Unification and Anti-Unification in the Calculus of Constructions

Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Internet: fp@cs.cmu.edu

Abstract

We present algorithms for unification and anti-unification in the Calculus of Constructions, where occurrences of free variables (the variables subject to instantiation) are restricted to higher-order patterns*, a notion investigated for the simply-typed λ-calculus by Miller. Most general unifiers and least common anti-instances are shown to exist and are unique up to a simple equivalence. The unification algorithm is used for logic program execution and type and term reconstruction in the current implementation of Elf and has shown itself to be practical. The main application of the anti-unification algorithm we have in mind is that of proof generalization.*

1 Introduction

Higher-order logic with an embedded simply-typed λ-calculus has been used as the basis for a number of theorem provers (for example [1, 19]) and the programming language λProlog [16]. Central to these systems is an implementation of Huet's pre-unification algorithm for the simply-typed λ-calculus [12] which has shown itself to be very useful in practice, despite the undecidability of higher-order unification [8]. However, the non-determinism, more so than the undecidability, presents some problems with full higher-order unification as the basis for proof development environments and logic programming languages. This has lead to search for a restriction on the occurrences of variables in simply-typed λ-terms such that the unification problem becomes deterministic and decidable. Such a class was discovered by Miller [15] and applied by Nipkow [17] to higher-order rewriting and by the author [21] to the simplification of constraints and type reconstruction in Elf. Though our class is more general, since it is situated in the Calculus of Constructions, we follow Nipkow and call such restricted terms here *higher-order patterns*. This concept is developed in Section 3.

Higher-order logic lacks an internal notion of proof. Thus, in circumstances where proofs are important as objects of study (say, for program extraction, proof transformation, or proof generalization, to give but three examples), benefits may be derived from using a richer type theory, such as offered by the LF Logical Framework [10] or the more general Calculus of Constructions [3]. A number of program development and theorem proving environments have been constructed on the basis of such type theories (see, for example, [6, 18, 20]).

In order to give sophisticated assistance for proof development and management in these frameworks, pre-unification algorithms for LF have been developed independently by Elliott [7] and Pym [23]. The drawbacks of non-determinism and undecidability were inherited by these algorithms from the simply-typed case.

A combination of the ideas of Miller and Elliott for a deterministic, though incomplete algorithm for the LF Logical Framework is presented by the author in [21]. This forms the basis for the programming language Elf and has shown itself very effective in practice. In Section 4 we present a generalization of this algorithm to the full Calculus of Constructions.

One of the fundamental operations on proofs is that of *generalization*: we abstract away from the particulars of a proof to obtain a more general one. Generalization from *one* proof (or explanation-based generalization, in the terminology of the Artificial Intelligence literature) has been investigated by Hagiya [9] in the setting of type theory, and by Dietzen and the author in setting of λProlog [4]. Our extension to the Calculus of Constructions yields an algorithm for finding the least general generalization (or anti-unifier) of two proofs, which may be a more general proof schema. The restriction to patterns in this context is a restriction to the language in which we are allowed to express

the generalization. Without this restriction, minimal generalizations are usually not interesting, since they can be instantiated to little else aside from the original two terms. Moreover, many incomparable (in the substitution ordering) generalizations may exist.

In this paper we outline a comprehensive theory of unification and anti-unification for patterns in the Calculus of Constructions. In this calculus, greatest common instances (most general unifiers) and least common anti-instances (least general anti-unifiers) exist and are unique up to a simple equivalence relation. We present algorithms to effectively compute these. In the resulting order structure (under the instantiation ordering), every pair of patterns which has a lower bound, has a greatest lower bound, and every pair of patterns which has an upper bound, has a greatest upper bound. However, unlike in the first-order situation, not every pair of patterns has an upper bound.

2 The Calculus of Constructions

We began the work described herein in the context of the LF Logical Framework [10] and we still consider Elf [20, 21] as the primary vehicle for the applications of the results presented in this paper. But they may also be useful in the context of a Mathematical Vernacular [5] or a program development environment based on the Calculus of Constructions (CC) [18, 6]. For this reason, and also because it streamlines the presentation, we chose the Calculus of Constructions as the formalism for the presentation of the ideas.

There are currently a number of closely related formulations of the Calculus of Constructions. The choice of formalization is mainly a matter of economy of presentation of the inference rules and the various algorithms. The formulation we use here is taken from [11], but closely related formulations appear throughout the literature.

We use M, N for terms in general and u, v and x, y, z for variables, where the occurrences of u in $[u{:}M]\,N$ and $\{u{:}M\}\,N$ are binding occurrences. We also assume an fixed signature Σ assigning types to constants, which are denoted by c. We use h as a notation for either a variable or a constant. We have

$$\begin{array}{rcll} \kappa & ::= & \mathsf{Prop} \mid \mathsf{Type} & \textit{kinds} \\ M & ::= & c \mid x \mid \kappa \mid \{x{:}M\}\,N & \\ & & \mid [x{:}M]\,N \mid (M\,N) & \textit{terms} \\ \Gamma & ::= & \cdot \mid \Gamma\,[x{:}M] & \textit{contexts} \\ \Sigma & ::= & \cdot \mid \Sigma\,[c{:}M] & \textit{signatures} \end{array}$$

Following [3] we call $\{x{:}M\}\,N$ a *product*; $[x{:}M]\,N$ is λ-abstraction. Unfortunately this terminology does not exactly match up with what is used in the LF logical framework, where Type is used instead of Prop, and Type would be called Kind (if it were made explicit). We write $\Delta \dot\cup \Gamma$ for the result of appending the variable disjoint contexts Δ and Γ. Given a context $\Gamma = [u_1{:}A_1] \ldots [u_p{:}A_p]$ and a term M we write $(\lambda\Gamma)\,M$ for the term $[u_1{:}A_1] \ldots [u_p{:}A_p]\,M$. We explicitly include constants in our formulation for a purely technical reason: we need to distinguish those identifiers which may appear in instantiation terms (constants) from those which may not (bound variables). Miller's *mixed prefixes* offer a slightly more general alternative, but it would further complicate the presentation.

We allow $A \to B$ as an abbreviation for $\{x{:}A\}\,B$ if x does not occur free in B. We sometimes omit the parentheses surrounding applications in which case application is written simply as juxtaposition and associates to the left. Juxtaposition binds tighter than "$\to$", which associates to the right. Abstraction and product also associate to the right and bind less tightly than "$\to$". The equality in the metalanguage is "$=$".

The inference system (see Figure 1) defines two main judgments: "$\Delta \vdash \Gamma$ valid" means that Γ is a valid context in the valid context Δ, and "$\Gamma \vdash M : A$" means that M is a valid term of type A in the valid context Γ. We use $A, B, \ldots$ for *types*, that is, terms of type Prop or Type. We say that A and B are *consistent* if they are both of type Prop or both of type Type. In general, we consider α-convertible terms to be identical, and $[N/x]M$ is the notation for substituting N for x in M, renaming bound variable names as necessary to avoid name clashes. We assume in the inference system that Σ and Γ assign unique types to constants and variables, respectively. We write $\mathrm{dom}(\Sigma)$ and $\mathrm{dom}(\Gamma)$ for the set of constants and variables declared in Σ and Γ, and $\Sigma(c)$ and $\Gamma(x)$ for the types assigned to c and x in Σ and Γ, respectively. We also assume that the fixed signature Σ is valid (in the obvious sense), but omit the corresponding judgment and rules.

