ソフトウェア基礎論 36−3
（1990. 9. 21）

定理証明手続きとしての高階単一化

萩谷昌己

京都大学数理解析研究所
京都市左京区北白川追分町

概要

Logical Framework などの高階の型理論における高階の単一化手続きは、軟々対にも射影を許すことにより、定理証明手続きとして用いることができることを指摘する。特に、Logical Framework における高階の単一化手続きから、プログラム変換によって Prolog のインタープリタが得られることを示す。

Higher-order Unification as a Theorem Proving Procedure

Masami Hagiya

RIMS, Kyoto University
Sakyo, Kyoto 606, JAPAN
hagiya@kurims.kyoto-u.ac.jp

Abstract

It is shown that higher-order unification procedures for higher-order type systems such as Logical Framework can be used as a theorem proving procedure if projection is also allowed on flexible-flexible pairs. In particular, Prolog interpreter is derived from a higher-order unification procedure for Logical Framework by program transformation.

1 Introduction

There have been developed a number of theorem provers or proof checkers that use higher-order unification as their pattern matching procedure. For example, λPROLOG [8] successfully incorporated higher-order unification into Prolog and established a foundation for Prolog-based theorem proving and program transformation. Isabelle [9] also has a higher-order unifier as one of its repertoire of unification procedures. Elf [10] is another example, in which higher-order unification for dependent types and surjective pairing is used as a pattern matching procedure.

For those systems, higher-order unification is only a part of a whole theorem proving procedure. This paper pursues the opposite direction; it shows that a unification procedure for a higher-order type system having dependent types, without any further device, can also serve as a theorem proving procedure for the type system.

Higher-order unification for *simple type theory* was first formulated by Huet [7]. He devised a practical procedure for higher-order unification, called *preunification,* in which *flexible-flexible pairs* are not solved and left as *constraints* for the resulting answer substitution. In practical situations, flexible-flexible pairs seldom appear and preunification works very well.

Roughly speaking, preunification is a procedure for systematically generating terms for a given simple type. Leaving flexible-flexible pairs as constraints means to stop the systematic generation of terms where no information is suggested by the original unification problem. However, among the operations of preunification, *projection* can be applied on flexible-flexible pairs with no modification. We propose in this paper to extend preunification by allowing projection on flexible-flexible pairs for systematically generating proofs for a given proposition.

In order to use higher-order unification as a theorem proving procedure, we have to depart from simple type theory. As Barendregt neatly describes in [1], higher-order type systems can be classified by three dimensions in what he calls *λ-cube.* Simple type theory is the simplest system in λ-cube and called $\lambda\rightarrow$ (lambda arrow). Under the most powerful system in λ-cube, called λC or CC (*Calculus of Constructions* [2]), we can define types depending on individuals and types. Such types are called *dependent types* in general. The smallest subsystem of CC having types dependent on individuals is called λP or LF (*Logical Framework* [4]). In LF or in CC, by the *Curry-Howard-de Bruijn isomorphism* [6], formulas can be represented by types and proofs by terms.

Preunification was recently extended to LF by Elliott [3] and is also used as the unification procedure for Elf. The essence of the extension is to unify types of terms as well as terms themselves. One of the most important results of the extension is the possibility of unification between terms representing proofs.

Given a type representing a proposition, we can systematically generate terms representing proofs of the proposition using the unification procedure for LF, provided that the procedure allows projection on flexible-flexible pairs. Particularly, SLD-resolution of Prolog can be completely simulated by high-order unification for LF.

As a theorem proving procedure, higher-order unification is less efficient than the Prolog interpreter because it explicitly constructs proofs, which correspond to program traces in Prolog. This drawback, however, can be overcome by program transformation. Using the standard *unfold/fold transformation rules* for Prolog programs [12], we can transform the unification procedure written in Prolog into the Prolog interpreter itself. During the transformation process, the code for constructing proofs is stripped out and the code for computing constraints among variables is extracted.

Outline of the paper: In Section 2, we briefly explain higher-order unification for $\lambda\rightarrow$ and LF. In Section 3, we define LF and its unification procedure with the extension of allowing projection on flexible-flexible pairs.

Section 4 is the most important part of the paper. We show that the extended unification procedure, with a slight modification, can completely simulate SLD-resolution of Prolog. In Section 5, we formally prove that higher-order unification can simulate SLD-resolution by transforming the unification procedure into Prolog interpreter using the unfold/fold transformation rules for Prolog.

Section 6 is devoted to conclusions and related works.

2 Introduction to Higher-order Unification

Higher-order unification is a procedure for nondeterministically generating all the possible substitutions for functional variables. As an example in higher-order unification, consider the following unification problem

$$\langle f0,\ 0+0\rangle,$$

where f is a free variable, $+$ a binary function, written in infix notation, and 0 a constant. In this problem, we should match the pattern (term containing free variables) $f0$ against the closed term $0+0$ and obtain a substitution for the free variable f. The problem is called *higher-order* because the free variable f denotes a function, i.e., the type of f is $\mathbf{N}\rightarrow\mathbf{N}$, where $\mathbf{N}$ is the type of natural numbers.

By the *imitation* operation, we substitute for f the following term:

$$f \leftarrow \lambda x.(f_1 x)+(f_2 x),$$

where f_1 and f_2 are new free variables of type $\mathbf{N}\rightarrow\mathbf{N}$ and x is a bound variable of type $\mathbf{N}$. This operation imitates the outermost function symbol $+$ of the given term $0+0$. The pair $\langle f0, 0+0\rangle$ then becomes

$$\langle (f_1 0)+(f_2 0),\ 0+0\rangle$$

after β-reduction. By the *decomposition* (or *simplification*) operation, the pair is decomposed into two pairs:

$$\langle f_1 0,\ 0\rangle,\ \langle f_2 0,\ 0\rangle.$$

The decomposition rule for $+$ is, in general, of the form

$$\langle M_1 + M_2,\ N_1 + N_2\rangle \implies \langle M_1,\ N_1\rangle,\ \langle M_2,\ N_2\rangle.$$

Using the *projection* operation on the pair $\langle f_1 0, 0\rangle$, we can substitute for f_1 the following term:

$$f_1 \leftarrow \lambda x.x.$$

Note that the term $\lambda x.x$ represents a function that does projection on its first (and only) argument. The pair $\langle f_1 0, 0\rangle$ then becomes $\langle 0, 0\rangle$, which immediately disappears by the decomposition operation because the term 0 does not have any argument. If we use the imitation operation on the pair $\langle f_2 0, 0\rangle$, we can substitute for f_2 the term

$$f_2 \leftarrow \lambda x.0.$$

This completes the unification process and we have substituted for f the term

$$f \leftarrow \lambda x.x + 0.$$

Depending on whether to do imitation or projection for f_1 and f_2, we can nondeterministically obtain the following four substitutions for f:

$$\begin{aligned} f &\leftarrow \lambda x.0 + 0 \\ f &\leftarrow \lambda x.x + 0 \\ f &\leftarrow \lambda x.0 + x \\ f &\leftarrow \lambda x.x + x. \end{aligned}$$

