that we can follow normalization by enough η-expansions to convert to long form (where the body $a\,M_1 \cdots M_m$ is not of approximate function type). □

Because of this uniqueness property, we will apply the words "head" and "heading" to an arbitrary approximately well-typed term M to mean the head or heading of any LHNF of M.

This normal form allows us to make an important distinction (as in HOU$_{\rightarrow}$) among approximately well-typed terms:

Definition 5 *Given an approximately well-typed term M whose heading is $\lambda x_1. \cdots \lambda x_n. a$, we will call M* rigid *if a is either a constant or one of the x_i, and* flexible *otherwise.*

The value of this distinction is that applying a substitution cannot change the heading of a rigid approximately well-typed term.

2.3. Unification

Our formulation of higher-order unification is a generalization of the usual formulation that is well suited for exposition of the algorithm. First we need to define well-typed and approximately well-typed substitutions:

Definition 6 *The set $\Theta_{\Gamma}^{\Gamma'}$ of* well-typed substitutions from Γ to Γ' *is the set of those substitutions θ such that for every $v\colon A$ in Γ, we have $\Gamma' \vdash_{\Sigma} \theta v \in \theta A$. The set Θ_{Γ} of* well-typed substitutions over Γ *is the union over all well-formed contexts Γ' of $\Theta_{\Gamma}^{\Gamma'}$. Similarly the* approximately well-typed substitutions *from Γ to Γ' are those θ such that $\overline{\Gamma'} \vdash_{\overline{\Sigma}} \overline{\theta v} \in \overline{\theta A}$.*

For our purposes, a unification problem is with respect to a *unification context* Γ, and is made up of an initial substitution θ_0 and a disagreement set[7] D whose free variables are in Γ. The intent is to compose the unifiers of D with θ_0. More precisely,

Definition 7 *A* unification problem *is a triple $\langle \Gamma, \theta_0, D\rangle$ consisting of a context Γ, a substitution $\theta \in \Theta_{\Gamma_0}^{\Gamma}$ for some context Γ_0, and a set D of pairs of terms whose free variables are typed by Γ.*

Definition 8 *The* set of unifiers *of a unification problem $\langle \Gamma, \theta_0, D\rangle$ is*[8]

$$\mathcal{U}(\langle \Gamma, \theta_0, D\rangle) \triangleq \{\, \theta \circ \theta_0 \mid \theta \in \Theta_{\Gamma} \wedge \forall \langle M, M'\rangle \in D.\, \theta M =_{\beta\eta} \theta M' \,\}$$

Note that when θ_0 is an identity substitution and D is a singleton set, we have the usual problem of simply unifying two terms.

This specification serves as an organizational tool. What we really want to implement is something like the most general unifiers computed in first-order unification. However, in the higher-order case, (a) there are not unique most general unifiers, (b) even producing a "complete set of unifiers", the set of whose instances forms the set of all solutions, becomes at a certain

[7] We adopt the standard term *disagreement pair* for a pair of terms to be unified, and *disagreement set* for a set of disagreement pairs to be simultaneously unified.

[8] We use functional order composition, *i.e.*, $(\theta \circ \theta_0)M = \theta(\theta_0 M)$.

point too undirected to be useful, and (c) it is not possible to eliminate redundant solutions [12]. Huet's idea of *pre-unification* [12] (implicit in [13]) solved these difficulties. For us, each pre-unifier of a unification problem $\langle \Gamma, \theta_0, D \rangle$ is a new unification problem $\langle \Gamma', \theta', D' \rangle$, such that each solution of $\langle \Gamma', \theta', D' \rangle$ is a solution of $\langle \Gamma, \theta_0, D \rangle$, and $\langle \Gamma', \theta', D' \rangle$ is in *solved form*, *i.e.*, D' contains only pairs of flexible terms. Huet showed (constructively) that in $\lambda_{\rightarrow}$ such D' are always unifiable [13, Section 3.3.3], and hence the existence of a pre-unifier implies the existence of a unifier. This is not always the case in λ_{Π}, but by maintaining a certain invariant on unification problems, we will show that the pre-unifiers constructed by our algorithm do always lead to unifiers.

3. Problems with Adapting Huet's Algorithm

Huet's algorithm relies on two important invariants that we are forced to abandon in HOU_{Π}. This section motivates our algorithm by explaining why we cannot maintain these invariants, and discussing how we handle the resulting difficulties.

In $\mathrm{HOU}_{\rightarrow}$, one can can require all disagreement pairs to be "homogenous", *i.e.*, relating terms of the same type. In λ_{Π}, since substitution affects types, two terms might be unifiable even if they have different types. For example, the disagreement pair $\langle \lambda x{:}\sigma.\, x, \lambda y{:}\tau.\, y \rangle$ is unified by the unifiers of the types σ and τ. Also, even if we do not start out with heterogenous disagreement pairs, they arise naturally in the presence of dependent types. Consider, for instance, a disagreement pair $\langle \mathsf{q}\, M\, N, \mathsf{q}\, M'\, N' \rangle$ of well-typed terms in the signature

$$\langle\ \mathsf{a}{:}\mathsf{Type}\ ,\ \mathsf{b}{:}\mathsf{a}{\rightarrow}\mathsf{Type}\ ,\ \mathsf{c}{:}\mathsf{Type}\ ,\ \mathsf{q}{:}\Pi x{:}\mathsf{a}.\ (\mathsf{b}\, x){\rightarrow}\mathsf{c}\ \rangle$$

Note that both terms have type c. As in the SIMPL phase of Huet's algorithm, we can replace this disagreement pair by the unification-equivalent[9] set of pairs $\{\, \langle M, M' \rangle, \langle N, N' \rangle \,\}$. Note, however, that N has type $(\mathsf{b}\, M)$, while N' has type $(\mathsf{b}\, M')$.

