Practical Higher-Order Unification
with On-the-Fly Raising

Gopalan Nadathur
Computer Science Department
University of Minnesota

[Based on work with Natalie Linnell]
Motivating Higher-Order Pattern Unification

The following queries illustrate different levels of unification:

?- append (a::b::nil) (a :: nil) L.
   L = a :: b :: a :: nil.

?- append (a :: b :: nil) (a :: nil) (F a).
requires solving the unification problem
   (F a) = a :: b :: a :: nil
[multiple solutions, branching in unification]

?- ∀a append (a::b::nil) (a::nil) (F a).
requires solving
   ∃F∀a (F a) = a::b::a::nil.
[most general unifier, non-branching search]

The last is an instance of higher-order pattern unification.
Features of Higher-Order Pattern Unification

- Arises naturally in computations over higher-order abstract syntax
- Mixed quantifier prefixes are an essential component of the problem and usually evolve dynamically
- Has properties similar to first-order unification
  - most general unifiers can be provided
  - unification is decidable and near linear-time algorithm exists

*Question*: How close can we get to first-order like treatment in an implementation?
Talk Outline

• Formal presentation of the problem
• Naive, transformation rules based algorithm
• Eliminating quantifier prefixes
• Sketch of a more sophisticated algorithm based on
  – recursive traversal of terms
  – on-the-fly application of pruning and raising
• Comparison with other approaches
• Concluding comments
The Structure of Unification Problems

Universal, existential and abstracted variables are distinguished.

In particular, terms are given by

$$t ::= x | u | i | \lambda(i, t) | t(t)$$

where $i$ is a positive number and $t$ is a sequence of terms.

Unification problems are lists of equations under a quantifier prefix.

Examples of such problems are

- $\forall u \exists x (x = u :: \text{nil})$
- $\exists x \forall u (x = u :: \text{nil})$
- $\forall u \exists x_1 \exists x_2 (x_1 = x_2 :: \text{nil})$
- $\forall u_1 \forall u_2 \exists x (x(u_2) = u_1(u_2) :: \text{nil})$

Note: All existential, universal and lambda bound variables must be explicitly bound in the prefix or by an abstraction.
Solutions to Unification Problems

- A term $t$ is *proper* for existential variable $x$ if every “free” variable in it is bound outside the scope of $x$’s quantifier.

- A unifier for a unification problem is a substitution for existential variables such that
  - each pair in it is proper, and
  - it renders the terms in each equation equal modulo the $\beta$- and $\eta$-rules

Prefix may be extended with existential quantifiers over new variables in the process.

- A unifier is *most general* if any other unifier can be obtained from it by composition with a proper substitution.
Examples

- \( \forall u \exists x (x = u :: \text{nil}) \) has \( \{\langle x, u \rangle\} \) as a unifier.

- \( \exists x \forall u (x = u :: \text{nil}) \) has no unifiers.

- \( \forall u \exists x_1 \exists x_2 (x_1 = x_2 :: \text{nil}) \) has as a unifier \( \{\langle x_1, x_3 \rangle, \langle x_2, x_3 \rangle\} \) after modification to \( \forall u \exists x_3 \exists x_1 \exists x_2 (x_1 = x_2 :: \text{nil}) \).

- \( \forall u_1 \forall u_2 \exists x (x(u_2) = u_1(u_2) :: \text{nil}) \) has as unifiers
  \( \{\langle x, \lambda(1, u_1(1))\rangle\} \) and \( \{\langle x, \lambda(1, u_1(u_2))\rangle\} \).

  This problem has no most general unifier.
Higher-Order Pattern Unification Problems

These are problems in which the terms in the equations satisfy the following property:

Every existential variable occurrence has as arguments distinct
- lambda bound variables or
- universal variables bound within the scope of the quantifier for the existential variable.

For example, $\forall u_1 \exists x \forall u_2 (x(u_2) = u_1(u_2) :: \text{nil})$ is such a problem.

Restriction leads to most general unifiers and decidable unification.