We consider β and η-conversion ($\cong$) in the "full" form (see [3, Page 102]). η-conversion plays an important role, because the various algorithms we give are defined on *canonical forms* (or long $\beta\eta$-normal forms). This notion is defined formally through the inference system in Figure 2. The main judgment is $\Gamma \vdash M \Rightarrow A$ (read: M is canonical of type A in context Γ). To our knowledge, it has not been shown that every term in the Calculus of Constructions is equivalent to a unique canonical form—we will take this as a working hypothesis as in [5]. For the restriction of this system to the LF type theory, this has recently been proved independently by Coquand [2] and Salvesen [25]. It is possible

$$\frac{}{\cdot \ \text{valid}} \qquad \frac{\Gamma \vdash A : \kappa}{\Gamma[x{:}A] \ \text{valid}}$$

$$\frac{\Sigma(c) = A \qquad \Gamma \ \text{valid}}{\Gamma \vdash c : A} \qquad \frac{\Gamma(x) = A \qquad \Gamma \ \text{valid}}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \ \text{valid}}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}} \qquad \frac{\Gamma[x{:}A] \vdash B : \kappa}{\Gamma \vdash \{x{:}A\}\, B : \kappa}$$

$$\frac{\Gamma[x{:}A] \vdash M : B}{\Gamma \vdash [x{:}A]\, M : \{x{:}A\}\, B} \qquad \frac{\Gamma \vdash M : \{x{:}A\}\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : [N/x]B}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \cong B \qquad \Gamma \vdash B : \kappa}{\Gamma \vdash M : B}$$

Figure 1: The Calculus of Constructions

that the work here may be adapted to a language in which η-conversion is not assumed, but the additional complexity does not seem to be warranted, as in practice, the system with η-conversion is preferable.

The calculus shares the basic properties of the LF type theory such as strong normalization and decidability of type-checking. We tacitly use these properties in the development below. A term is *atomic* if it has the form Prop or $h\, M_1 \ldots M_m$ for a constant or variable h. We use the letter C to range over atomic terms.

We define the notion of a *canonical form* through the inference system in Figure 2. The main judgment is $\Gamma \vdash M \Rightarrow A$ (read: M is canonical of type A in context Γ).

3 Higher-Order Patterns

In the theory of first-order unification and anti-unification [22, 24, 13, 14], the authors construct a semi-lattice of terms with free variables, ordered under instantiation of these variables. Here we are dealing with a typed language, so we need to consider terms in a context which assigns types to the free variables. Similarly, substitutions will have to be typed.

In this section, we give the basic definitions which allow us to present the algorithms for unification and anti-unification.

A *valid cterm* is a pair $\langle \Delta, M \rangle$ such that $\Delta \vdash M \Rightarrow A$ (which implies that $\Delta \vdash M : A$ and M in canonical form). We consider α-convertible cterms as identical, where the notion of α-conversion includes renaming variables in Δ and applying appropriate renaming in the tail of Δ and M. Note that $\langle \Delta, \mathsf{Type} \rangle$ is not a valid cterm.

A central notion in the development of both, unification and anti-unification, is the notion of a partial permutation. Given natural numbers n and p, a *partial permutation* ϕ from n into p is an injective mapping from $\{1, \ldots, n\}$ into $\{1, \ldots, p\}$, that is, $\phi(i) = \phi(i')$ implies $i = i'$.

For the remainder of the paper, we let Δ range over contexts giving types to variables which may be instantiated by substitutions. The next step is to capture the equivalence relation "up to renaming of free variables" which is used in first-order unification. Because of dependencies, variables in Δ cannot be reordered arbitrarily. Moreover, variables in Δ which do not appear in M or elsewhere in Δ may be dropped and yield an equivalent cterm in the substitution ordering.

Let ϕ be a partial permutation from n into p. Then we define

$$\begin{aligned}\phi\langle [u_1{:}A_1] \ldots [u_p{:}A_p], M \rangle \\ = \langle [u_{\phi(1)}{:}A_{\phi(1)}] \ldots [u_{\phi(n)}{:}A_{\phi(n)}], M \rangle\end{aligned}$$

and, if the right-hand side is valid, call it a *pattern renaming* of the left-hand side. Let $\cong$ be the equivalence relation on valid cterms generated by α-conversion and pattern renaming.

Given $\Delta = [u_1{:}A_1] \ldots [u_p{:}A_p]$. A *valid substitution θ with domain Δ and codomain Δ'* is a tuple

$$\frac{\Gamma \text{ valid}}{\Gamma \vdash \mathsf{Prop} \Rightarrow \mathsf{Type}} \qquad \frac{\Gamma \vdash A \Rightarrow \kappa' \qquad \Gamma[x{:}A] \vdash B \Rightarrow \kappa}{\Gamma \vdash \{x{:}A\}\, B \Rightarrow \kappa}$$

$$\frac{\Gamma \vdash A \Rightarrow \kappa' \qquad \Gamma[x{:}A] \vdash M \Rightarrow B}{\Gamma \vdash [x{:}A]\, M \Rightarrow \{x{:}A\}\, B}$$

$$\frac{\Gamma \vdash h\, M_1 \ldots M_n : D \qquad \Gamma \vdash M_1 \Rightarrow A_1 \quad \ldots \quad \Gamma \vdash M_n \Rightarrow A_n}{\Gamma \vdash h\, M_1 \ldots M_n \Rightarrow D}$$

where D is atomic or Type

Figure 2: Canonical Forms

$\langle M_1, \ldots, M_p \rangle$ such that for $1 \leq i \leq p$

$$\Delta' \vdash M_i : [M_{i-1}/u_{i-1}] \ldots [M_1/u_1] A_i.$$

A substitution with domain Δ and codomain Δ' can be applied to a cterm of the form $\langle \Delta, M \rangle$, yielding $\langle \Delta', [M_p/u_p] \ldots [M_1/u_1] M \rangle$. It is easy to verify that valid substitutions, when applied to valid cterms, yield valid cterms. Composition of substitutions is defined in the obvious manner. For each valid context Δ there exists an identity substitution δ^{Δ}. We omit the superscript Δ when it can easily be inferred.

Since we assume that variables in a context have distinct names, we sometimes use a named notation for substitutions. Let Δ be $[x_1{:}A_1] \ldots [x_i{:}A_i] \ldots [x_p{:}A_p]$ and let θ be a substitution from Δ to Δ' of the form $\langle x_1, \ldots, M, \ldots x_p \rangle$ where Δ' has the form $[x_1{:}A_1] \ldots [y_1{:}B_1] \ldots [y_q{:}B_q] \ldots [x_p{:}[M/x_i]A_p]$. In this case we may write θ as $[x_i \leftarrow M]$.

A cterm $\langle \Delta', M' \rangle$ is an *instance of* $\langle \Delta, M \rangle$ if there exists a valid substitution θ with domain Δ and codomain Δ' such that $\theta \langle \Delta, M \rangle = \langle \Delta', M' \rangle$. We write $\langle \Delta', M' \rangle \leq \langle \Delta, M \rangle$.

In the first-order case, we can consider a term with free variables as a representation of all of its ground instances. This is not possible here, since many terms do not have any ground instances. Since it is undecidable if a given type is inhabited, this would make the equivalence relation on cterms undecidable.