The second invariant we have to abandon is that disagreement pairs contain only well-typed terms. To see why, consider a disagreement set $D' \cup \{\langle v, \mathsf{c} \rangle\}$, where $v{:}\sigma$ in Γ and $\mathsf{c}{:}\tau$ in Σ. In Huet's algorithm, assuming the situation were allowed to arise, the treatment of this disagreement set would involve applying the substitution $[\, \mathsf{c}/v \,]$ to D' and continuing to work on the result. However, if σ and τ are different types, and v occurs (freely) in D' this substitution will construct ill-typed terms.[10] A similar phenomenon can happen in $\lambda_{\rightarrow}$ but is carefully avoided by Huet's algorithm. In the MATCH phase, if the flexible head has type $\alpha_1{\rightarrow}\cdots{\rightarrow}\alpha_m{\rightarrow}\alpha_0$, for a base type α_0, then among the m possible "projection substitutions", only the well-typed ones are tried. Because substitution does not affect types in $\lambda_{\rightarrow}$, this is just a matter of comparing type constants. When substitutions instantiate types though (as in λ_{Π}), we need to unify types, not just compare them.

These considerations might suggest that we can regain our invariants if, at certain points in the algorithm, we do type unification before continuing to unify terms. The fatal flaw in this idea is that we are doing *pre-unification*, not full unification, and after pre-unifying types, there may still be some remaining flexible-flexible pairs. Thus, some heterogeneity or ill-typedness may still be present.

---

[9] We use *unification-equivalent* to mean having the same set of unifiers.

[10] On the other hand, if v does not occur in D', for instance if $D' = \{\ \}$, we will have forgotten that we must unify σ and τ, and so the result would be wrong.

Our solution is to perform just enough of the type unification to insure that the substitution we are about to apply is approximately well-typed and cannot therefore destroy head normalizability. We do this by defining a partial function R_Γ that converts a pair of types, into a (unification-equivalent) set of disagreement pairs. If the function fails (is undefined) then the types are nonunifiable. R_Γ is defined by the following cases, and is undefined if no case applies. For brevity, we will write "$\lambda\Gamma.\ M$" for a context $\Gamma = [\, x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n \,]$ to mean $\lambda x_1{:}\alpha_1.\ \cdots \lambda x_n{:}\alpha_n.\ M$.

$$\begin{array}{rcl} \mathsf{R}_\Gamma(\mathsf{c},\mathsf{c}) & = & \{\,\} \\ \mathsf{R}_\Gamma(A\,M, A'\,M') & = & \mathsf{R}_\Gamma(A,A') \cup \{\langle \lambda\Gamma.\ M, \lambda\Gamma'.\ M'\rangle\} \\ \mathsf{R}_\Gamma(\Pi v{:}A.\ B, \Pi v'{:}A'.\ B') & = & \mathsf{R}_\Gamma(A,A') \cup \mathsf{R}_{(\Gamma, v:A)}(B, [\, v/v' \,]B') \end{array}$$

We will use R as follows: Before performing a substitution of a term of type τ to a variable of type σ, in a unification context Γ, we compute $\mathsf{R}_{[]}(\sigma,\tau)$. If this is undefined, we know that the substitution cannot lead to a (well-typed) unifier. If it yields a disagreement set $\hat{D}$, we perform the substitution (which we now know to be approximately well-typed), and add $\hat{D}$ to the resulting disagreement set, to account for any ill-typedness introduced by the substitution.

In order to prove unifiability of the disagreement sets in the pre-unifiers produced by our algorithm, we will need to keep track of which disagreement pairs account for which others. This relationship and the required approximate well-typedness conditions are embodied in the following invariant, on which our algorithm depends and which it maintains while transforming disagreement sets.

Definition 9 *A unification problem $\langle \Gamma, \theta_0, D\rangle$ is* acceptable, *which we will write as "$\mathcal{A}(Q)$", iff the following conditions hold:*

1. *Each of the disagreement pairs in D relates terms of the same approximate type (which are therefore approximately well-typed).*
2. *There is a strict partial order*[11] *"$\sqsubset$"on D such that for any disagreement pair $P \in D$, every unifier of $\{\, P' \in D \mid P' \sqsubset P \,\}$ instantiates P to a pair of terms of the same type (which are therefore well-typed).*
3. *Any well-typed unifier of D, when composed with θ_0, yields a well-typed substitution.*

It is important to note that the algorithm only maintains the *existence* of these strict partial orders, but never actually constructs them.

4. The Dependent Pre-unification Algorithm

This section presents an abstract algorithm for HOU_Π, based on collection of transformations from unification problems to sets of unification problems, which preserve sets of unifiers and maintain our invariant of acceptability. The goal of the transformations is to eventually construct unification problems in solved form.