Unlike imitation, the projection operation only depends on the pattern and not on the form of the term against which the pattern should be matched. As an example, consider free variable f of type $\mathbf{N} \to (\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$ and term

$$\lambda z.\lambda s.fzs,$$

where the type of z is $\mathbf{N}$ and the type of s is $\mathbf{N} \to \mathbf{N}$. This term is the so-called *long normal form* of f. As a unification problem, we consider the pair of two copies of $\lambda z.\lambda s.fzs$, i.e.,

$$\langle \lambda z.\lambda s.fzs,\ \lambda z.\lambda s.fzs\rangle.$$

By the projection operation on the first argument z in fzs, we obtain the substitution

$$f \leftarrow \lambda z.\lambda s.z.$$

On the other hand, by the projection operation on the second argument s, we obtain the substitution

$$f \leftarrow \lambda z.\lambda s.s(f_1 zs),$$

and the pair

$$\langle \lambda z.\lambda s.f_1 zs,\ \lambda z.\lambda s.f_1 zs\rangle$$

after decomposition, where f_1 is a new free variable and is also of type $\mathbf{N} \to (\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$. By the projection operation on the first argument of f_1, we have

$$f_1 \leftarrow \lambda z.\lambda s.z,$$

i.e, for f, we have obtained

$$f \leftarrow \lambda z.\lambda s.sz.$$

Depending on the argument on which to do projection, we can successively obtain the following substitutions for f:

$$\begin{aligned} f &\leftarrow \lambda z.\lambda s.z \\ f &\leftarrow \lambda z.\lambda s.sz \\ f &\leftarrow \lambda z.\lambda s.s(sz) \\ f &\leftarrow \lambda z.\lambda s.s(s(sz)) \\ &\cdots \end{aligned}$$

Note that these are all the closed (normal) terms of type $\mathbf{N} \to (\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$ that do not contain constants and are known as *Church's numerals*. This example shows how the projection operation can serve as a generator of closed terms for a given type.

LF (Logical Framework) is a higher-order type system in which a type may depend on individuals. For example, an array f of natural numbers whose length is n is declared in LF as follows:

$$f : \texttt{Narray}\ n,$$

where $\texttt{Narray}\ n$ denotes the type of an array of natural numbers whose length is n. Note that the type $\texttt{Narray}\ n$ depends on the individual n of type $\mathbf{N}$. Let `zeroarray` be a function such that $\texttt{zeroarray}\ x$ returns an array of length x whose elements are all 0. Using `Narray`, we can declare the type of `zeroarray` as follows:

$$\texttt{zeroarray} : \Pi x{:}\mathbf{N}.\texttt{Narray}\ x.$$

The type $\Pi x{:}A.B$ denotes the function space (indexed product)

$$\prod_{x \in A} B,$$

where B may depend on the element x of A. If B does not depend on x, $\Pi x{:}A.B$ simply denotes the function space $A \to B$, i.e., $A \to B$ is considered to be an abbreviation of $\Pi x{:}A.B$ if x does not occur free in B.

Assume that f and n are both free variables and consider the unification problem

$$\langle f,\ \texttt{zeroarray}\ 3\rangle.$$

Solving this problem, we not only obtain

$$f \leftarrow \texttt{zeroarray}\ 3,$$

but also

$$n \leftarrow 3,$$

because we have to unify the types of f and $\texttt{zeroarray}\ 3$, i.e., $\texttt{Narray}\ n$ and $\texttt{Narray}\ 3$. This is the essence of the unification procedure for LF; we unify types of terms as well as terms themselves.

3 Higher-order Unification for LF

In this section, we give the formal definition of LF and its unification procedure. LF consists of three layers of expressions: kinds, types and terms. Types are denoted by A, B, etc., terms by M, N, etc. and kinds by K, etc. In this paper, we adopt a variant of LF having the kind of propositions, denoted by Prop, in addition to the kind of types Type. Kinds, types and terms are defined as follows:

$$K ::= \text{Type} \mid \text{Prop} \mid \Pi x{:}A.K$$

$$A ::= x \mid \Pi x{:}A.B \mid AM$$

$$M ::= x \mid \lambda x{:}A.M \mid MN$$

By an abuse of notation, however, we allow M, N, etc. to denote types and A, B, etc. to denote kinds.

As in untyped λ-calculus, variable x is said to be *bound in* term $\lambda x{:}A.M$; x is also bound in type $\Pi x{:}A.B$. A *context* is a sequence of pairs of a variable and its type of the form $x{:}A$ and denoted by Γ, Δ, etc. A concatenation of contexts is written with a comma as in $\Gamma, x{:}A, \Delta$ and the empty context is denoted by ( ). We say that x is *declared in* Γ if $x{:}A$ is a member of the sequence Γ for some A, and use the expression $V(\Gamma)$ to denote the set of all the variables declared in Γ. If $\Gamma = \Gamma_1, x{:}A, \Gamma_2$ and x is not declared in Γ_2, we use the expression $\Gamma(x)$ to denote A. If $\Gamma = x_1{:}A_1, \cdots, x_n{:}A_n$, we use the expressions $\lambda\Gamma$ and $\Pi\Gamma$ to denote $\lambda x_1{:}A_1.\cdots.\lambda x_n{:}A_n$ and $\Pi x_1{:}A_1.\cdots.\Pi x_n{:}A_n$, respectively.

The β- and η-reduction rules are defined as in untyped λ-calculus. We have

$$(\lambda x{:}A.M)N \rightarrow_\beta [N/x]M,$$

where $[N/x]$ denotes the substitution that maps x to N, and

$$\lambda x{:}A.Mx \rightarrow_\eta M,$$

where x does not occur free in M. We write $M \approx_{\beta\eta} N$ if M and N are $\beta\eta$-convertible to each other. The α-conversion is implicit in this paper.

There are four kinds of judgement: "Γ context" means that Γ is a valid context, "$\Gamma \vdash K \in \text{Kind}$" means that K is a valid kind under Γ, "$\Gamma \vdash A \in K$" means that A is a type of kind K under Γ, and "$\Gamma \vdash M \in A$" means that M is a term of type A under Γ. These judgements are defined by the following inference rules, in which K_0 and K_0' denote either Type or Prop.