In general, unification is undecidable already in the simply-typed λ-calculus [8]. Thus, some restriction must be placed on the occurrences of free variables (those in Δ) in order to obtain a class of cterms for which unification is decidable and most general unifiers exist. A very natural class has been discovered by Miller [15] as the basis for the logic programming language L_λ. This class has been applied to higher-order rewriting by Nipkow [17]. A generalization useful in the programming language Elf based on the LF Logical Framework is given by the author in [21]. It is used in Elf for term reconstruction and proof search (in the manner of constraint logic programming). The practical experience for the implementation of type reconstruction for Elf and the Calculus of Constructions provides strong evidence for the utility of this class of terms. Following Nipkow, we call cterms in this class *higher-order patterns* or *patterns*, for short. The main judgment defining higher-order patterns is $\Delta; \Gamma \vdash M$ Pat. It is defined on canonical forms M and is shown in Figure 3. A cterm $\langle \Delta, M \rangle$ is a *higher-order pattern* if $\Delta; \cdot \vdash M$ Pat. We will sometimes call M a pattern if Δ and Γ are fixed.

In the definition, ϕ is a partial permutation from n into p. Strictly speaking, the arguments to x in the last rule should be converted to their canonical forms (for example, when the arguments are of function type). Here and for the remainder of this paper, we will take the liberty of leaving such conversions implicit. For the remainder of this paper we deal primarily with valid patterns. We refer to a pattern of the form $x\, u_{\phi(1)} \ldots u_{\phi(n)}$ as a *generalized variable* or *Gvar*.

It is easy to see that given two valid patterns $\langle \Delta, M \rangle$ and $\langle \Delta', M' \rangle$, if $\langle \Delta, M \rangle \cong \langle \Delta', M' \rangle$ then $\langle \Delta, M \rangle \leq \langle \Delta', M' \rangle$ and $\langle \Delta', M' \rangle \leq \langle \Delta, M \rangle$. The implication in the other direction does not hold, but we will give an exact characterization of equivalence in Theorem 2.

4 Unification

We now develop a notion which is analogous to most-general unifiers for higher-order patterns. For technical reasons, this is expressed as the greatest common instance of two higher-order patterns, though the uni-

$$\frac{\Delta;\Gamma \vdash A \ \text{Pat} \qquad \Delta;\Gamma[u{:}A] \vdash M \ \text{Pat}}{\Delta;\Gamma \vdash [u{:}A]\,M \ \text{Pat}} \qquad \frac{\Delta;\Gamma \vdash A \ \text{Pat} \qquad \Delta;\Gamma[u{:}A] \vdash B \ \text{Pat}}{\Delta;\Gamma \vdash \{u{:}A\}\,B \ \text{Pat}}$$

$$\frac{\Delta;\Gamma \vdash M_1 \ \text{Pat} \quad \ldots \quad \Delta;\Gamma \vdash M_m \ \text{Pat}}{\Delta;\Gamma \vdash c\,M_1 \ldots M_m \ \text{Pat}} \qquad \frac{\Delta;\Gamma \vdash M_1 \ \text{Pat} \quad \ldots \quad \Delta;\Gamma \vdash M_m \ \text{Pat} \quad u \text{ in } \mathrm{dom}(\Gamma)}{\Delta;\Gamma \vdash u\,M_1 \ldots M_m \ \text{Pat}}$$

$$\frac{}{\Delta;\Gamma \vdash \mathsf{Prop} \ \text{Pat}} \qquad \frac{\Gamma = [u_1{:}A_1] \ldots [u_p{:}A_p] \qquad x \text{ in } \mathrm{dom}(\Delta)}{\Delta;\Gamma \vdash x\,u_{\phi(1)} \ldots u_{\phi(n)} \ \text{Pat}}$$

Figure 3: Valid Higher-Order Patterns

fying substitution (the most general unifier) is also explicitly calculated. Miller [15] develops a closely related notion for the simply-typed λ-calculus. His presentation is untyped, which already suggests that the algorithm does not rely on specific properties of simple types and may generalize even to an impredicative setting such as the Calculus of Constructions.

A *common instance* for valid patterns $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$ is any valid pattern $\langle \Delta, M \rangle$ such that $\langle \Delta, M \rangle \leq \langle \Delta', M' \rangle$ and $\langle \Delta, M \rangle \leq \langle \Delta'', M'' \rangle$. $\langle \Delta, M \rangle$ is a *greatest common instance* if it is a greatest lower bound with respect to $\leq$: for any common instance $\langle \Delta^*, M^* \rangle$, $\langle \Delta^*, M^* \rangle \leq \langle \Delta, M \rangle$. We refer to the problem of finding a greatest common instance as *unification*.

We now present a unification algorithm for patterns which finds a greatest common instance. The algorithm is presented as an inference system, written in such a way that it has a direct operational interpretation.

We define the judgment

$$\Delta^*; \Gamma \vdash_\theta M' \sqcap M'' = \langle \Delta, M \rangle$$

with the idea that, if given $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$ and a deduction of

$$\Delta' \dot\cup \Delta''; \cdot \vdash_\theta M' \sqcap M'' = \langle \Delta, M \rangle$$

then $\langle \Delta, M \rangle$ is a greatest common instance of $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$. θ will be the unifying substitution (with domain $\Delta' \dot\cup \Delta''$ and codomain Δ).

For the remainder of this section, we assume that Γ has the form $[u_1{:}A_1] \ldots [u_p{:}A_p]$. We refer to a variable in Γ as a *Uvar* (short for *universal variable*). Recall that $(\lambda[u_1{:}A_1] \ldots [u_p{:}A_p])\,M = [u_1{:}A_1] \ldots [u_p{:}A_p]\,M$. In order to implement the idea above, we preserve the following invariants in the inference system:

1. $\theta\langle \Delta^*, (\lambda\Gamma)\,M' \rangle = \langle \Delta, (\lambda\Gamma)\,M \rangle$,
2. $\theta\langle \Delta^*, (\lambda\Gamma)\,M'' \rangle = \langle \Delta, (\lambda\Gamma)\,M \rangle$,
3. $\Delta \dot\cup \Gamma \vdash M : A$,
4. $\Delta^* \dot\cup \Gamma \vdash M' : A$,
5. $\Delta^* \dot\cup \Gamma \vdash M'' : A$.

Operationally, during the execution of the algorithm, we assume we are given Δ^*, Γ, M' and M'' and we construct θ, Δ and M. The requirement on M' and M'' to have the same type and be well-typed is crucial here, since it guarantees the existence of canonical forms for M' and M''. However, it is not restrictive: Given any two valid patterns, we first unify their types and then the terms after applying the substitution (see Theorem 1). Such a well-typedness requirement cannot be maintained when considering unification for LF (or the Calculus of Constructions) without the restriction to patterns, where a complicated "acceptability" condition must be substituted instead (see [7]).

To obtain some intuition about the unification of higher-order patterns, consider the following unification problems. Types are not important to understand these examples (assume that all Uvars are of some base type i and restore the types of the remaining variables and constants accordingly).

$$[x]; [u_1][u_2] \vdash x\,u_1\,u_2 \sqcap u_1 = \langle \cdot, u_1 \rangle$$
$$[x]; [u_1][u_2] \vdash x\,u_1\,u_2 \sqcap c\,u_2 = \langle \cdot, c\,u_2 \rangle$$
$$[x]; [u_1][u_2][u_3] \vdash x\,u_2\,u_3\,u_1 \sqcap x\,u_1\,u_3\,u_2 = \langle [z], z\,u_3 \rangle$$
$$[x_1][x_2]; [u_1][u_2][u_3] \vdash x_1\,u_3\,u_1 \sqcap x_2\,u_2\,u_3 = \langle [z], z\,u_3 \rangle$$

Consider also these two simple examples which show that greatest common instances may not exist if the

restriction to patterns is relaxed. Of course, our inference system does not treat such terms, so the assertions are hypothetical.