We define the property required of our transformations as follows:

Definition 10 *For a unification problem Q and a set of unification problems $\mathcal{Q}$, we say that "$Q \lhd \mathcal{Q}$" iff when Q is acceptable, (a) so are all of the members of $\mathcal{Q}$, (b) the set of unifiers of*

[11] *i.e.*, a transitive, antisymmetric, *non*reflexive relation

$\mathcal{Q}$ is the union of the sets of unifiers of the members of $\mathcal{Q}$, and (c) the members of $\mathcal{Q}$ have no unifiers in common. More formally, $Q \lhd \mathcal{Q}$ iff $\mathcal{A}(Q)$ implies the following three conditions

$$\forall Q' \in \mathcal{Q}.\ \mathcal{A}(Q')$$

$$\mathcal{U}(Q) = \bigcup_{Q' \in \mathcal{Q}} \mathcal{U}(Q')$$

$$\forall Q', Q'' \in \mathcal{Q}.\ Q' \neq Q'' \Rightarrow \mathcal{U}(Q') \cap \mathcal{U}(Q'') = \{\ \}$$

Our algorithm is based on three transformations. Others may be added as optimizations, but these three suffice for completeness.[12] The transformations deal with unification problems containing a rigid-rigid, rigid-flexible, or flexible-rigid pairs. When no transformation applies, we have a unification problem in solved form. Collectively, these three transformations form a subrelation $\lhd_{\mathcal{U}}$ of $\lhd$.

4.1. The Transformations

For brevity, in all cases we assume for our unification problem Q that

$$\begin{array}{rcl} Q & = & \langle \Gamma, \theta_0, D \rangle \\ D & = & \{\ \langle M, M' \rangle\ \} \cup D' \end{array}$$

Without loss of generality, we can assume that M and M' are in LHNF (if not, convert them), so let

$$\begin{array}{rcl} M & = & \lambda x_1{:}\alpha_1.\ \cdots \lambda x_k{:}\alpha_k.\ a\, M_1 \cdots M_m \\ M' & = & \lambda x'_1{:}\alpha'_1.\ \cdots \lambda x'_k{:}\alpha'_k.\ a'\, M'_1 \cdots M'_{m'} \end{array}$$

(The invariant that M and M' have the same approximate type insures that they start with the same number k of λ's.)

The rigid-rigid transformation. Assume that M and M' are rigid, *i.e.*, a is a constant or some x_j, and a' is a constant or some $x'_{j'}$. If a and a' are the same modulo their binding contexts[13] then approximate well-typedness insures that $m = m'$, and we have

$$Q \ \lhd_{\mathcal{U}}\ \{\ \langle \Gamma, \theta_0, D' \cup \{\ \langle \hat{M}_i, \hat{M}'_i \rangle \mid 1 \leq i \leq m\ \} \rangle\ \}$$

where $\hat{M}_i = \lambda x_1{:}\alpha_1.\ \cdots \lambda x_k{:}\alpha_k.\ M_i$ and $\hat{M}'_i = \lambda x'_1{:}\alpha'_1.\ \cdots \lambda x'_k{:}\alpha'_k.\ M'_i$ for $1 \leq i \leq m$. Otherwise,

$$Q \ \lhd_{\mathcal{U}}\ \{\ \}$$

This transformation corresponds to one step of Huet's SIMPL phase. It preserves the set of unifiers because substitution does not affect the heading of M or M'. Thus rigid terms with distinct heads are nonunifiable, and rigid terms with the same head are unified exactly by the unifiers of their corresponding arguments. As for the invariant, (1) comes from M and M' being approximately well-typed, and (3) follows because the initial substitution is the same and the new disagreement set has the same unifiers as D. For (2), a new strict partial order can be

[12] Of particular value is the variable-term case, using a *rigid path* check [13]. Several others for HOU$_{\rightarrow}$ are found in [16].

[13] *i.e.*, they are the same constant, or $a = x_j$ and $a' = x'_j$ for some j

derived from an old one by replacing the old disagreement pair by the new ones, and adding $\langle \hat{M}_i, \hat{M}'_i \rangle \sqsubset \langle \hat{M}_j, \hat{M}'_j \rangle$ for $1 \leq i < j \leq m$. The reason for adding these is that each of the M_i can appear in the types of later M_j, and similarly for the M'_i.

As an example, consider the earlier case of $\{ \langle \mathsf{q}\, M\, N, \mathsf{q}\, M'\, N' \rangle \}$. The rigid-rigid transformation replaces this pair by $\{ \langle M, M' \rangle, \langle N, N' \rangle \}$. The disagreement pair $\langle M, M' \rangle$ accounts for the difference between the type $(\mathsf{b}\, M)$ of N and the type $(\mathsf{b}\, M')$ of N'. As another example, consider the disagreement set $\{\langle \lambda x{:}\sigma.\ x, \lambda y{:}\tau.\ y \rangle\} \cup D'$. The rigid-rigid transformation yields simply D', which is correct, because the invariant insures that the difference between σ and τ is already accounted for in D'.

The flexible-rigid transformation. Assume that M is flexible and M' is rigid. In this case, we will form a set of substitutions, each of which partially instantiates the flexible head a in such a way that completely determines its head but leaves its arguments completely undetermined. These are the imitation and projection substitutions generated by Huet's MATCH phase.