$$\frac{}{(\,)\ \text{context}}$$

$$\frac{\Gamma \vdash K \in \text{Kind}}{\Gamma, x{:}K\ \text{context}}$$

$$\frac{\Gamma \vdash A \in K_0}{\Gamma, x{:}A\ \text{context}}$$

$$\frac{\Gamma\ \text{context}}{\Gamma \vdash \text{Type} \in \text{Kind}}$$

$$\frac{\Gamma\ \text{context}}{\Gamma \vdash \text{Prop} \in \text{Kind}}$$

$$\frac{\Gamma\ \text{context} \quad x \in V(\Gamma)}{\Gamma \vdash x \in \Gamma(x)}$$

$$\frac{\Gamma \vdash A \in K_0 \quad \Gamma, x{:}A \vdash K \in \text{Kind}}{\Gamma \vdash \Pi x{:}A.K \in \text{Kind}}$$

$$\frac{\Gamma \vdash A \in K_0 \quad \Gamma, x{:}A \vdash B \in K_0'}{\Gamma \vdash \Pi x{:}A.B \in K_0'}$$

$$\frac{\Gamma \vdash A \in K_0 \quad \Gamma, x{:}A \vdash M \in B \quad \Gamma, x{:}A \vdash B \in K_0'}{\Gamma \vdash \lambda x{:}A.M \in \Pi x{:}A.B}$$

$$\frac{\Gamma \vdash M \in \Pi x{:}A.B \quad \Gamma \vdash N \in A}{\Gamma \vdash MN \in [M/x]B}$$

$$\frac{\Gamma \vdash M \in A \quad A \approx_{\beta\eta} A' \quad \Gamma \vdash A' \in K_0}{\Gamma \vdash M \in A'}$$

$$\frac{\Gamma \vdash A \in K \quad K \approx_{\beta\eta} K' \quad \Gamma \vdash K' \in \text{Kind}}{\Gamma \vdash A \in K'}$$

In the second and third rules, we assume that x is not declared in Γ so that variables in a valid context are always distinct.

Definition 3.1 (atomic type) *If x is a variable and $\Gamma(x) = \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.$Type or $\Gamma(x) = \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.$Prop then $xM_1\cdots M_m$ is called an atomic type under Γ. By an abuse of terminology,* Type *and* Prop *are also called atomic types.*

In the following discussions, we fix a context Γ_Σ such that

$$\Gamma_\Sigma = \Gamma_c, \Gamma_f, \quad V(\Gamma_c) \cap V(\Gamma_f) = \emptyset.$$

Variables declared in Γ_c are called *constants* and variables in Γ_f *free variables.* We further assume that $\Gamma_\Sigma \vdash \Gamma_f(f) \in \text{Type}$ or $\Gamma_\Sigma \vdash \Gamma_f(f) \in \text{Prop}$ for each $f \in V(\Gamma_f)$; i.e., we do not have free type variables.

Definition 3.2 (equation, unification problem and unifier) *An equation is an unordered pair of terms or types. A unification problem is a multiset U_0 of equations such that if $\langle M, N\rangle \in U_0$ then*

$$\Gamma_\Sigma \vdash M \in A, \quad \Gamma_\Sigma \vdash N \in B$$

for some A and B, and $\langle A, B\rangle \in U_0$ unless $A = B$. A unifier of U_0 is a substitution ϕ for free variables in Γ_f such that for each $f \in V(\Gamma_f)$, $\Gamma_c \vdash \phi(f) \in \phi(\Gamma_f(f))$ and $\phi(M) \approx_{\beta\eta} \phi(N)$ for each $\langle M, N\rangle \in U_0$.

As is described in [11], unification procedures including higher-order unification can be formulated as a set of transformation rules on a multiset of equations. We use U, U', etc. to denote a multiset of equations. For example, let

$$U' = \{\langle aM_1\cdots M_m,\ aN_1\cdots N_m\rangle\} \cup U,$$

where a is a constant and $\cup$ denotes the union operator for multisets. The *decomposition* rule can then be formulated as follows:

$$U' \implies U'',$$

where

$$U'' = \{\langle M_i,\ N_i\rangle \mid 1 \le i \le m\} \cup U.$$

More formally, the unification procedure is defined as a set of transformation rules on a *triple* of a multiset of equations, a substitution and a context of free variables. A context is included in a triple for recording the type of each new free variable. It is initialized with Γ_f and modified as free variables are newly introduced. If the substitution and the context of a triple are not changed under a transformation rule, they are not written explicitly; i.e., in a transformation rule of the form $U \Rightarrow U'$, the substitution and the context of a triple are not changed. In such cases, the unchanged substitution is denoted by θ and the unchanged context of free variables by Γ.

Definition 3.3 (flexible and rigid)
Let $M = \lambda\Delta.hM_1\cdots M_m$ be such that $(\Gamma_c, \Gamma, \Delta)(h) = \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.A_0$ and A_0 is an atomic type under Γ_c. If h is a variable (or a constant), M is called a long head normal form and h the head of M. If h is a constant in Γ_c or a variable declared in Δ then M is called rigid, and if h is a free variable in Γ then M is called flexible.

The unification procedure for LF, which allows projection on flexible-flexible pairs, is presented below. In the following descriptions, we assume that Δ and Δ' are of the same length and the sequence of variables declared in Δ is the same as that of Δ'.

(head-β-reduction)

$$\begin{aligned} &\{\langle\lambda\Delta.(\lambda x.M)M_0M_1\cdots M_m,\ N\rangle\} \cup U \\ \implies\ &\{\langle\lambda\Delta.([M_0/x]M)M_1\cdots M_m,\ N\rangle\} \cup U. \end{aligned}$$

(head-inverse-η-reduction)

$$\begin{aligned} &\{\langle\lambda\Delta.aM_1\cdots M_m,\ N\rangle\} \cup U \\ \implies\ &\{\langle\lambda\Delta.\lambda x{:}A.aM_1\cdots M_m x,\ N\rangle\} \cup U, \end{aligned}$$

where a is a functional variable taking more than m arguments, i.e.,

$$(\Gamma_c, \Gamma, \Delta)(a) = \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.\Pi x{:}A.B.$$

(decomposition)

$$\begin{aligned} &\{\langle\lambda\Delta.aM_1\cdots M_m,\ \lambda\Delta'.aN_1\cdots N_m\rangle\} \\ &\cup U \\ \implies\ &\{\langle\lambda\Delta.M_i,\ \lambda\Delta'.N_i\rangle \mid 1 \le i \le m\} \cup U, \end{aligned}$$

where a is a constant in Γ_c or a variable declared in Δ such that

$$(\Gamma_c, \Delta)(a) = \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.C$$

and C is an atomic type under Γ_c (or C = Type or C = Prop).