$$[x][y];[u] \vdash c\,(x\,u\,u)\,(x\,a\,b) \sqcap c\,u\,y = \langle \cdot, c\,u\,a\rangle$$
$$[x][y];[u] \vdash c\,(x\,u\,u)\,(x\,a\,b) \sqcap c\,u\,y = \langle \cdot, c\,u\,b\rangle$$

$$[x][y];\cdot \vdash c\,(x\,a)\,(x\,b) \sqcap c\,a\,y = \langle \cdot, c\,a\,b\rangle$$
$$[x][y];\cdot \vdash c\,(x\,a)\,(x\,b) \sqcap c\,a\,y = \langle \cdot, c\,a\,a\rangle$$

Lam-Lam This is the rule responsible for unifying two λ-expressions. It will always be applicable for any two valid unifiable expressions of product type, since we maintain the invariant that the terms to be unified have the same type, and since the canonical form of a term of product type will be a λ-abstraction.

$$\frac{\Delta^*;\Gamma[u{:}A] \vdash_\theta M' \sqcap M'' = \langle \Delta, M\rangle}{\Delta^*;\Gamma \vdash_\theta [u{:}A]\,M' \sqcap [u{:}A]\,M'' = \langle \Delta, [u{:}A]\,M\rangle}$$

Prod-Prod When unifying products we have to first unify the corresponding types and pass the substitution along in the manner of the Rigid-Rigid rule below. We also require that A' and A'' be consistent, that is, both be of the same type (either Prop or Type).

$$\frac{\begin{array}{l}\Delta^*;\Gamma \vdash_{\theta_1} A' \sqcap A'' = \langle \Delta_1, A\rangle \\ \Delta_1;\Gamma[u{:}A] \vdash_{\theta_2} \theta_1 M' \sqcap \theta_1 M'' = \langle \Delta, M\rangle\end{array}}{\Delta^*;\Gamma \vdash_\theta \{u{:}A'\}\,M' \sqcap \{u{:}A''\}\,M'' = \langle \Delta, \{u{:}A^*\}\,M\rangle}$$

where $\theta = \theta_2 \circ \theta_1$ is a substitution from Δ^* into Δ and $A^* = \theta_2 A$.

Rigid-Rigid The terminology is borrowed from Huet [12]. This has two analogous subcases: it may be that we are unifying two atomic terms beginning with constants or with variables in $\mathrm{dom}(\Gamma)$. Thus h stands either for a constant c or a Uvar. The algorithm will fail in this case if it encounters a situation where the two heads are different or have a different number of arguments.

$$\frac{\begin{array}{l}\Delta^*;\Gamma \vdash_{\theta_1} M_1' \sqcap M_1'' = \langle \Delta_1, M_1\rangle \\ \quad\cdots \\ \Delta_{m-1};\Gamma \vdash_{\theta_m} \theta^{m-1} M_m' \sqcap \theta^{m-1} M_m'' = \langle \Delta_m, M_m\rangle\end{array}}{\begin{array}{l}\Delta^*;\Gamma \vdash_{\theta^m} \quad h\,M_1' \ldots M_m' \sqcap h\,M_1'' \ldots M_m'' \\ \qquad\qquad = \langle \Delta_m, h\,M_1^* \ldots M_m^*\rangle\end{array}}$$

where $\theta^j = \theta_j \circ \cdots \circ \theta_1$ for $1 \le j \le m$ is a substitution from Δ^* into Δ_j and $M_j^* = (\theta_{j+1} \circ \cdots \circ \theta_m) M_j$. The method of unifying the later arguments to h given the unifiers for the earlier ones is important for the operational reading of the rules, since it is this order which guarantees the invariant that the terms being unified have the same type. If $m = 0$ then $\theta^0 = \delta^{\Delta^*}$, the identity substitution on Δ^*. We also include in **Rigid-Rigid** the case where $h = \mathsf{Prop}$. Note that we have no case for unifying Type, since $\langle \Gamma, \mathsf{Type}\rangle$ is not a valid cterm.

Gvar-Const Rather than performing the substitution all at once, we will incrementally "imitate" (in the sense of Huet) the term on the right-hand side. The rule below applies only if x does not occur free in $c\,M_1 \ldots M_m$—otherwise the two terms are not unifiable. For this condition to be correct, we need the requirement that the terms be in canonical form. For example, x and $([u]\,[v]\,v)\,x$ are unifiable.

$$\frac{\begin{array}{l}\Delta_0;\Gamma \vdash_{\theta_1} x_1\,u_{\phi(1)} \ldots u_{\phi(n)} \sqcap M_1'' = \langle \Delta_1, M_1\rangle \\ \quad\cdots \\ \Delta_{m-1};\Gamma \vdash_{\theta_m} \quad x_m\,u_{\phi(1)} \ldots u_{\phi(n)} \sqcap \theta^{m-1} M_m'' \\ \qquad\qquad\qquad\qquad = \langle \Delta_m, M_m\rangle\end{array}}{\begin{array}{l}\Delta^*;\Gamma \vdash_{\theta^m} \quad x\,u_{\phi(1)} \ldots u_{\phi(n)} \sqcap c\,M_1'' \ldots M_m'' \\ \qquad\qquad = \langle \Delta_m, c\,M_1^* \ldots M_m^*\rangle\end{array}}$$

where Δ_0 is Δ^* where $[x]$ has been replaced by $[x_1] \ldots [x_m]$ at appropriate types. Furthermore,

$$\begin{array}{l}\theta_0 = [x \leftarrow [u_{\phi(1)}{:}A_{\phi(1)}] \ldots [u_{\phi(n)}{:}A_{\phi(n)}] \\ \qquad c\,(x_1\,u_{\phi(1)} \ldots u_{\phi(n)}) \ldots (x_m\,u_{\phi(1)} \ldots u_{\phi(n)})]\end{array}$$

(considered as a substitution from Δ^* to Δ_0) and $\theta^j = \theta_j \circ \cdots \theta_1 \circ \theta_0$ and $M_j^* = (\theta_{j+1} \circ \cdots \circ \theta_m) M_j$. The types omitted above are fully determined by the type of the constant c and the types assigned by Γ and Δ^*. We also include here the case that the right-hand side is Prop.

Gvar-Uvar In this case we perform a "projection" (in the sense of Huet), if the Uvar at the head of the term is one of the arguments of x and x does not occur in the right-hand side. If this condition is not satisfied, unification will fail. Thus, for some $1 \le i \le n$

$$\frac{\begin{array}{l}\Delta_0;\Gamma \vdash_{\theta_1} x_1\,u_{\phi(1)} \ldots u_{\phi(n)} \sqcap M_1'' = \langle \Delta_1, M_1\rangle \\ \quad\cdots \\ \Delta_{m-1};\Gamma \vdash_{\theta_m} \quad x_m\,u_{\phi(1)} \ldots u_{\phi(n)} \sqcap \theta^{m-1} M_m'' \\ \qquad\qquad\qquad\qquad = \langle \Delta_m, M_m\rangle\end{array}}{\begin{array}{l}\Delta^*;\Gamma \vdash_{\theta^m} \quad x\,u_{\phi(1)} \ldots u_{\phi(n)} \sqcap u_{\phi(i)}\,M_1'' \ldots M_m'' \\ \qquad\qquad = \langle \Delta_m, u_{\phi(i)}\,M_1^* \ldots M_m^*\rangle\end{array}}$$

where Δ_0 is Δ^* where $[x]$ has been replaced by $[x_1] \ldots [x_m]$ at appropriate types, where

$$\begin{array}{l}\theta_0 = [x \leftarrow [u_{\phi(1)}{:}A_{\phi(1)}] \ldots [u_{\phi(n)}{:}A_{\phi(n)}] \\ \qquad u_{\phi(i)}\,(x_1\,u_{\phi(1)} \ldots u_{\phi(n)}) \ldots (x_m\,u_{\phi(1)} \ldots u_{\phi(n)})]\end{array}$$

(considered as a substitution from Δ^* to Δ_0), and where $\theta^j = \theta_j \circ \cdots \circ \theta_1 \circ \theta_0$ and $M_j^* = (\theta_{j+1} \circ \cdots \circ \theta_m) M_j$. The types omitted above are fully determined by the types assigned in Γ and Δ^*.