These substitutions are motivated by consideration of the LHNF of θa for any unifier θ of M and M'. Let the type of a be $\Pi y_1{:}\tau_1.\ \cdots \Pi y_m{:}\tau_m.\ \tau_0$, where τ_0 is not a Π type. Then for any well-typed substitution θ, any LHNF of θa has the form

$$\lambda y_1{:}\hat{\tau}_1.\ \cdots \lambda y_m{:}\hat{\tau}_m.\ b\, N_1 \cdots N_n$$

for some variable or constant b. Since the head of θM has to be the rigid head a' of M', the only possibilities for b are (1) a', if a' is a constant, or (2) some y_i, for $1 \leq i \leq m$. For each such b, we can capture this restriction on θa by equivalently saying that θ has the form $\hat{\theta} \circ [\, N_b/a \,]$, for some $\hat{\theta}$, where

$$N_b \;=\; \lambda y_1{:}\tau_1.\ \cdots \lambda y_m{:}\tau_m.\ b\,(v_{b1}\, y_1 \cdots y_m) \cdots (v_{bl_b}\, y_1 \cdots y_m)$$

Here $v_{b1}, \ldots, v_{bl_b}$ are distinct variables not in Γ or among the y_j.[14] (See [4, Section 3.2.2] for details.) Letting B be the set of possible b's described above, one can then show that[15]

$$\mathcal{U}(Q) \;=\; \bigcup_{b \in B} \mathcal{U}([\, N_b/a \,]Q)$$

where by an application "$\theta Q'$" of a substitution $\theta \in \Theta^{\Gamma''}_{\Gamma'}$ to a unification problem $Q' = \langle \Gamma', \theta'_0, D' \rangle$, we mean the unification problem

$$\langle \Gamma'', \theta \circ \theta'_0, \{\, \langle \theta M, \theta M' \rangle \mid \langle M, M' \rangle \in D' \,\} \rangle$$

Now, to reestablish the invariant, we must add disagreement pairs to account for any difference between the types of a and N_b. Let σ_a and σ_{N_b} be the types in Γ of a and N_b, and, for each $b \in B$ such that $\mathsf{R}_{[]}(\sigma_a, \sigma_{N_b})$ is defined, let

$$Q_b \;=\; (\{[\, N_b/a \,]\langle \Gamma, \theta_0, D \rangle\}) \cup \mathsf{R}_{[]}(\sigma_a, \sigma_{N_b})$$

[14] In order to account for the v_{b_i}, we assume a nonstandard definition of substitutions and their composition that causes temporary variables to be eliminated appropriately [4, Section 2.6].

[15] The key step is that $\mathcal{U}(\langle \Gamma, \theta_0, D \rangle)$ can be re-expressed as

$$\bigcup_{b \in B} \{\, (\hat{\theta} \circ [\, N_b/a \,]) \circ \theta_0 \mid \hat{\theta} \in \Theta_{\Gamma_b} \wedge \forall \langle M, M' \rangle \in D.\ (\hat{\theta} \circ [\, N_b/a \,])M =_{\beta\eta} (\hat{\theta} \circ [\, N_b/a \,])M' \,\}$$

where Γ_b is the result of removing a and adding $v_{b1}, \ldots, v_{bl_b}$ to Γ. Then by using associativity of substitution composition, and contracting the definition of $\mathcal{U}$, the result follows.

where by the union of a unification problem $\langle \Gamma', \theta_0', D' \rangle$ with a disagreement set D'', we mean $\langle \Gamma', \theta_0', D' \cup D'' \rangle$. Then we have

$$Q \quad \lhd_U \quad \{ Q_b \mid b \in B \text{ and } \mathsf{R}_{[]}(\sigma_a, \sigma_{N_b}) \text{ is defined} \}$$

Note that this addition of $\mathsf{R}_{[]}(\sigma_a, \sigma_{N_b})$ is the only place where our set of transformations differs from Huet's. For simply typed terms, these $\mathsf{R}_{[]}(\sigma_a, \sigma_{N_b})$ (when defined) are always empty, so our algorithm does no more work than Huet's.

It is important to note that the generated unification problems have no solutions in common. This is because, for each b, any unifier of Q_b is an instance of $[\, N_b/a \,] \circ \theta_0$, and since each N_b has a different rigid head (namely b), no two of the $[\, N_b/a \,] \circ \theta_0$ can have any instances in common. This property is what guarantees minimality (see Theorem 13).

A new strict partial order can be derived from an old one by replacing each disagreement pair by its newly instantiated version, and by adding $P \sqsubset P'$ for each $P \in \mathsf{R}_{[]}(\sigma_a, \sigma_{N_b})$ and $P' \in [\, N_b/a \,]D$, since we added $\mathsf{R}_{[]}(\sigma_a, \sigma_{N_b})$ to account for any introduced ill-typedness.

The rigid-flexible transformation. The rigid-flexible case can be handled simply by reflecting it into the flexible-rigid case: If M is rigid and M' is flexible then

$$Q \quad \lhd_U \quad \{\langle \Gamma, \theta_0, \{ \langle M', M \rangle \} \cup D' \rangle\}$$

4.2. The Algorithm

Now that we have presented the three transformations, together defining the relation $\lhd_U$, we will describe a search process that operates on a set of unification problems and enumerates a set of pre-unifiers. Informally, the process goes as follows: If there are no unification problems left, stop. Otherwise, choose a unification problem to work on next. If it is in solved form, add it the the set of solutions. Otherwise, apply one of the transformations, in some way, to replace the unification problem by a finite set of new unification problems. Then continue.