(type decomposition)

$$\begin{aligned} &\{\langle\lambda\Delta.\Pi x{:}A.B,\ \lambda\Delta'.\Pi x{:}A'.B'\rangle\} \cup U \\ \implies\ &\{\langle\lambda\Delta.A,\ \lambda\Delta'.A'\rangle, \\ &\ \langle\lambda\Delta.\lambda x{:}A.B,\ \lambda\Delta'.\lambda x{:}A'.B'\rangle\} \cup U. \end{aligned}$$

(imitation) Let $\langle\lambda\Delta.fM_1\cdots M_m,\ \lambda\Delta'.aN_1\cdots N_n\rangle$ be a pair in U, where f is a free variable in Γ and a is a constant in Γ_c. We further assume

$$\begin{aligned} \Gamma &= \Gamma_1, f{:}F, \Gamma_2 \\ F &= \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.A_0 \\ \Gamma_c(a) &= \Pi y_1{:}B_1.\cdots.\Pi y_n{:}B_n.B_0, \end{aligned}$$

where A_0 and B_0 are atomic types under Γ_c. Let L be the following term:

$$\begin{aligned} &\lambda x_1{:}A_1.\cdots.\lambda x_m{:}A_m. \\ &a(f_1x_1\cdots x_m)\cdots(f_nx_1\cdots x_m), \end{aligned}$$

where f_i are new free variables. We then have

$$\begin{aligned} U,\ \theta,\ \Gamma \implies\ &[L/f]U \cup \{\langle A', B'\rangle\},\ [L/f]\circ\theta, \\ &(\Gamma_1, f_1{:}F_1, \cdots, f_n{:}F_n, [L/f]\Gamma_2), \end{aligned}$$

where

$$\begin{aligned} A' &= \lambda x_1{:}A_1.\cdots.\lambda x_m{:}A_m.A_0 \\ B' &= \lambda x_1{:}A_1.\cdots.\lambda x_m{:}A_m. \\ &\quad [f_1x_1\cdots x_m/y_1, \cdots, f_nx_1\cdots x_m/y_n]B_0 \\ F_i &= \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m. \\ &\quad [f_1x_1\cdots x_m/y_1, \cdots, f_{i-1}x_1\cdots x_m/y_{i-1}]B_i \end{aligned}$$

for $1 \le i \le n$. $[L/f]U$ denotes the result of replacing f with L in U. $[L/f]\circ\theta$ denotes the composition of the substitutions $[L/f]$ and θ. $[L/f]\Gamma_2$ denotes the result of replacing f with L in Γ_2.

(projection) Let $\langle\lambda\Delta.fM_1\cdots M_m,\ N\rangle$ be a pair in U, where f is a free variable in Γ. We further assume

$$\begin{aligned} \Gamma &= \Gamma_1, f{:}F, \Gamma_2 \\ F &= \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.A_0 \\ A_p &= \Pi y_1{:}B_1.\cdots.\Pi y_k{:}B_k.B_0 \end{aligned}$$

for some p such that $1 \le p \le m$, where A_0 and B_0 are atomic types under Γ_c. Let L be the following term:

$$\begin{aligned} &\lambda x_1{:}A_1.\cdots.\lambda x_m{:}A_m. \\ &x_p(f_1x_1\cdots x_m)\cdots(f_kx_1\cdots x_m), \end{aligned}$$

where f_i are new free variables. We then have

$$U,\ \theta,\ \Gamma \implies [L/f]U \cup \{\langle A', B'\rangle\},\ [L/f]\circ\theta,\ (\Gamma_1, f_1{:}F_1, \cdots, f_k{:}F_k, [L/f]\Gamma_2),$$

where

$$\begin{aligned} A' &= \lambda x_1{:}A_1.\cdots.\lambda x_m{:}A_m.A_0 \\ B' &= \lambda x_1{:}A_1.\cdots.\lambda x_m{:}A_m. \\ &\quad [f_1 x_1 \cdots x_m/y_1, \cdots, f_k x_1 \cdots x_m/y_k]B_0 \\ F_i &= \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m. \\ &\quad [f_1 x_1 \cdots x_m/y_1, \cdots, f_{i-1} x_1 \cdots x_m/y_{i-1}]B_i \end{aligned}$$

for $1 \le i \le k$.

Notice that in the course of higher-order unification, expressions that are neither terms nor types in LF may appear. In **(imitation)** or **(projection)**, we add expressions of the form $\lambda x_1{:}A_1.\cdots.\lambda x_m{:}A_m.A_0$, where A_0 is an atomic type. Such expressions are decomposed into terms by **(decomposition)**. See also **(type decomposition)**.

Assume that Γ_c contains only type constants, i.e., $\Gamma_c \vdash \Gamma_c(c) \in \mathrm{Kind}$ for each $c \in V(\Gamma_c)$. Under this assumption, the rule **(imitation)** becomes useless, and we have the following proposition.

Proposition 3.1 (ground completeness) *If $\langle f, f\rangle \in U_0$ for each $f \in V(\Gamma_f)$ and ϕ is a unifier of U_0 then there exists a sequence of triples*

$$U_0, [\,], \Gamma_f \Rightarrow \cdots \Rightarrow \{\,\}, \theta, (\,)$$

such that $\phi = \theta|_V$, where $V = V(\Gamma_f)$.

$\{\,\}$ denotes an empty multiset, $[\,]$ an empty substitution and $(\,)$ an empty context. $\theta|_V$ denotes the substitution η such that $\eta(f) = \theta(f)$ if $f \in V$ and $\eta(f) = f$ otherwise. The proposition is proved by the existence of long normal forms in LF; i.e., every well-typed term in LF can be reduced into a unique long normal form. Therefore we can assume that $\phi(f)$ is in a long normal form for each free variable f in Γ_f. If we use ϕ to guide the application of transformation rules, we can reach in a finite number of steps an empty multiset. The proposition is simpler than the corresponding one in [3] because flexible-flexible pairs are also selected.

Conversely, we have

Proposition 3.2 (gound soundness) *If $\langle f, f\rangle \in U_0$ for each $f \in V(\Gamma_f)$ and there exists a sequence of triples*

$$U_0, [\,], \Gamma_f \Rightarrow \cdots \Rightarrow \{\,\}, \theta, (\,)$$

then $\theta|_V$ is a unifier of U_0, where $V = V(\Gamma_f)$.

4 Higher-order Unification as a Theorem Proving Procedure

Proposition 3.1 is the basis for using higher-order unification as a theorem proving procedure. Let Γ_c be a context of type constants and Δ a context of constants. Let A be a type such that

$$\Gamma_c, \Delta \vdash A \in \mathrm{Prop}.$$

Since the problem of proving A is that of finding a term M satisfying

$$\Gamma_c, \Delta \vdash M \in A,$$

we can prove A by solving the following unification problem:

$$U_0 = \{\langle f, f\rangle\}, \quad \Gamma_f = f : \Pi\Delta.A.$$

By Proposition 3.1, if A has proof M, we can find a unifier ϕ such that $\phi(f) \approx_{\beta\eta} \lambda\Delta.M$.

Let us try to solve a query in Prolog under our framework. Consider as an example the following Prolog program:

```
append(nil,X,nil).
append(cons(E,X),Y,cons(E,Z)) :-
    append(X,Y,Z).
```

This program has three constants: `nil`, `cons` and `append`.

Let Γ_c be the following context:

$$\mathbf{N}{:}\mathrm{Type},\ \mathtt{Nlist}{:}\mathrm{Type},\ \mathtt{ok}{:}\mathrm{Prop},$$
$$\mathtt{append}{:}(\mathtt{Nlist} \to \mathtt{Nlist} \to \mathtt{Nlist} \to \mathrm{Prop}),$$

where `Nlist` denotes the type of a list of natural numbers and `append` is a tertiary predicate on `Nlist`. `ok` is a proposition denoting the success of a Prolog query.