Gvar-Gvar-Same In Huet's algorithm this case is simply postponed as unifiable. Here we collect the arguments on which the two terms agree.

$$\Delta^*; \Gamma \vdash_\theta \quad x\, u_{\phi(1)} \ldots u_{\phi(n)} \sqcap x\, u_{\psi(1)} \ldots u_{\psi(n)} = \langle \Delta', z\, u_{\rho(1)} \ldots u_{\rho(l)} \rangle$$

where Δ' is Δ^* where $[x]$ has been replaced by $[z]$ and $\theta = [x \leftarrow [u_{\phi(1)}{:}A_{\phi(1)}] \ldots [u_{\phi(n)}{:}A_{\phi(n)}]\, z\, u_{\rho(1)} \ldots u_{\rho(l)}]$ and ρ is a partial permutation satisfying that $\phi(i) = \psi(i)$ iff there is a k such that $\rho(k) = \phi(i)$. ρ always exists and is unique up to permutation.

Gvar-Gvar-Diff To unify two different generalized variables, we have to collect the common arguments.

$$\Delta^*; \Gamma \vdash_\theta \quad x\, u_{\phi(1)} \ldots u_{\phi(n)} \sqcap y\, u_{\psi(1)} \ldots u_{\psi(m)} = \langle \Delta', z\, u_{\rho(1)} \ldots u_{\rho(l)} \rangle$$

where Δ' is Δ^* where the leftmost among $[x]$ and $[y]$ has been replaced by $[z]$ and the rightmost has been deleted. θ substitutes

$$x \leftarrow [u_{\phi(1)}] \ldots [u_{\phi(n)}]\, z\, u_{\rho(1)} \ldots u_{\rho(l)}, \text{ and}$$
$$y \leftarrow [u_{\psi(1)}] \ldots [u_{\psi(m)}]\, z\, u_{\rho(1)} \ldots u_{\rho(l)}.$$

ρ is a partial permutation satisfying that $\phi(i) = \psi(j)$ iff there is a k such that $\rho(k) = \phi(i)$. Here, too, ρ always exists and is unique up to permutation.

Theorem 1 (Greatest Common Instance) *Given valid patterns $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$. If*

1. *$\Delta' \vdash M' : A'$ and $\Delta'' \vdash M'' : A''$, and*
2. *Either*
   - *A and A' are consistent (of the same type) and $\Delta' \dot\cup \Delta''; \cdot \vdash_{\theta^*} A' \sqcap A'' = \langle \Delta^*, A \rangle$, or*
   - *$A' = A'' = \mathsf{Type}$, $\theta^* = \delta$, $\Delta^* = \Delta' \dot\cup \Delta''$ and*
3. *$\Delta^*; \cdot \vdash_\theta \theta^* M' \sqcap \theta^* M'' = \langle \Delta, M \rangle$*

then $\langle \Delta, M \rangle$ is a greatest common instance of $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$ with the unifying substitution $\theta \circ \theta^*$. Moreover, there is an effective algorithm which computes a greatest common instance $\langle \Delta, M \rangle$ if it exists and fails otherwise.

The proof is delicate and tedious, but not difficult, following the standard patterns (see, for example, Lassez *et al.* [14]). Given the invariants maintained in the deductive system, the main difficulty is in showing that the partial permutations ρ we construct cover all possible instances (for the basic idea of this proof, see Miller [15]).

It is easy to see that the operational interpretation of this system will always terminate (either with failure, if no deduction can be constructed) or with a greatest common instance and a most general unifier.

Examining the deductive system shows that the only non-determinism arises in the choice of the permutation ρ in the Gvar-Gvar cases. This observation gives rise to a characterization of equivalence.

Let $\langle \Delta, M \rangle$ be a pattern and assume Δ is of the form $\Delta_1\, [x{:}\{u_1{:}A_1\} \ldots \{u_n{:}A_n\}\, A]\, \Delta_2$ and that ϕ is a permutation from n into n. Then let

$$\begin{aligned} N &= [u_1] \ldots [u_n]\, x'\, u_{\phi(1)} \ldots u_{\phi(n)}, \\ \Delta' &= \Delta_1\, [x'{:}\{u_{\phi(1)}\} \ldots \{u_{\phi(n)}\}\, A]\, [N/x] \Delta_2, \\ \theta &= [x \leftarrow N] \end{aligned}$$

where θ is a substitution from Δ into Δ'. We refer to θ as a *permuting substitution* and say that $\theta\langle \Delta, M \rangle = \langle \Delta', M' \rangle$ is a *pattern permutation* of $\langle \Delta, M \rangle$. This is not to be confused with the weaker notion of *pattern renaming* ($\cong$) defined earlier.

Theorem 2 *Let $\equiv$ be the least equivalence relation on patterns which includes pattern renaming ($\cong$) and pattern permutation. Then $\langle \Delta, M \rangle \le \langle \Delta', M' \rangle$ and $\langle \Delta', M' \rangle \le \langle \Delta, M \rangle$ iff $\langle \Delta, M \rangle \equiv \langle \Delta', M' \rangle$.*

As an example, consider

$$\begin{aligned} &\langle [x{:}i \to i \to i \to i], [u_1{:}i][u_2{:}i][u_3{:}i]\, x\, u_3\, u_1\, u_2 \rangle \\ &\quad \equiv \langle [x'{:}i \to i \to i \to i], [u_1{:}i][u_2{:}i][u_3{:}i]\, x'\, u_2\, u_3\, u_1 \rangle \end{aligned}$$

It is easy to see that these two are equivalent under the instantiation ordering.

5 Anti-Unification

A *common anti-instance* for valid patterns $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$ is any valid pattern $\langle \Delta, M \rangle$ such that $\langle \Delta', M' \rangle \le \langle \Delta, M \rangle$ and $\langle \Delta'', M'' \rangle \le \langle \Delta, M \rangle$. $\langle \Delta, M \rangle$ is a *least common anti-instance* if for any common anti-instance $\langle \Delta^*, M^* \rangle$, $\langle \Delta, M \rangle \le \langle \Delta^*, M^* \rangle$. We refer to the problem of finding a least common anti-instance as *anti-unification*.

For first-order terms, any variable x serves as the top element of the semi-lattice formed by the instantiation ordering, since a variable can be instantiated to

an arbitrary term. In the Calculus of Constructions, there are pairs of patterns which have no upper bound. Consider, for example, $\mathsf{Prop} \sqcup (\mathsf{Prop} \to \mathsf{Prop})$. Since we have no variables of type Type, there is no variable or other term which can be instantiated to both of these terms.

The central ingredient in the elegant formulation of first-order anti-unification by Huet [13] is a global function Φ which maps pairs of terms to variables. Consider $f(a,a) \sqcup f(b,b) = f(x,x)$ if $\Phi(a,b) = x$. The function is used to guarantee that the same disagreements are mapped to the same variable (x, in this example). We were not able to find a corresponding formulation in this setting, due to the nature of the type dependencies. Thus we return to a formulation similar to algorithms presented by Plotkin [22] and Reynolds [24].