Note that two kinds of choices are made in this process. First, there is the choice of which unification problem to work on next, and second, there is the choice of which transformation to apply and how to apply it. It turns out that the second kind of choice may be made completely arbitrarily, but, in order to have completeness, the first kind must be done in a fair way.[16] Huet formulated this difference by constructing "matching trees", in which the nodes are disagreement sets and the edges are substitutions, and then showed that all matching trees are complete. His pre-unifiers are constructed by composing substitutions along edges that form a path from the original disagreement set to one in solved form. In our formulation, these composed substitutions are part of the unification problem.

Definition 11 *For a relation ρ between unification problems and sets of unification problems, and a unification problem Q, a* ρ search tree from Q *is a tree T of unification problems such that*

- *The root of T is Q.*
- *For every node Q' in T, the set of children of Q' in T is either empty if Q' is in solved form, or is some $\mathcal{Q}$ satisfying $Q' \rho \mathcal{Q}$ if Q' is not in solved form.*

[16] In implementation terms, this means that we can use *e.g.*, breadth-first search or depth-first search with iterative deepening, but not simple depth-first search.

Definition 12 *For a relation ρ between unification problems and sets of unification problems, we define the relation ρ^{**} as follows: $Q\rho^{**}\mathcal{Q}$ iff there is some ρ search tree from Q whose set of solved nodes is $\mathcal{Q}$.*

We can then show the following

Theorem 13 *Let Q be a unification problem such that $\mathcal{A}(Q)$, and let $\mathcal{Q}$ be any set of unification problems such that $Q \lhd_{\mathcal{U}}^{**} \mathcal{Q}$. Then*

1. *$\mathcal{A}(Q')$ for each $Q' \in \mathcal{Q}$.*
2. *Every $Q' \in \mathcal{Q}$ is a pre-unifier of Q, i.e., it is in solved form and $\mathcal{U}(Q') \subseteq \mathcal{U}(Q)$.*
3. *$\mathcal{Q}$ is minimal, i.e., for any two distinct members Q', Q'' of $\mathcal{Q}$, $\mathcal{U}(Q') \cap \mathcal{U}(Q'') = \{\ \}$.*
4. *$\mathcal{Q}$ is complete, i.e., for any unifier θ of Q, there is a $Q' \in \mathcal{Q}$ such that $\theta \in \mathcal{U}(Q')$.*

Proof sketch:

1. Each transformation maintains the invariant for each constructed unification problem.
2. The transformations do not introduce new unifiers, and $\mathcal{Q}$ contains only solved form unification problems.
3. As noted in the discussion of the flexible-rigid transformation, when the search for pre-unifiers branches, the new unification problems have no unifiers in common.
4. Because our invariant insures head normalizability, the completeness proof goes much as in [13], and has two main parts: (a) For a given unification problem Q, there can be only finitely many successive applications of the rigid-rigid and rigid-flexible transformations. (b) For any unification problem $\langle \Gamma, \theta_0, D \rangle$ to which the flexible-rigid transformation applies, and any unifier θ of D, there is flexible-rigid-successor $\langle \Gamma', \theta_0', D' \rangle$ of $\langle \Gamma, \theta_0, D \rangle$ and a unifier θ' of D' such that θ' has strictly lower complexity than θ, where complexity is defined in terms of sizes of the long $\beta\eta$ normal forms involved. Another consideration, not required in HOU$_{\rightarrow}$, is that R_Γ always terminates.

□

Given a pair of terms M and M' to unify, we can satisfy the invariant initially in either of two ways. The first is to simply check that M and M' are well-typed and have the same type. (This is possible because type checking is decidable [10].) This method is simple but does not allow for terms that will become well-typed or disagreement pairs that will become homogeneous after substitution.[17] The second method is much more flexible. Instead of type-checking the terms, we perform only approximate type-checking, and at the same time, construct a disagreement set whose unifiers (if any) instantiate the terms to well-typed terms of the same type. This process is defined in Section 6.

[17] Ill-typedness and heterogeneity can still arise during unification though.

5. Unifiability of Flexible-flexible Disagreement Sets

The value of pre-unification in $\lambda_{\rightarrow}$ is that solved disagreement sets (ones containing only flexible-flexible pairs) are always unifiable, and so pre-unifiability implies unifiability [13]. This is not true in general for λ_{Π}, but it is true of solved sets satisfying our invariant. By making vital use the strict partial order in the definition of $\mathcal{A}$, we can generalize Huet's constructive proof of this fact to λ_{Π}. For the simply typed subset of λ_{Π}, the substitution that we use specializes to Huet's.

Definition 14 *For a context* Γ, *the* canonical unifier θ^{Q}_{Γ} *over* Γ *is the substitution assigning to each variable* $v{:}\Pi x_1{:}\sigma_1.\cdots\Pi x_m{:}\sigma_m.\,\mathsf{c}\,Q_1\cdots Q_n$ *in* Γ, *the term* $\lambda x_1{:}\sigma_1.\cdots\lambda x_m{:}\sigma_m.\,h_{\mathsf{c}}\,Q_1\cdots Q_n$, *where* h_{c} *is a variable of type* $\Pi y_1{:}\tau_1.\cdots\Pi y_n{:}\tau_n.\,\mathsf{c}\,y_1\cdots y_n$.