Let Δ' be the following context:

$$0{:}\mathbf{N},\ 1{:}\mathbf{N},\ 2{:}\mathbf{N},$$
$$\mathtt{nil}{:}\mathtt{Nlist},\ \mathtt{cons}{:}(\mathbf{N} \to \mathtt{Nlist} \to \mathtt{Nlist}),$$
$$b{:}B,\ s{:}S,$$

where B and S are defined as follows:

$$\begin{aligned} B &= \Pi x{:}\mathtt{Nlist}.\mathtt{append}\ \mathtt{nil}\ x\ x \\ S &= \Pi e{:}\mathbf{N}.\Pi x{:}\mathtt{Nlist}\Pi y{:}\mathtt{Nlist}\Pi z{:}\mathtt{Nlist}. \\ &\quad \mathtt{append}\ x\ y\ z \\ &\quad \to \mathtt{append}\ (\mathtt{cons}\ e\ x)\ y\ (\mathtt{cons}\ e\ z). \end{aligned}$$

$b{:}B$ corresponds to the first clause of `append` (base case) and $s{:}S$ to the second clause of `append` (recursion step).

Consider the following query for the above Prolog program:

```
?- append(X,X,cons(1,cons(1,nil))).
```

In order to answer this query, let

$$\Delta = \Delta', a{:}A,$$

where

$$A = \Pi x{:}\mathtt{Nlist}.\ \mathtt{append}\ x\ x\ (\mathtt{cons}\ 1\ (\mathtt{cons}\ 1\ \mathtt{nil})) \to \mathtt{ok}.$$

We then declare free variable f : $\Pi\Delta.\mathbf{ok}$. Finally let $U_0 = \{\langle f, f\rangle\}$. Since a is the only variable that can be the head of a term of type ok, if there exists a unifier ϕ of U_0, $\phi(f)$ must be of the form

$$\phi(f) \;=\; \lambda\Delta.aMN,$$

where

$$\begin{array}{lll}\Gamma_c, \Delta & \vdash & M \in \mathtt{Nlist} \\ \Gamma_c, \Delta & \vdash & N \in \mathtt{append}\ M\ M\ (\mathtt{cons}\ 1\ (\mathtt{cons}\ 1\ \mathtt{nil})).\end{array}$$

M is the answer for X in the above Prolog query and N is the justification proof for the success of the query.

By Proposition 3.1, the unification procedure for LF is guaranteed to find the unifier ϕ. However the procedure is far more redundant and inefficient than the ordinary SLD-resolution procedure of Prolog because it not only tries to generate proofs but also tries to directly generate answers to queries. In the above example, it tries to find the term M by generating closed terms consisting of 0, 1, 2, nil and cons, such as cons 0 (cons 0 (cons 0 nil)). In Prolog, on the other hand, the answer to a query is obtained from the structure of the proof by first-order unification.

Consider the previous example again. In order to directly simulate SLD-resolution of Prolog, let Γ_c be the following context:

$$\begin{array}{l}\mathtt{N}{:}\mathrm{Type},\ \mathtt{Nlist}{:}\mathrm{Type}, \\ \mathtt{append}{:}(\mathtt{Nlist} \to \mathtt{Nlist} \to \mathtt{Nlist} \to \mathrm{Prop}), \\ 0{:}\mathtt{N},\ 1{:}\mathtt{N},\ 2{:}\mathtt{N}, \\ \mathtt{nil}{:}\mathtt{Nlist},\ \mathtt{cons}{:}(\mathtt{N} \to \mathtt{Nlist} \to \mathtt{Nlist})\end{array}$$

and Δ be

$$b{:}B,\ s{:}S,$$

where B and S are defined as before. Γ_f now consists of the following two free variables:

$$\begin{array}{lcl}X & : & \Pi\Delta.\mathtt{Nlist} \\ f & : & \Pi\Delta.\mathtt{append}\ (Xbs)\ (Xbs)\ (\mathtt{cons}\ 1\ (\mathtt{cons}\ 1\ \mathtt{nil})).\end{array}$$

The free variable X corresponds to the variable X in the query. Let $U_0 = \{\langle X, X\rangle\ \langle f, f\rangle\}$. Note that $\langle X, X\rangle$ and $\langle f, f\rangle$ immediately become $\langle \lambda\Delta.Xbs, \lambda\Delta.Xbs\rangle$ and $\langle \lambda\Delta.fbs, \lambda\Delta.fbs\rangle$ by **(head-inverse-η-reduction)**. Since Δ consists of only variables corresponding to Prolog clauses, the procedure is now expected to work exactly as SLD-resolution of Prolog. Unfortunately, however, it still tries to directly generate the term for X and in this case, worse than before, it immediately fails because 1, nil and cons are not in Δ but in Γ_c. We therefore have to make a distinction between X and f.

Definition 4.1 *Let*

$$\begin{array}{rcl}(\Gamma_c, \Gamma, \Delta)(h) & = & \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.cN_1\cdots N_n \\ \Gamma_c(c) & = & \Pi y_1{:}B_1.\cdots.\Pi y_n{:}B_n.K.\end{array}$$

Term $\lambda\Delta.hM_1\cdots M_m$ is called an individual term if K = Type and a proof term if K = Prop. By an abuse of terminology, $\lambda\Delta.hM_1\cdots M_m$ is also called individual if $\Gamma_c(h) = \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.$Type or $\Gamma_c(h) = \Pi x_1{:}A_1.\cdots.\Pi x_m{:}A_m.$Prop.

Note that $\lambda\Delta.Xbs$ is an individual term and $\lambda\Delta.fbs$ a proof term.

In order to completely simulate SLD-resolution of Prolog, we modify the unification procedure as follows: Projection is allowed on a flexible-flexible pair only if it is a pair of proof terms. We further assume the following priority for selecting an equation from a unification problem:

1. rigid-rigid pairs
2. flexible-rigid pairs of individual terms
3. flexible-flexible pairs of proof terms.

In case of simulating Prolog, we further restrict the procedure so that only imitation is applied on flexible-rigid individual pairs.

As for the example, we can successively obtain the following substitutions for f:

$$\begin{array}{lcl}f & \leftarrow & \lambda\Delta.fbs \\ f & \leftarrow & \lambda\Delta.s(\cdots)(\cdots)(\cdots)(\cdots)(f_1 sb) \\ & \cdots & \\ f & \leftarrow & \lambda\Delta.s(\cdots)(\cdots)(\cdots)(\cdots)(b(\cdots)) \\ & \cdots & \\ f & \leftarrow & \lambda\Delta.s\ 1\ \mathtt{nil}\ (\mathtt{cons}\ 1\ \mathtt{nil})(\mathtt{cons}\ 1\ \mathtt{nil}) \\ & & \qquad (b(\mathtt{cons}\ 1\ \mathtt{nil}))\end{array}$$

Note that selection among clauses exactly corresponds to selection among arguments (b or s) in projection. The uninstantiated individual terms are denoted by $(\cdots)$ and they are instantiated by imitation on flexible-rigid pairs.