In our setting, there are several complicating factors. We discuss each of them in turn. We assume that we are considering a subproblem of anti-unifying $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$, and we fix Δ' and Δ''. In the process of traversing M' and M'' to find disagreements, we build up local contexts Γ, Γ' and Γ'' (similar to the way this is required for unification) and anti-unify the subterms of M' and M'' as $\langle \Gamma', N' \rangle$ and $\langle \Gamma'', N'' \rangle$ within their local context.

The first of the complications is the presence of variable binders in the terms. For example, given constants $a{:}i$ and $f{:}i \to i \to i$, consider

$$[u{:}i]\, f\, u\, u \;\sqcup\; [u{:}i]\, f\, a\, u = [u{:}i]\, f\, (x u)\, u$$

where the variable x has type $i \to i$. Observe that $[u{:}i]\, f\, x\, u$ is not a solution here, since it cannot be instantiated to the first term. On the other hand,

$$[u{:}i]\, f\, b\, u \;\sqcup\; [u{:}i]\, f\, a\, u = [u{:}i]\, f\, x\, u$$

since u does not appear in a or b.

Thus, when N and N' differ at the top-level, $\langle \Gamma', N' \rangle \sqcup \langle \Gamma'', N'' \rangle = x\, u_{\phi(1)} \ldots u_{\phi(n)}$ for some partial permutation ϕ from n into p, where $\Gamma = [u_1{:}A_1] \ldots [u_p{:}A_p]$. In order to be an upper bound, $u_{\phi(1)} \ldots u_{\phi(n)}$ must include at least all the variables of Γ' and Γ'' which are free in N' and N''. The suspicion arises that one would collect *exactly* those u_k which appear either in N' or N''. However, this is not quite possible, due to the presence of types—the second complication. Consider

$$\begin{array}{ll} & [u_1{:}o][u_2{:}\mathrm{true}(u_1)][u_3{:}\mathrm{true}(u_1)]\, u_2 \\ \sqcup & [u_1{:}o][u_2{:}\mathrm{true}(u_1)][u_3{:}\mathrm{true}(u_1)]\, u_3 \\ = & [u_1{:}o][u_2{:}\mathrm{true}(u_1)][u_3{:}\mathrm{true}(u_1)]\, x\, u_1\, u_2\, u_3 \end{array}$$

where $\mathrm{true} : o \to \mathsf{Prop}$. Here, the argument u_1 to x is required so that the variable x can be typed (with the type $\{u_1{:}o\}\{u_2{:}\mathrm{true}(u_1)\}\{u_3{:}\mathrm{true}(u_1)\}\,\mathrm{true}(u_1)$).

The third complication is the possibility of permuting the arguments to a generalized variable in order to obtain a more specific solution. Consider

$$\begin{array}{ll} & [u{:}i][v{:}i]\, f\, (f\, (g\, u)\, v)\, (f\, (g\, v)\, u) \\ \sqcup & [u{:}i][v{:}i]\, f\, (g\, v\, (g\, u))\, (g\, u\, (g\, v)) \\ = & [u{:}i][v{:}i]\, f\, (x\, u\, v)\, (x\, v\, u) \end{array}$$

Note that we can use the same variable x even though the two disagreements $(f\,(g\,u)\,v) \sqcup (g\,v\,(g\,u))$ and $(f\,(g\,v)\,u) \sqcup (g\,u\,(g\,v))$ are different! The key observation here is that the two disagreements, each considered in the context $[u{:}i][v{:}i]$ are permutations of each other. This allows the generalization to capture certain symmetries.

The main judgment is

$$\Delta'; \Delta''; \Delta; \Delta^*; \Gamma \vdash^{\theta^*,\theta^{**}}_{\theta',\theta''} M = \langle \Gamma', M' \rangle \sqcup \langle \Gamma'', M'' \rangle \mathbin{@} A$$

with the intent that, if

$$\Delta'; \Delta''; \Delta; \cdot; \cdot \vdash^{\delta,\delta}_{\theta',\theta''} M = \langle \cdot, M' \rangle \sqcup \langle \cdot, M'' \rangle \mathbin{@} A$$

then $\langle \Delta, M \rangle$ is a least common anti-instance of $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$ and $\Delta \vdash M : A$. Our rules preserve the following invariants (for some A' and A'')

1. $\theta' \langle \Delta, (\lambda\Gamma)\, M \rangle = \langle \Delta', (\lambda\Gamma')\, M' \rangle$,
2. $\theta'' \langle \Delta, (\lambda\Gamma)\, M \rangle = \langle \Delta'', (\lambda\Gamma'')\, M'' \rangle$,
3. $\theta^* \langle \Delta^*, (\lambda\Gamma)\, A \rangle = \langle \Delta', (\lambda\Gamma')\, A' \rangle$,
4. $\theta^{**} \langle \Delta^*, (\lambda\Gamma)\, A \rangle = \langle \Delta'', (\lambda\Gamma'')\, A'' \rangle$,
5. $\Delta' \dot\cup \Gamma' \vdash M' : A'$ and $\Delta'' \dot\cup \Gamma'' \vdash M'' : A''$,
6. $\Delta \dot\cup \Gamma \vdash M : A$.

In the operational reading of these rules, Δ^*, Γ, θ^*, θ^{**}, Γ', M', and Γ'' and M'' are inputs, and Δ, M, and θ' and θ'' are outputs. A is also considered an input, which is not a restriction (see Theorem 3).

Operationally, Δ^* is the domain of the substitutions θ^* and θ^{**}: it contains types for all the variables which had to be introduced during the anti-unification so far. Δ^* will be an initial prefix of the final result Δ.

In the presentation of the rules in Figure 4, we suppress Δ' and Δ'', since they remain constant throughout the inferences. We also assume throughout that Γ has the form $[u_1{:}A_1] \ldots [u_p{:}A_p]$.

$$\frac{\Delta; \Delta^*; \Gamma[u{:}A] \vdash^{\theta^*,\theta^{**}}_{\theta',\theta''} M = \langle \Gamma'[u{:}A'], M' \rangle \sqcup \langle \Gamma''[u{:}A''], M'' \rangle \mathbin{@} B}{\Delta; \Delta^*; \Gamma \vdash^{\theta^*,\theta^{**}}_{\theta',\theta''} [u{:}A]\, M = \langle \Gamma', [u{:}A']\, M' \rangle \sqcup \langle \Gamma'', [u{:}A'']\, M'' \rangle \mathbin{@} \{u{:}A\}\, B} \textbf{Lam-Lam}$$

$$\frac{\begin{array}{l}\Delta_1; \Delta^*; \Gamma \vdash^{\theta^*,\theta^{**}}_{\theta'_1,\theta''_1} A = \langle \Gamma', A' \rangle \sqcup \langle \Gamma'', A'' \rangle \mathbin{@} \kappa' \\ \Delta; \Delta_1; \Gamma[u{:}A] \vdash^{\theta'_1,\theta''_1}_{\theta',\theta''} B = \langle \Gamma', B' \rangle \sqcup \langle \Gamma'', B'' \rangle \mathbin{@} \kappa\end{array}}{\Delta; \Delta^*; \Gamma \vdash^{\theta^*,\theta^{**}}_{\theta',\theta''} \{u{:}A\}\, B = \langle \Gamma', \{u{:}A'\}\, B' \rangle \sqcup \langle \Gamma'', \{u{:}A''\}\, B'' \rangle \mathbin{@} \kappa} \textbf{Prod-Prod-Same}$$