Theorem 15 *If* Q *is a acceptable unification problem in solved form with unification context* Γ, *then* $\theta^{Q}_{\Gamma}\in\mathcal{U}(Q)$.

Proof: Let $\sqsubset$ be the strict partial order imposed on disagreement sets by our invariant. Since disagreement sets are always finite, $\sqsubset$ is a well founded ordering, and thus we will give an inductive argument. Let $\langle M, M'\rangle$ be an arbitrary member or our disagreement set for

$$\begin{array}{rcl} M & = & \lambda z_1{:}\alpha_1.\cdots\lambda z_k{:}\alpha_k.\,v\,M_1\cdots M_m \\ M' & = & \lambda z'_1{:}\alpha'_1.\cdots\lambda z'_k{:}\alpha'_k.\,v'\,M'_1\cdots M'_{m'} \end{array}$$

where v and v' are variables with types

$$\begin{array}{rcl} v & : & \Pi x_1{:}\sigma_1.\cdots\Pi x_m{:}\sigma_m.\,\mathsf{c}\,Q_1\cdots Q_n \\ v' & : & \Pi x'_1{:}\sigma'_1.\cdots\Pi x'_{m'}{:}\sigma'_{m'}.\,\mathsf{c}\,Q'_1\cdots Q'_n \end{array}$$

and

$$\mathsf{c} \;:\; \Pi y_1{:}\tau_1.\cdots\Pi y_n{:}\tau_n.\,\mathsf{Type}$$

The reason that the types of both v and v' must involve the same type constant c, is that our invariant insures that M and M' have the same approximate type. Now, for $1\le j\le n$, let

$$\begin{array}{rcl} N_j & = & \lambda z_1{:}\alpha_1.\cdots\lambda z_k{:}\alpha_k.\,[\,(\theta^{Q}_{\Gamma}M_1)/x_1,\ldots,(\theta^{Q}_{\Gamma}M_m)/x_m\,]Q_j \\ N'_j & = & \lambda z'_1{:}\alpha'_1.\cdots\lambda z'_k{:}\alpha'_k.\,[\,(\theta^{Q}_{\Gamma}M'_1)/x_1,\ldots,(\theta^{Q}_{\Gamma}M'_m)/x_m\,]Q'_j \end{array}$$

Then, for some choice of $\hat{\alpha}_1,\ldots,\hat{\alpha}_k$ and $\hat{\alpha}_1,\ldots,\hat{\alpha}_k$, we have

$$\begin{array}{rcl} \theta^{Q}_{\Gamma}M & = & \lambda z_1{:}\hat{\alpha}_1.\cdots\lambda z_k{:}\hat{\alpha}_k.\,h_{\mathsf{c}}\,(N_1\,z_1\cdots z_k)\cdots(N_n\,z_1\cdots z_k) \\ \theta^{Q}_{\Gamma}M' & = & \lambda z'_1{:}\hat{\alpha}'_1.\cdots\lambda z'_k{:}\hat{\alpha}'_k.\,h_{\mathsf{c}}\,(N'_1\,z'_1\cdots z'_k)\cdots(N'_n\,z'_1\cdots z'_k) \end{array}$$

By induction, assume that θ^{Q}_{Γ} unifies all disagreement pairs below $\langle M, M'\rangle$ in the ordering. Thus, by our invariant, $\theta^{Q}_{\Gamma}M$ and $\theta^{Q}_{\Gamma}M'$ are well-typed terms of the same type, so

$$\begin{array}{rcl} & & \Pi z_1{:}\hat{\alpha}_1.\cdots\Pi z_k{:}\hat{\alpha}_k.\,\mathsf{c}\,(N_1\,z_1\cdots z_k)\cdots(N_n\,z_1\cdots z_k) \\ & = & \Pi z'_1{:}\hat{\alpha}'_1.\cdots\Pi z'_k{:}\hat{\alpha}'_k.\,\mathsf{c}\,(N'_1\,z'_1\cdots z'_k)\cdots(N'_n\,z'_1\cdots z'_k) \end{array}$$

It then follows that $\theta^{Q}_{\Gamma}M =_{\beta\eta}\theta^{Q}_{\Gamma}M'$. □

6. Automatic Term Inference

It is well known that first-order unification provides for type inference in $\lambda_{\rightarrow}$ with type variables and in similar languages [17]. Recently, it has been shown that $\text{HOU}_{\rightarrow}$ is the key ingredient for the corresponding problem in the ω-order polymorphic λ-calculus [21]. In λ_{Π} there is a new problem of interest, namely *term inference*, which requires HOU_{Π}. This problem has two important applications. One is making our unification algorithm more widely applicable, by initially establishing the required invariant, as mentioned at the end of Section 4, and made precise below. The other is to provide automatic object-language type inference. This section gives a very simple algorithm for λ_{Π} term inference, using HOU_{Π}.