If projection is not allowed on flexible-flexible individual pairs, the procedure is not guaranteed to generate a unifier but may leave some flexible-flexible pairs as constraints. In case of simulating Prolog, we can immediately obtain a unifier from the constraints.

5 Program Transformation

Let P be a Prolog program and $?{-}A_1, \ldots, A_n$ a Prolog query under P. As in the example in the previous section, we construct contexts Γ_c and Δ from P. Let $\bar{x}$ be the sequence of variables declared in Δ. For each variable X in $?{-}A_1, \ldots, A_n$, we declare free variable X of type $\Pi\Delta.c$, where c is the data type of X in P, and for each atom A_i in the query, we declare free variable f_i of type $\Pi\Delta.A_i'$, where A_i' is the result of substituting term $X\bar{x}$ for each X. These free variables comprise the context Γ_f. Finally let

$$\begin{array}{rcl}U_0 & = & \{\langle \lambda\Delta.X\bar{x},\ \lambda\Delta.X\bar{x}\rangle \mid \\ & & \quad \mathtt{X}\text{ is a variable in the query}\} \\ & & \cup\ \{\langle \lambda\Delta.f_i\bar{x},\ \lambda\Delta.f_i\bar{x}\rangle \mid 1 \le i \le n\}.\end{array}$$

We define the following functions on a triple $T = \langle U, \theta, \Gamma \rangle$ that appears in higher-order unification:

$$\begin{aligned} goal(T) &= \{A \mid \langle \lambda\Delta.f\bar{x},\ \lambda\Delta.f\bar{x}\rangle \in U, \\ &\qquad \lambda\Delta.f\bar{x} : \text{proof},\ \Gamma(f) = \Pi\Delta.A\} \\ constraint(T) &= \{\langle M,\ N\rangle \mid \langle \lambda\Delta.M,\ \lambda\Delta.N\rangle \in U, \\ &\qquad \lambda\Delta.M : \text{individual}\} \\ subst(T) &= \theta|_{V'}, \end{aligned}$$

where $V' = \{f \in V(\Gamma) \mid \lambda\Delta.f\bar{x} : \text{individual}\}$. $goal(T)$ denotes the multiset of unsolved goals. $constraint(T)$ denotes the multiset of constraints between individual terms. $subst(T)$ denotes the substitution for individual variables. We further define

$$gcs(T) = \langle goal(T),\ constraint(T),\ subst(T)\rangle.$$

Starting with the triple $T_0 = \langle U_0, [\,], \Gamma_f\rangle$, the unification procedure nondeterministically computes a sequence of triples

$$T_0 \Rightarrow T_1 \Rightarrow \cdots \Rightarrow T_i \Rightarrow \cdots.$$

From the relation $\Rightarrow$, we want to derive a relation $\rightsquigarrow$ between triples $t_i = gcs(T_i)$ and show that $\rightsquigarrow$ is an implementation of SLD-resolution. In order to accomplish our goal, however, we cannot use the relation $\Rightarrow$ directly but should define a new relation consisting of transformation rules that are results of chunking those of $\Rightarrow$.

The new relation is denoted by $\Rightarrow^\dagger$ and consists of the following two rules:

1. Apply **(projection)** on a flexible-flexible pair of proof terms of the form $\langle\lambda\Delta.f\bar{x}, \lambda\Delta.f\bar{x}\rangle$. Apply **(head-$\beta$-reduction)** successively on the pair. Finally apply **(decomposition)** on the pair.

2. Apply **(imitation)** on a flexible-rigid pair of individual terms. Apply **(head-β-reduction)** successively on the pair. Finally apply **(decomposition)** on the pair.

Using $\Rightarrow^\dagger$, we define $\rightsquigarrow$ such that $t \rightsquigarrow t'$ if and only if $t = gcs(T)$, $t' = gcs(T')$ and $T \Rightarrow^\dagger T'$, where $T_0 \Rightarrow^\dagger \cdots \Rightarrow^\dagger T$.

We formalize the derivation of $\rightsquigarrow$ from $\Rightarrow^\dagger$ using the framework of program transformation in Prolog. We define $\Rightarrow^\dagger$ and $\rightsquigarrow$ in Prolog and transform the definition of $\rightsquigarrow$ into a form that does not refer to $\Rightarrow^\dagger$.

In the following Prolog programs, $\bar{x}$, $\bar{y}$, etc. denote a list of variables and $\bar{M}$, $\bar{N}$, etc. a list of terms. Contexts are represented by lists of expressions of the form $x{:}A$. The Prolog programs contain some computable functions. If Γ is a context, M a term and $\bar{M}$ a list of terms then the expression $\lambda\Gamma.M\bar{M}$ denotes a term in which M is applied to $\bar{M}$ and abstracted by Γ. We use $l_1@l_2$ to denote the concatenation of lists l_1 and l_2, $l[i]$ the i-th element of list l and $|l|$ the length of l. Multisets of equations are represented by lists of pairs of the form $\langle M, N\rangle$, where $\langle M, N\rangle = \langle N, M\rangle$.

The relation $\Rightarrow^\dagger$, denoted by predicate $transform(U, \theta, \Gamma, U', \theta', \Gamma')$, is defined as follows:

```
transform(U, θ, Γ, U', θ', Γ') :-
    U = U1@[⟨M, M⟩]@U2,
    proof(M, Γ), M = λΔ.fx̄
    project(M, Γ,
        L, f, Φ, λΓA.A0, λΓA.B0'),
    head_beta*([L/f]M, M'),
    decompose(M', M', U''),
    U' = [L/f](U1@U2)@[⟨λΓA.A0, λΓA.B0'⟩]@U'',
    θ' = [L/f] ∘ θ,
    Γ = Γ1@[f:F]@Γ2,
    Γ' = Γ1@Φ@[L/f]Γ2.
transform(U, θ, Γ, U', θ', Γ') :-
    U = U1@[⟨M, N⟩]@U2,
    individual(M, Γ),
    imitate(M, N, Γ,
        L, f, Φ, λΓA.A0, λΓA.B0'),
    head_beta*([L/f]M, M'),
    decompose(M', M', U''),
    U' = [L/f](U1@U2)@[⟨λΓA.A0, λΓA.B0'⟩]@U'',
    θ' = [L/f] ∘ θ,
    Γ = Γ1@[f:F]@Γ2,
    Γ' = Γ1@Φ@[L/f]Γ2.
imitate(M, N, Γ, L, f, Φ, λΓA.A0, λΓA.B0') :-
    M = λΓM.fM̄, Γ(f) = ΠΓA.A0,
    N = λΓN.aN̄, Γc(a) = ΠΓB.B0,
    context_vars(ΓA, x̄),
    make_new_free_vars(ΓA, x̄, ΓB, B0, Φ, B0'),
    apply_context_vars(Φ, x̄, L̄),
    L = λΓA.aL̄.
project(M, Γ, L, f, Φ, λΓA.A0, λΓA.B0') :-
    M = λΓM.fM̄, Γ(f) = ΠΓA.A0,
    xp = x̄[p], ΓA[p] = ΠΓB.B0,
    context_vars(ΓA, x̄),
    make_new_free_vars(ΓA, x̄, ΓB, B0, Φ, B0'),
    apply_context_vars(Φ, x̄, L̄),
    L = λΓA.xpL̄.
context_vars([ ], [ ]).
context_vars([x:A|ΓA], [x|x̄]) :-
    context_vars(ΓA, x̄).
make_new_free_vars(ΓA, x̄, [ ], B0, [ ], B0).
make_new_free_vars(ΓA, x̄,
        [y:B|ΓB], B0,
        [f:ΠΓA.B|Φ], B0') :-
    get_new_free_var(f),
    make_new_free_vars(ΓA, x̄,
```