$$\frac{\begin{array}{l}\Delta_1; \Delta^*; \Gamma \vdash^{\theta^*,\theta^{**}}_{\theta'_1,\theta''_1} M_1 = \langle \Gamma', M'_1 \rangle \sqcup \langle \Gamma'', M''_1 \rangle \mathbin{@} A_1 \\ \quad \ldots \\ \Delta_m; \Delta_{m-1}; \Gamma \vdash^{\theta'_{m-1},\theta''_{m-1}}_{\theta'_m,\theta''_m} M_m = \langle \Gamma', M'_m \rangle \sqcup \langle \Gamma'', M''_m \rangle \mathbin{@} A_m\end{array}}{\Delta_m; \Delta^*; \Gamma \vdash^{\theta^*,\theta^{**}}_{\theta'_m,\theta''_m} h\, M_1 \ldots M_m = \langle \Gamma', h\, M'_1 \ldots M'_m \rangle \sqcup \langle \Gamma'', h\, M''_1 \ldots M''_m \rangle \mathbin{@} A} \textbf{Atom-Atom-Same}$$

$$\frac{}{\Delta^*; \Delta^*; \Gamma \vdash^{\theta^*,\theta^{**}}_{\theta^*,\theta^{**}} x\, u_{\phi(1)} \ldots u_{\phi(n)} = \langle \Gamma', M' \rangle \sqcup \langle \Gamma'', M'' \rangle \mathbin{@} C} \textbf{Diff-Old}$$

where C is atomic, x in $\mathrm{dom}(\Delta^*)$ and ϕ satisfies

$$\begin{array}{rcl}(\theta^* x)\, u_{\phi(1)} \ldots u_{\phi(n)} & = & M', \quad \text{and} \\ (\theta^{**} x)\, u_{\phi(1)} \ldots u_{\phi(n)} & = & M''.\end{array}$$

$$\frac{}{\Delta; \Delta^*; \Gamma \vdash^{\theta^*,\theta^{**}}_{\theta',\theta''} x\, u_{\phi(1)} \ldots u_{\phi(n)} = \langle \Gamma', M' \rangle \sqcup \langle \Gamma'', M'' \rangle \mathbin{@} C} \textbf{Diff-New}$$

where C is atomic (which excludes Type) and

$$\begin{array}{rcl}\Delta & = & \Delta^*\, [x{:}\{u_{\phi(1)}{:}A_{\phi(1)}\} \ldots \{u_{\phi(n)}{:}A_{\phi(n)}\}\, C] \\ \theta' & = & \theta^*, [u_{\phi(1)}{:}A_{\phi(1)}] \ldots [u_{\phi(n)}{:}A_{\phi(n)}]\, M' \\ \theta'' & = & \theta^{**}, [u_{\phi(1)}{:}A_{\phi(1)}] \ldots [u_{\phi(n)}{:}A_{\phi(n)}]\, M''.\end{array}$$

Figure 4: Rules for Anti-Unification

Lam-Lam The main observation here is that we do not have to anti-unify the types: the invariants guarantee that A serves as a least common anti-instance.

Prod-Prod-Same If both M' and M'' are products, we proceed in the same way as for abstractions, except that we also first need to anti-unify the types. For this rule to apply, we also require that A' and A'' be consistent—otherwise anti-unification fails.

Atom-Atom-Same When unifying two atomic terms with equal head we have to proceed from left to right, anti-unifying the corresponding arguments and building up the substitutions θ' and θ'' and the context Δ.

Here the types A_i are determined by the type of h and the terms $M_1, \ldots M_{i-1}$. This case is intended to cover the cases where h is Prop, a constant c or a variable bound in Γ, Γ', and Γ'' (by the invariants, they bind the same variables, though possibly at different types).

Diff-Old This case applies when we encounter a disagreement we have seen already (which will have been recorded in θ^* and θ^{**}). See earlier motivating examples for a note on the need to check modulo a partial permutation.

Diff-New For this case we require, that none of the other cases apply. Here we need to create a new variable and give it just enough arguments so that the resulting term can be instantiated to both M' and M'' and is well-typed. If C is not atomic then anti-

unification fails.

$[u_{\phi(1)}{:}A_{\phi(1)}] \ldots [u_{\phi(n)}{:}A_{\phi(n)}]\, M'$ is the substitution term for x in θ', which otherwise behaves like θ^*. The partial permutation ϕ must be a smallest permutation such that there exists an i such that $\phi(i) = k$ iff u_k occurs in M' or in M'' and such that $x u_{\phi(1)} \ldots u_{\phi(n)}$ is well-typed in $\Delta \dot\cup \Gamma$. Such a partial permutation always exists under the stated conditions on C above and is unique up to permutation.

Theorem 3 (Least Common Anti-Instance) *Given valid patterns $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$. If*

1. $\Delta' \vdash M' : A'$ *and* $\Delta'' \vdash M'' : A''$, *and*
2. *either*

   A' *and* A'' *are consistent (both of type* κ*) and* $\Delta'; \Delta''; \Delta^*; \cdot; \cdot \vdash^{\delta,\delta}_{\theta^*,\theta^{**}} A = \langle \cdot, A' \rangle \sqcup \langle \cdot, A'' \rangle \mathbin{@} \kappa$, *or*

   $A' = A'' = \mathsf{Type} = A, \theta^* = \theta^{**} = \delta, \Delta^* = \cdot$, *and*

3. $\Delta'; \Delta''; \Delta; \Delta^*; \cdot \vdash^{\theta^*,\theta^{**}}_{\theta',\theta''} M = \langle \cdot, M' \rangle \sqcup \langle \cdot, M'' \rangle \mathbin{@} A$

then $\langle \Delta, M \rangle$ is a least common anti-instance of $\langle \Delta', M' \rangle$ and $\langle \Delta'', M'' \rangle$. Moreover, there is an effective algorithm which computes a least common anti-instance $\langle \Delta, M \rangle$ if it exists and fails otherwise.

The proof follows patterns similar to the one given by Reynolds [24], though significantly more intricate in the details, due the presence of binding operators and types. Key here is a lemma that the invariants stated at the beginning are preserved. Termination of the operational version of the algorithm can easily be seen.

6 Examples of Anti-Unification

The examples below are mostly situated in the LF subcalculus of the Calculus of Constructions [10]. We show those parts of a full axiomatization of first-order logic in LF which are necessary for our examples in Figure 5. The logical connectives are written in infix notation.

In the examples below, both Δ' and Δ'' are always empty, since (free) variables in the patterns to be anti-unified do not play an interesting role: they behave essentially as if they were constants. The first set of simple examples can be found in Figure 6.