We will construct the term inference algorithm using two partial functions. The first one, Mi_{Γ}, for a given context Γ and signature Σ (the latter of which we will leave implicit) takes a term M and, if defined, yields a pair consisting of a type A and a disagreement set D. $\text{Mi}_{\Gamma}(M)$ is undefined when M is not even approximately well-typed. Otherwise, for every unifier θ of D, it is the case that $\Gamma \vdash_{\Sigma} \theta M \in \theta A$. The second partial function Ai_{Γ}, takes a type A and, if defined, yields a pair consisting of a kind K and a disagreement set D. If $\text{Ai}_{\Gamma}(A)$ is undefined then A has no well-kinded instance. Otherwise, for every unifier θ of D, we have $\Gamma \vdash_{\Sigma} \theta A \in \theta K$. The structure of these definitions is determined by the typing rules in Section 2, and uses the partial function R defined in Section 3.

$$
\begin{aligned}
\text{Mi}_{\Gamma}(v) &= \langle A, \{\,\} \rangle \text{ where } v{:}A \text{ in } \Gamma \\
\text{Mi}_{\Gamma}(\mathsf{c}) &= \langle A, \{\,\} \rangle \text{ where } \mathsf{c}{:}A \text{ in } \Sigma \\
\text{Mi}_{\Gamma}(M\,N) &= \langle [\,N/v\,]B, D \cup D' \cup \text{R}_{[]}(A, A') \rangle \\
&\qquad \text{where } \left\{ \begin{array}{l} \text{Mi}_{\Gamma}(M) = \langle (\Pi v{:}A.\ B), D \rangle \text{[18]} \\ \text{Mi}_{\Gamma}(N) = \langle A', D' \rangle \end{array} \right. \\
\text{Mi}_{\Gamma}(\lambda v{:}A.\ M) &= \langle \Pi v{:}A.\ B, D \cup D' \rangle \\
&\qquad \text{where } \left\{ \begin{array}{l} \text{Ai}_{\Gamma}(A) = \langle \text{Type}, D \rangle \\ \text{Mi}_{\Gamma, v:A}(M) = \langle B, D' \rangle \end{array} \right.
\end{aligned}
$$

$$
\begin{aligned}
\text{Ai}_{\Gamma}(\mathsf{c}) &= \langle K, \{\,\} \rangle \text{ where } \mathsf{c}{:}K \text{ in } \Sigma \\
\text{Ai}_{\Gamma}(\Pi v{:}A.\ B) &= \langle \text{Type}, D \cup D' \rangle \\
&\qquad \text{where } \left\{ \begin{array}{l} \text{Ai}_{\Gamma}(A) = \langle \text{Type}, D \rangle \\ \text{Ai}_{\Gamma, v:A}(B) = \langle \text{Type}, D' \rangle \end{array} \right. \\
\text{Ai}_{\Gamma}(A\,M) &= \langle [\,M/v\,]K, D \cup D' \cup \text{R}_{[]}(B, B') \rangle \\
&\qquad \text{where } \left\{ \begin{array}{l} \text{Ai}_{\Gamma}(A) = \langle (\Pi v{:}B.\ K), D \rangle \\ \text{Mi}_{\Gamma}(M) = \langle B', D' \rangle \end{array} \right.
\end{aligned}
$$

Given a term M in a context Γ we do type-checking/term inference as follows. If $\text{Mi}_{\Gamma}(M)$ is undefined, then M is not approximately well-typed, and hence it has no well-typed instance, so we indicate failure. Otherwise, let $\langle A, D \rangle = \text{Mi}_{\Gamma}(M)$, and let $\mathcal{Q}$ be such that $\langle \Gamma, \theta^{id}_{\Gamma}, D \rangle \lhd^{**}_{\mathcal{U}} \mathcal{Q}$, where θ^{id}_{Γ} is the identity substitution over Γ. If $\mathcal{Q}$ is empty, then M has no well-typed instance.

[18] The intended interpretation is that if the type part of $\text{Mi}_{\Gamma}(M)$ is not a Π type, then $\text{Mi}_{\Gamma}(M\,N)$ is undefined.

Otherwise, for each $\langle \Gamma', \theta', D' \rangle \in \mathcal{Q}$, we return the instantiated term $\theta' M$ together with the "constraint" D'.[19]

If on the other hand we have two terms M and M' to be unified in a context Γ, we can proceed as follows: If $\mathsf{Mi}_\Gamma(M)$ or $\mathsf{Mi}_\Gamma(M')$ is undefined, then M or M' is *not* approximately well-typed, so we indicate typing error. Otherwise, let $\langle A, D \rangle = \mathsf{Mi}_\Gamma(M)$ and $\langle A', D' \rangle = \mathsf{Mi}_\Gamma(M')$. Then if $\mathsf{R}_{[]}(A, A')$ is undefined, we indicate typing mismatch. Otherwise, let $D'' = \mathsf{R}_{[]}(A, A')$. Then apply the pre-unification algorithm to the unification problem $\langle \Gamma, \theta^{id}_\Gamma, \{\langle M, M' \rangle \cup D \cup D' \cup D''\} \rangle$.

The reason that $\lambda\Pi$ term inference often gives object-language type inference is that we can use $\lambda\Pi$ terms to encode object-language types and then construct object-language terms using constants whose (dependent) types record the object-language's typing system. One example of this is in the encoding of higher-order logic, given in [10]. Another is a simple first-order typed expression language [4, Section 7.3.3].

7. Related Work

Our algorithm is clearly influenced by the ideas underlying Huet's. A related transformational approach is Snyder and Gallier's for $\mathrm{HOU}_{\to}$ and equational unification [27,26]. One minor difference is that, rather than carrying along a substitution as part of their unification problems, they represent these substitutions as a "solved" part of their disagreement sets. A more important difference is that their transformations map a unification problem to a single unification problem, rather than a set of unification problems. However, doing so prevents an important distinction between the two kinds of "nondeterminism" in the algorithm, namely between the multiplicity of pre-unifiers in a complete set, and the multiplicity of ways in which transformation rules can be chosen and applied (resulting in different complete sets of preunifiers). We do not see how one could construct *minimal* sets of pre-unifiers using their approach.