E.g. the problem shown as $\{\langle x, \lambda(1, u_1(1)) \rangle\}$ as an mgu.
Solving Unification Problems

- Algorithm based on transformation rules of the form
  \[ \langle Q_1(E_1), \theta_1 \rangle \rightarrow \langle Q_2(E_2), \theta_2 \rangle \]
  such that if \( \langle Q(E), \emptyset \rangle \xrightarrow{*} \langle Q'(nil), \theta \rangle \) then \( \theta \) is an mgu for \( Q(E) \)

- Rules assume symmetry of = and normal forms for terms

- Higher-order pattern restriction is assumed to be satisfied

- Transformation system is complete for higher-order pattern unification:
  - successful reduction yields mgu
  - getting “stuck” indicates non-unifiability

- Equation list yields a processing order corresponding to recursion over term structure
Notation Used in Rules

• Associated with a sequence of terms $\bar{t}$:
  
  \begin{align*}
  |\bar{t}| & \quad \text{length of } \bar{t} \\
  \bar{t}[i] & \quad \text{i}th \text{ element of } \bar{t} \\
  \bar{t} + \bar{s} & \quad \text{concatenation of } \bar{t} \text{ and } \bar{s}
  \end{align*}

• Associated with sequences of distinct lambda bound and universal variables $\bar{y}$ and $\bar{z}$:
  
  \begin{itemize}
  \item if $a = \bar{z}[i]$ then $a\downarrow\bar{z} = |\bar{z}| + 1 - i$
  \item $\bar{y}\downarrow\bar{z} = \bar{y}[1]\downarrow\bar{z}, \ldots, \bar{y}[|\bar{y}|]\downarrow\bar{z}$, provided all elements of $\bar{y}$ appear in $\bar{z}$.
  \item $\bar{y}\cap\bar{z}$ is some listing of the list of elements common to $\bar{y}$ and $\bar{z}$.
  \end{itemize}
Simplification Transformations

- **Removing Abstractions**
  \[ \langle Q(\lambda(n, s) = \lambda(n, t) :: E), \theta \rangle \longrightarrow \langle Q(s = t :: E), \theta \rangle \]

- **Descending Under Rigid Heads**
  \[ \langle Q(a(s_1, \ldots, s_n) = a(t_1, \ldots, t_n) :: E), \theta \rangle \longrightarrow \langle Q(s_1 = t_1 :: \ldots :: s_n = t_n :: E), \theta \rangle \]
  if \( a \) is a lambda bound or universal variable.

  Note that failure occurs implicitly if heads are different.
Flexible-Rigid Transformation

\[ \langle Q_1 \exists f \, Q_2(f(\overline{y}) = a(t_1, \ldots, t_n) :: E), \theta \rangle \rightarrow \langle Q_1 \exists h_1 \ldots \exists h_n \exists f \, Q_2(h_1(\overline{y}) = t_1 :: \ldots :: h_n(\overline{y}) = t_n :: \theta'(E)), \theta' \circ \theta \rangle \]

where \( \theta' = \{ \langle f, \lambda(|\overline{y}|, a'(h_1(|\overline{y}|, \ldots, 1), \ldots, h_n(|\overline{y}|, \ldots, 1))) \rangle \} \)

provided

- \( f \) does not appear in \( a(t_1, \ldots, t_n) \), and

- \( a \) is a lambda bound or universal variable such that
  - \( a \) appears in \( \overline{y} \) and \( a' = a_{\downarrow} \overline{y} \), or
  - \( a \) is quantified in \( Q_1 \) and \( a' = a \).
Flexible-Flexible Transformation (Same Var)

\[ \langle Q_1 \exists f Q_2(f(y_1, \ldots, y_n)) = f(z_1, \ldots, z_n) :: E), \theta \rangle \]
\[ \quad \rightarrow \langle Q_1 \exists h \exists f Q_2(\theta'(E)), \theta' \circ \theta \rangle \]

where

- \( \theta' = \{ \langle f, \lambda(n, h(\overline{w})) \rangle \} \) and
- \( \overline{w} \) is some listing of the set \( \{ m + 1 - i | y_i = z_i \text{ for } i \leq n \} \)
Flexible-Flexible Transformation (Different Vars)

- **No Intervening Universal Quantifiers**
  \[
  \langle Q_1 \exists f \exists g Q_3 (f(y) = g(z) :: E), \theta \rangle \rightarrow \\
  \langle Q_1 \exists h \exists f Q_2 \exists g Q_3 (\theta'(E)), \theta' \circ \theta \rangle
  \]
  for \( \theta = \{ \langle f, \lambda(|y|, h(u)) \rangle, \langle g, \lambda(|z|, h(v)) \rangle \} \)
  where \( \overline{u} = \overline{w} \downarrow y \) and \( \overline{v} = \overline{w} \downarrow z \) for \( w = y \cap z \)