We omit the types of the generalization variables from here on, since they can easily be filled in.

$$
\begin{array}{rcl}
o & : & \mathsf{Prop} \\
i & : & \mathsf{Prop} \\
\mathrm{zero} & : & i \\
\mathrm{succ} & : & i \to i \\
\wedge & : & o \to o \to o \\
\supset & : & o \to o \to o \\
\mathrm{true} & : & o \to \mathsf{Prop} \\
\mathrm{andi} & : & \{p{:}o\}\{q{:}o\}\ \mathrm{true}(p) \to \mathrm{true}(q) \to \mathrm{true}(p \wedge q) \\
\mathrm{andel} & : & \{p{:}o\}\{q{:}o\}\ \mathrm{true}(p \wedge q) \to \mathrm{true}(p) \\
\mathrm{ander} & : & \{p{:}o\}\{q{:}o\}\ \mathrm{true}(p \wedge q) \to \mathrm{true}(q) \\
\mathrm{impliesi} & : & \{p{:}o\}\{q{:}o\}\ (\mathrm{true}(p) \to \mathrm{true}(q)) \to \mathrm{true}(p \supset q) \\
\mathrm{impliese} & : & \{p{:}o\}\{q{:}o\}\ \mathrm{true}(p \supset q) \to \mathrm{true}(p) \to \mathrm{true}(q)
\end{array}
$$

Figure 5: Example Signature

$$
\begin{array}{rl}
 & [u{:}i][v{:}i]\, f\, u\, v \\
\sqcup & [u{:}i][v{:}i]\, f\, v\, u \\
= & [u{:}i][v{:}i]\, f\, (x\, u\, v)\, (x\, v\, u)
\end{array}
$$

$$
\begin{array}{rl}
 & [u{:}i][v{:}i][w{:}i]\, g\, (f\, u\, w)\, (f\, v\, w)\, (f\, v\, u) \\
\sqcup & [u{:}i][v{:}i][w{:}i]\, g\, (h\, v\, u)\, (h\, u\, v)\, (h\, w\, v) \\
= & [u{:}i][v{:}i][w{:}i]\, g\, (x\, u\, v\, w)\, (x\, v\, u\, w)\, (x\, v\, w\, u)
\end{array}
$$

$$
\begin{array}{rl}
 & [u{:}i][v{:}i]\, f(g\, u\, v)\, ([w{:}i]\, g\, w\, v) \\
\sqcup & [u{:}i][v{:}i]\, f(h\, v\, u)\, ([w{:}i]\, h\, v\, w) \\
= & [u{:}i][v{:}i]\, f(x\, v\, u)\, ([w{:}i]\, x\, v\, w)
\end{array}
$$

The next example exhibits a phenomenon which can arise due to the presence of explict type abstraction and application. The terms in this example are not presented in canonical form.

$$
\begin{array}{rl}
 & [u{:}\{v{:}\mathsf{Prop}\}(v \to v)]\, u\, (\{v{:}\mathsf{Prop}\}(v \to v))\, u \\
\sqcup & [u{:}\{v{:}\mathsf{Prop}\}(v \to v)]\, u \\
= & [u{:}\{v{:}\mathsf{Prop}\}(v \to v)]\, x\, u
\end{array}
$$

The remaining examples are two simple proof generalizations. In the first one, x_3 and x_4 must be chosen differently, since they must be typed differently.

7 Conclusion and Further Work

The unification algorithm we present has been implemented (with some optimizations) in its full generality

i	$\sqcup$	$i \to i$	$=$	x	where $\Delta = [x{:}\mathsf{Prop}]$
zero	$\sqcup$	succ	$=$	y	where $\Delta = [x{:}\mathsf{Prop}][y{:}x]$
i	$\sqcup$	zero		undefined	
Prop	$\sqcup$	Prop $\to$ Prop		undefined	
$[f]\, f$ zero zero	$\sqcup$	$[f]\, f$ (succ zero) (succ zero)	$=$	$[f] f\, x\, x$	where $\Delta = [x{:}i]$

Figure 6: Examples of Anti-Unification

$$
\begin{array}{ll}
 & [p{:}o][u{:}\mathrm{true}(p \supset p)][v{:}\mathrm{true}(p)]\ \mathrm{impliese}\, p\, p\, u\, v \\
\sqcup & [p{:}o][u{:}\mathrm{true}(p)][v{:}\mathrm{true}(p \supset p)]\ \mathrm{impliese}\, p\, p\, v\, u \\
= & [p{:}o][u{:}\mathrm{true}(x_1\, p)][v{:}\mathrm{true}(x_2\, p)]\ \mathrm{impliese}\, p\, p\, (x_3\, p\, u\, v)\, (x_4\, p\, u\, v) \\
 & \\
 & [p{:}o][q{:}o][u{:}\mathrm{true}(p \wedge q)]\ \mathrm{andi}\, q\, p\, (\mathrm{ander}\, p\, q\, u)\, (\mathrm{andel}\, p\, q\, u) \\
\sqcup & [p{:}o][q{:}o][u{:}\mathrm{true}(p \wedge q)]\ \mathrm{andi}\, p\, q\, (\mathrm{andel}\, p\, q\, u)\, (\mathrm{ander}\, p\, q\, u) \\
= & [p{:}o][q{:}o][u{:}\mathrm{true}(p \wedge q)]\ \mathrm{andi}\, (x_1\, p\, q)\, (x_1\, q\, p)\, (x_2\, p\, q\, u)\, (x_3\, p\, q\, u)
\end{array}
$$

Figure 7: Examples of Proof Generalization via Anti-Unification

in the implementation of the Elf language [20, 21] and exhibits good characteristics in terms of performance. The algorithm is used for the solution of constraints during the execution of a logic program and for performing term and type reconstruction after parsing. However, many improvements are still possible and the subject of current investigation.

The implementation of anti-unification is not yet complete, and its efficiency is still undetermined. From examples it is clear that the algorithm can perform useful proof generalizations, but the question exactly how this tool is to be tied into a proof development environment or language has not yet been settled.

The form of anti-unification would be different in a calculus with universes. It appears plausible that in a calculus with a cumulative hierarchy of universes we can recover the property of first-order terms that any pair of terms has an anti-unifier. This anti-unifier should exist at a level at most one above the maximum of the two patterns which are anti-unified.

From some examples it appears that the intuitiveness of generalizations can be significantly improved if anti-unification takes into account additional equations which come from the object theory under consideration. It is conceivable that there is an interesting theory of equational anti-unification to be discovered, similar to the theory of equational first-order unification.

Acknowledgments

The author would like to thank Scott Dietzen for a number of discussions on generalization and anti-unification and the anonymous referees for helpful suggestions. This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

References

[1] Peter B. Andrews, Sunil Issar, Dan Nesmith, and Frank Pfenning. The TPS theorem proving system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, Germany*, pages 641–642. Springer-Verlag LNCS 449, July 1990. System abstract.

[2] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991. To appear.

[3] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[4] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. *Machine Learning*, 1991. To

appear. Available as Technical Report CMU-CS-89-160, Carnegie Mellon University.

[5] Gilles Dowek. A proof synthesis algorithm for a mathematical vernacular in a restriction of the Calculus of Constructions. Unpublished manuscript, January 1991.

[6] Dominic Duggan. *Caliban: A Programming Language and Environment Based on Types as Specifications*. PhD thesis, University of Maryland, College Park, 1991. In preparation.

[7] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

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

[9] Masami Hagiya. Generalization from partial parameterization in higher-order type theory. *Theoretical Computer Science*, 63:113–139, 1989.

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, to appear. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[11] Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the Calculus of Constructions. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 241–256. Springer-Verlag LNCS 352, March 1989.

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

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

[14] J-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases*, chapter 15, pages 587–626. Morgan Kaufmann, 1988.

[15] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, pages 253–281. Springer-Verlag LNCS 475, 1991.

[16] 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, Cambridge, Massachusetts, August 1988. MIT Press.

[17] Tobias Nipkow. Higher-order critical pairs. In *Sixth Annual Symposium on Logic in Computer Science*. IEEE, July 1991. To appear.

[18] Christine Paulin-Mohring. Extracting F_ω programs from proofs in the calculus of constructions. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 89–104. ACM Press, January 1989.

[19] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, Computer Laboratory, University of Cambridge, January 1990.

[20] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE, June 1989.

[21] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991. To appear.

[22] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[23] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, University of Edinburgh, 1990. Available as CST-69-90, also published as ECS-LFCS-90-125.

[24] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[25] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, May 1990.