Recently, Pym [24] reported an independently developed algorithm for HOU_Π.

8. Conclusions and Further Work

In this paper, we have presented an algorithm for HOU_Π, *i.e.*, higher-order (pre-)unification in a typed λ-calculus with dependent function types. This algorithm makes possible many valuable extensions to current applications of $\mathrm{HOU}_{\to}$, as well as mechanized theorem proving in object-logics encoded as in the Edinburgh Logical Framework (LF). We also presented a particularly useful application of HOU_Π to perform $\lambda\Pi$ *term inference*. This algorithm makes HOU_Π more widely applicable and allows for automatic type inference in a variety of object-languages.

Our algorithm has good efficiency properties. For simply typed examples, it does the same work as Huet's algorithm. Thus the additional power of the algorithm is only payed for where it is used.

A critical property of pre-unification in $\lambda_{\to}$ is that pre-unifiability is a sufficient condition for unifiability. Although this is not generally true in $\lambda\Pi$ under the relaxed typing conditions that we are forced to allow, we showed that the pre-unifiers constructed by our algorithm do indeed lead to unifiers.

[19] Depending on the application, if $\mathcal{Q}$ has more than one element, and/or if D' is nonempty for some $\langle \Gamma', \theta', D' \rangle \in \mathcal{Q}$, it may be appropriate to request a user to provide a more constrained term.

We have implemented a prototype version of an extension of our HOU_Π algorithm, which also handles type variables. Although the treatment of type variables is incomplete, it is quite useful in practice. We also plan to add treatment of the dependent version of Cartesian product types (often called "strong sum" or "Σ" types).[20] This implementation will form the basis of (a) a generalization of the programming language λProlog [18] to λ_Π, to serve as a convenient implementation language for applications of HOU_Π, and (b) the new language Elf for logic definition and verified meta-programming [20].

An area for future work is to develop a complete treatment of type variables, and if this succeeds, explicit polymorphism as in the second- or ω-order polymorphic λ-calculus [6,25].

9. Acknowledgment

I am very grateful to Frank Pfenning for originally suggesting the problem in λ_Π, and for several very helpful discussions yielding many useful ideas, in particular the idea of approximate well-typedness.

References

[1] Peter B. Andrews, Dale Miller, Eve Cohen, and Frank Pfenning. Automating higher-order logic. *Contemporary Mathematics*, 29:169–192, August 1984.

[2] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. *HOPE: an Experimental Applicative Language.* Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, Edinburgh, U.K., 1981.

[3] Michael R. Donat and Lincoln A. Wallen. Learning and applying generalised solutions using higher order resolution. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 41–60, Springer-Verlag LNCS 310, Berlin, May 1988.

[4] Conal Elliott. *Some Extensions and Applications of Higher-order Unification: A Thesis Proposal.* Ergo Report 88–061, Carnegie Mellon University, Pittsburgh, June 1988. Thesis to appear June 1989.

[5] Amy Felty and Dale A. Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 61–80, Springer-Verlag LNCS 310, Berlin, May 1988.

[6] Jean-Yves Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, North-Holland Publishing Co., Amsterdam, London, 1971.

[7] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[8] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 942–959, MIT Press, Cambridge, Massachusetts, August 1988.

[9] Robert Harper. *Standard ML.* Technical Report ECS-LFCS-86-2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[20] Preliminary work on this appears in [4].

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, IEEE, June 1987.

[11] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus*. Cambridge University Press, 1986.

[12] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

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

[14] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[15] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Journal of Pure and Applied Logic*, 1988. Submitted.

[16] Dale A. Miller. Unification under mixed prefixes. 1987. Unpublished manuscript.

[17] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[18] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, MIT Press, Cambridge, Massachusetts, August 1988.

[19] Lawrence C. Paulson. *The Representation of Logics in Higher-Order Logic*. Technical Report 113, University of Cambridge, Cambridge, England, August 1987.

[20] Frank Pfenning. *Elf: A Language for Logic Definition and Verified Meta-Programming*. Ergo Report 88–067, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 1988.

[21] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, ACM Press, July 1988.

[22] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, ACM Press, June 1988. Available as Ergo Report 88–036.

[23] Garrel Pottinger. Proof of the normalization and Church-Rosser theorems for the typed λ-calculus. *Notre Dame Journal of Formal Logic*, 19(3):445–451, July 1978.

[24] David Pym. A unification algorithm for the logical framework. November 1988. Laboratory for Foundations of Computer Science, University of Edinburgh. To appear as LFCS report.

[25] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, Springer-Verlag LNCS 19, New York, 1974.

[26] Wayne Snyder. *Complete Sets of Transformations for General Unification*. PhD thesis, University of Pennsylvania, 1988.

[27] Wayne Snyder and Jean H. Gallier. Higher-order unification revisited: complete sets of transformations. *Journal of Symbolic Computation*, 1988. To appear in the special issue on unification.

[28] David A. Turner. Miranda: a non-strict functional lanugage with polymorphic types. In *Functional Programming Languages and Computer Architecture*, Springer-Verlag, Berlin, September 1985.