$[f\bar{x}/y]\Gamma_B,\ [f\bar{x}/y]B_0,\ \Phi,\ B_0').$

$apply\_context\_vars([\,],\ \bar{x},\ [\,]).$

$apply\_context\_vars([f{:}F|\Phi],\ \bar{x},\ [f\bar{x}|\bar{L}]) :-$
$apply\_context\_vars(\Phi,\ \bar{x},\ \bar{L}).$

$head\_beta^*(\lambda\Gamma_A.(\lambda[\,].M)[\,],\ \lambda\Gamma_A.M).$

$head\_beta^*(\lambda\Gamma_A.(\lambda[y{:}B|\Gamma_B].M)[N|\bar{N}],\ M') :-$
$head\_beta^*(\lambda\Gamma_A.(\lambda[N/y]\Gamma_B.[N/y]M)\bar{N},\ M').$

$decompose(\lambda\Gamma_M.a\bar{M},\ \lambda\Gamma_N.a\bar{N},\ U) :-$
$a \in V(\Gamma_c),$
$decompose^\dagger(\Gamma_M,\ \bar{M},\ \Gamma_N,\ \bar{N},\ U).$

$decompose(\lambda\Gamma_M.a\bar{M},\ \lambda\Gamma_N.b\bar{N},\ U) :-$
$\Gamma_M[i] = a{:}A,\ \Gamma_N[i] = b{:}B,$
$decompose^\dagger(\Gamma_M,\ \bar{M},\ \Gamma_N,\ \bar{N},\ U).$

$decompose^\dagger(\Gamma_M,\ [\,],\ \Gamma_N,\ [\,],\ [\,]).$

$decompose^\dagger(\Gamma_M,\ [M|\bar{M}],\ \Gamma_N,\ [N|\bar{N}],$
$[\langle\lambda\Gamma_M.M, \lambda\Gamma_N.N\rangle|U]) :-$
$decompose^\dagger(\Gamma_M,\ \bar{M},\ \Gamma_N,\ \bar{N},\ U).$

...

The relation $\rightsquigarrow$, denoted by predicate $simulate(G, C, \sigma, G', C', \sigma')$, is defined as follows:

$simulate(P,\ G,\ C,\ \sigma,\ G',\ C',\ \sigma') :-$
$program\_context(P,\ \Delta),$
$goal(\Delta,\ \Gamma,\ U,\ G),$
$constraint(\Delta,\ \Gamma,\ U,\ C),$
$subst(\Delta,\ \Gamma,\ \theta,\ \sigma),$
$transform(U,\ \theta,\ \Gamma,\ U',\ \theta',\ \Gamma'),$
$goal(\Delta,\ \Gamma',\ U',\ G'),$
$constraint(\Delta,\ \Gamma',\ U',\ C'),$
$subst(\Delta,\ \Gamma',\ \theta',\ \sigma').$

$program\_context([\,],\ [\,]).$

$program\_context([\forall\bar{y}(B_0{:}\!-\bar{B})|P],$
$[x{:}\Pi(\Gamma_{vars}@\Gamma_{body}).B_0|\Delta]) :-$
$vars\_context(\bar{y},\ \Gamma_{vars}),$
$body\_context(\bar{B},\ \Gamma_{body}),$
$get\_new\_var(x),$
$program\_context(P,\ \Delta).$

$vars\_context([\,],\ [\,]).$

$vars\_context([y|\bar{y}],\ [y{:}c|\Gamma_{vars}]) :-$
$vars\_context(\bar{y},\ \Gamma_{vars}),$

$body\_context([\,],\ [\,]).$

$body\_context([B|\bar{B}],\ [z{:}B|\Gamma_{body}]) :-$
$get\_new\_var(z),\ body\_context(\bar{B},\ \Gamma_{body}).$

$goal(\Delta,\ \Gamma,\ [\,],\ [\,]).$

$goal(\Delta,\ \Gamma,\ [\langle\lambda\Delta.f\bar{x}, \lambda\Delta.f\bar{x}\rangle|U],\ [A|G]) :-$
$context\_vars(\Delta,\ \bar{x}),\ proof(\lambda\Delta.f\bar{x},\ \Gamma),$
$\Gamma(f) = \Pi\Delta.A,\ goal(\Delta,\ \Gamma,\ U,\ G).$

$goal(\Delta,\ \Gamma,\ [\langle M, N\rangle|U],\ G) :-$
$individual(M,\ \Gamma),\ goal(\Delta,\ \Gamma,\ U,\ G).$

$constraint(\Delta,\ \Gamma,\ [\,],\ [\,]).$

$constraint(\Delta,\ \Gamma,\ [\langle\lambda\Delta.M, \lambda\Delta.N\rangle|U],$
$[\langle M, N\rangle|C]) :-$
$individual(\lambda\Delta.M,\ \Gamma),$
$constraint(\Delta,\ \Gamma,\ U,\ C).$

$constraint(\Delta,\ \Gamma,\ [\langle M, N\rangle|U],\ C]) :-$
$proof(M,\ \Gamma),\ constraint(\Delta,\ \Gamma,\ U,\ C).$

...

goal, *constraint* and *subst* are the predicates corresponding to the functions of the same names defined above. Clauses in P are represented by expressions of the form $\forall\bar{y}(B_0{:}\!-\bar{B})$, where $\bar{y}$ denotes the sequence of variables that appear in clause $B_0{:}\!-\bar{B}$. For simplicity, we assume that all the variables in clauses and queries have the same type c. We further assume various conditions on the triple $T = \langle U, \Gamma, \theta\rangle$ and hence on $t = \langle G, C, \sigma\rangle = gcs(T)$. Those conditions are satisfied if $T_0 \Rightarrow^\dagger \cdots \Rightarrow^\dagger T$.

Due to the limitation of space, we cannot completely describe the transformation process. Following are some important steps.