- **Raising Transformation**
  \[
  \langle Q_1 \exists f Q_2 \exists g Q_3 (f(y) = g(z) :: E), \theta \rangle \rightarrow \\
  \langle Q_1 \exists f \exists h Q_2 \exists g Q_3 (f(y) = h(\overline{w} + z) :: \theta'(E), \theta' \circ \theta) \rangle
  \]
  where \( \overline{w} \) is a listing of the variables quantified universally in \( Q_2 \), and \( \theta' = \{ \langle g, h(\overline{w}) \rangle \} \).
Inefficiencies in the Naive Algorithm

- Raising Transformation
  - Maintaining and examining the quantifier prefix
  - Creating large lists of arguments
  - Introducing unnecessary arguments that have to be pruned later

- Incremental substitution generation in flexible-rigid case
  - unnecessary term construction
  - repeated occurs check

- Legitimacy check for rigid head in flex-rigid case
  - requires prefix examination
  - depends also on size of argument list for flexible term
Relevance of the Quantifier Prefix

Quantifier prefix is used for the following:

- Distinguishing existential and universal variables
  
  Store type tags with variables

- Checking adherence to higher-order pattern condition
  
  Record quantifier position
  
  In particular, maintain $l_x$, the number of changes from existential to universal quantification before the quantifier for $x$

- Effecting the raising transformation
  
  Relativize raising to the arguments of the other flexible term instead
Raising without the Quantifier Prefix

Consider the equation

\[ f(\overline{y}) = g(\overline{z}) \]

where \( f \) and \( g \) are existential variables such that \( l_f \leq l_g \).

To solve this equation, we have to transform both sides to the form

\[ h(\overline{w}) \]

where

\( h \) is a new existential variable, and

\( \overline{w} \) consists of two parts:

- variables \( u \) in \( \overline{y} \) such that \( l_u \leq l_g \)
- variables shared between \( \overline{y} \) and \( \overline{z} \).

Substitutions for \( f \) and \( g \) must be coordinated to generate this term.
Modified Flex-Flex (Different Vars) Rule

Let $\bar{y} \uparrow g$ denote a listing of the set

$$\{u \mid u \text{ is a universal variable in } \bar{y} \text{ such that } l_u \leq l_g\}$$

Then rules for the flexible-flexible with different heads case can be replaced by

$$\langle f(\bar{y}) = g(\bar{z}) :: E, \theta \rangle \rightarrow \langle \theta'(E), \theta' \circ \theta \rangle$$

for $\theta' = \{\langle f, \lambda(|\bar{y}|, h(\bar{q} + \bar{v})) \rangle, \langle g, \lambda(\bar{z}, h(\bar{p} + \bar{u})) \rangle\}$

where

- $h$ is a new existential variable such that $l_u \leq l_f$,
- $\bar{p} = \bar{y} \uparrow g$ and $\bar{q} = \bar{p} \downarrow \bar{y}$, and
- $\bar{v} = (\bar{y} \cap \bar{z}) \downarrow \bar{y}$ and $\bar{u} = (\bar{y} \cap \bar{z}) \downarrow \bar{z}$

assuming that $l_f \leq l_g$. 
The Full Algorithm

• Based on a recursive traversal of terms in two modes:
  – First-order like term simplification
  – Variable binding, initiated by flex-flex or flex-rigid pair

• Variable binding computation is parameterized by
  – variable to be bound,
  – vector of its arguments, and
  – term constituting the other half of the equation

• Subpart of variable binding is a “make substitution” phase that returns
  – a substitution term, and
  – possible substitutions for embedded variables
Example

Consider the unification problem

\[
\exists x \forall a \forall b \forall c \exists y \forall d (b(x(a,d)) = b(a(y)) :: \text{nil})
\]

After labelling of variables and dropping of the prefix this becomes

\[
(b_{c(1)}(x_{v(0)}(a_{c(1)}, d_{c(2)})) = b_{c(1)}(a_{c(1)}(y_{v(1)})) :: \text{nil})
\]
Comparison with Other Algorithms

Two existing styles of algorithms:

• Based on an explicit \textit{a priori} raising
  
  e.g. [Nipkow], [Qian]
  
  – must maintain list of all universals encountered
  
  – blind raising coupled with pruning of redundant variables

• explicit substitution based approach, characterized by graftable metavariables
  
  e.g. [Dowek, Hardin, Kirchner, Pfenning], [Pfenning, Pientka]
  
  – can avoid initial raising, but
  
  – dynamic behaviour can be akin to blind raising
Conclusions and Future Work

• Algorithm has been implemented in C and SML and used in actual systems

• Relevance of explicit substitutions needs to be better understood:
  – seems useful for delaying reduction substitution, but
  – do graftable metavariables really offer a benefit?

• Compilation issues and impact on $\lambda$Prolog processing model to be examined.