We first define the new predicate $make\_new\_free\_vars^\dagger$, which is a generalization of $make\_new\_free\_vars$:

$make\_new\_free\_vars^\dagger(\Gamma_A,\ \bar{x},$
$[\,],\ \Gamma,\ B_0,\ [\,],\ \Gamma,\ B_0).$
$make\_new\_free\_vars^\dagger(\Gamma_A,\ \bar{x},$
$[y{:}B|\Gamma_B],\ \Gamma,\ B_0,$
$[f{:}\Pi\Gamma_A.B|\Phi],\ \Gamma',\ B_0') :-$
$get\_new\_free\_var(f),$
$make\_new\_free\_vars^\dagger(\Gamma_A,\ \bar{x},$
$[f\bar{x}/y]\Gamma_B,\ [f\bar{x}/y]\Gamma,\ [f\bar{x}/y]B_0,$
$\Phi,\ \Gamma',\ B_0').$

We then unfold *transform* in the body of *simulate* using the first clause of *transform*, which calls *project*. We also unfold *project*, *program_context*, etc. Referring to the unfolded body of *simulate* and the definition of $make\_new\_free\_vars^\dagger$, we define the following predicate *make_variant*:

$make\_variant(\Delta,\ \bar{x},\ x_p,\ f,$
$\bar{y},\ \bar{B},\ B_0,\ \bar{B}',\ B_0') :-$
$vars\_context(\bar{y},\ \Gamma_{vars}),$
$body\_context(\bar{B},\ \Gamma_{body}),$
$make\_new\_free\_vars^\dagger(\Delta,\ \bar{x},\ \Gamma_{vars},\ \Gamma_{body},\ B_0,$
$\Phi_{vars},\ \Gamma'_{body},\ B_0'),$
$make\_new\_free\_vars(\Delta,\ \bar{x},\ \Gamma'_{body},\ B_0',$
$\Phi_{body},\ B_0'),$
$apply\_context\_vars(\Phi_{vars},\ \bar{x},\ \bar{L}_{vars}),$
$apply\_context\_vars(\Phi_{body},\ \bar{x},\ \bar{L}_{body}),$

$L = \lambda\Delta.x_p(\bar{L}_{vars}@\bar{L}_{body})$.
$head\_beta^*([L/f](\lambda\Delta.f\bar{x}),\ M')$,
$decompose(M',\ M',\ U'')$.
$goal(\Delta,\ \Phi_{body},\ U'',\ \bar{B}')$.

Since $\bar{y}$ is a list, we can transform the above clause into the following ones using folding and eliminating unused parameters:

$make\_variant(\bar{x},\ [\,],\ \bar{B},\ B_0,\ \bar{B},\ B_0)$.
$make\_variant(\bar{x},\ [y|\bar{y}],\ \bar{B},\ B_0,\ \bar{B}',\ B_0') :-$
  $get\_new\_free\_var(f)$,
  $make\_variant(\bar{x},\ \bar{y},\ [f\bar{x}/y]\bar{B},\ [f\bar{x}/y]B_0,$
    $\bar{B}',\ B_0')$.

We can then derive the following clause for *simulate*:

$simulate(P,\ G,\ C,\ \sigma,\ G',\ C',\ \sigma) :-$
  $G = G_1@[A_0]@G_2$,
  $P[p] = \forall\bar{y}(B_0{:-}\bar{B})$,
  $program\_vars(P,\ \bar{x})$,
  $make\_variant(\bar{x},\ \bar{y},\ \bar{B},\ B_0,\ \bar{B}',\ B_0')$,
  $G' = G_1@G_2@\bar{B}'$,
  $C' = C@[\langle A_0, B_0'\rangle]$.
$program\_vars([\,],\ [\,])$.
$program\_vars([\_|P],\ [x|\bar{x}]) :-$
  $get\_new\_var(x)$, $program\_vars(P,\ \bar{x})$.

The derivation requires using various properties of predicates and functions appearing in the programs. It is also necessary to formalize substitutions and operations on them, including *subst*. Since the programs contain predicates with side-effects, i.e., *get_new_free_var* and *get_new_var*, the derivation also requires inferences on those predicates and regard clauses as equivalent if their results always coincide under renaming of new variables.

Using the other clause of *transform*, which calls *imitate*, we can obtain the other clause for *simulate*. In order to simulate first-order unification, however, we must carry out one more transformation step: We should replace each term of the form $f\bar{x}$ in constraints with a simple free variable f', and transform substitution σ into σ' such that $\sigma'(f') = M'$ if and only if $\sigma(f) = \lambda\Delta.M$, where M' is the result of replacing $f\bar{x}$ with f' in M. The details are omitted in this paper.

6 Conclusions

This paper showed that higher-order unification is powerful enough to simulate SLD-resolution of Prolog. We need no device to realize a prover for a higher-order type system other than a higher-order unification procedure for it.

In approaches such as [5], on the other hand, a resolution theorem prover is used for constructing a closed term for a given higher-order type. Such a procedure works almost in the same way as a higher-order unification procedure works as showed in this paper.

In order to implement higher-order unification under resolution, it is necessary to cope with both of the nondeterminism of unification and that of resolution [8]. If we use higher-order unification as a theorem proving procedure, the unification procedure becomes the only source of nondeterminism.

References

[1] Barendregt, H.: Introduction to generalized type systems, *Theoretical Computer Science — Proceedings of the Third Italian Conference,* World Scientific (1989), pp.1-37.

[2] Coquand, T., Huet, G.: The calculus of constructions, *Information and Computation,* Vol.76, No.3/4 (1988), pp.95-120.

[3] Elliott, C. M.: Higher-order unification with dependent function types, *Rewriting Techniques and Applications* (Dershowitz, N. ed.), LNCS355 (1989), pp.121-136.

[4] Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics, *Symposium on Logic in Computer Science,* 1987, pp.194-204.

[5] Helmink, L.: Resolution and type theory, *ESOP'90* (Jones, N. ed.), LNCS432 (1990), pp.197-211.

[6] Howard, W. A.: The formulae-as-types notion of construction, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism,* (Hindley, J. R., Seldin, J. P. eds.), Academic Press (1980), pp.479-490.

[7] Huet, G. P.: A unification algorithm for typed λ-calculus, *Theoretical Computer Science,* Vol.1 (1975), pp.27-57.

[8] Nadathur, G., Miller, D.: An overview of λPROLOG, *Proceedings of the Fifth International Conference and Symposium on Logic Programming,* 1988, pp.810-827.

[9] Paulson, L. C.: Natural deduction as higher-order resolution, *Journal of Logic Programming,* Vol.3 (1986), pp.237-258.

[10] Pfenning, F.: Elf: A language for logic definition and verified metaprogramming, *Symposium on Logic in Computer Science,* 1989, pp.313-322.

[11] Snyder, W., Gallier, J.: Higher-order unification revisited: Complete sets of transformations, *Journal of Symbolic Computation,* Vol.8, Nos 1&2 (1989), pp.101-140.

[12] Tamaki, H., Sato, T.: Unfold/fold transformation of logic programs, *International Conference on Logic Programming,* Uppsala, 1984, pp.127-138.