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

Gopalan Nadathur

Digital Technology Center and Department of Computer Science
University of Minnesota

LIX – January 10, 2006

[Joint work with Natalie Linnell]

Motivating Higher-Order Pattern Unification

Some "Prolog" queries illustrating different forms 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

$\forall b \forall a \exists F\, (\mathtt{F\ a}) = \mathtt{a :: b :: a :: nil}$

[multiple solutions, branching in unification]

?- $\forall a$ `append (a :: b :: nil) (a :: nil) (F a).`

requires solving

$\forall b \exists F \forall a\, (\mathtt{F\ a}) = \mathtt{a :: b :: a :: nil.}$

[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?

Outline of the Talk

- 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

Unification problems are lists of equations between lambda terms embedded within a quantifier prefix.

Term syntax uses de Bruijn notation and combines sequences of applications and abstractions:

$$t ::= x \mid u \mid i \mid \lambda(i, t) \mid t(\bar{t})$$

where i is a positive number and $\bar{t}$ is a sequence of terms.

Every variable appearing in the equations must be bound by an abstraction or a quantifier in the prefix.

Examples of unification problems:

$$\forall f \forall c \exists x \; (x = f(c) :: nil)$$
$$\forall f \exists x \forall c \; (x = f(c) :: nil)$$
$$\forall u \forall v \exists x \; (x(v) = u(v) :: nil)$$

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 β- and η-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

- $\forall f \forall c \exists x\, (x = f(c) :: nil)$ has $\{\langle x, f(c)\rangle\}$ as a unifier.
- $\forall f \exists x \forall c\, (x = f(c) :: nil)$ has no unifiers.
- $\forall u \forall v \exists x\, (x(v) = u(v) :: nil)$ has as unifiers

  $\{\langle x, \lambda(1, u(1))\rangle\}$ and $\{\langle x, \lambda(1, u(v))\rangle\}$.

  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, $\forall u \forall v \exists x\, (x(v) = u(v) :: nil)$ is not such a problem.

However, $\forall u \exists x \forall v\, (x(v) = u(v) :: nil)$ does satisfy the restriction.

Also, every first-order problem meets the requirement trivially.

Unification via Transformations of Equations

- Algorithm based on rewrite rules of the form

  $$\langle \mathcal{Q}_1(E_1), \theta_1 \rangle \longrightarrow \langle \mathcal{Q}_2(E_2), \theta_2 \rangle$$

  such that if $\langle \mathcal{Q}(E), \emptyset \rangle \stackrel{*}{\longrightarrow} \langle \mathcal{Q}'(nil), \theta \rangle$ then θ is an mgu for $\mathcal{Q}(E)$
- Rules assume symmetry of $=$ and normal forms for terms
- Higher-order pattern restriction is assumed to be satisfied
- Transformation system is complete in the sense that
  - successful reduction yields a most general unifier
  - getting "stuck" indicates non-unifiability in the pattern case
- Equation list modified so as to yield a processing order corresponding to recursion over term structure

Notation Used in Rules

- Associated with a sequence of terms $\bar{t}$:

$\lvert\bar{t}\rvert$	length of $\bar{t}$
$\bar{t}[i]$	ith element of $\bar{t}$
$\bar{t} + \bar{s}$	concatenation of $\bar{t}$ and $\bar{s}$

- Associated with sequences of distinct lambda bound and universal variables $\bar{y}$ and $\bar{z}$:
  - if $a = \bar{z}[i]$ then $a{\downarrow}\bar{z} = \lvert\bar{z}\rvert + 1 - i$
  - $\bar{y}{\downarrow}\bar{z} = \bar{y}[1]{\downarrow}\bar{z}, \ldots, \bar{y}[\lvert\bar{y}\rvert]{\downarrow}\bar{z}$, provided all elements of $\bar{y}$ appear in $\bar{z}$.
  - $\bar{y} \cap \bar{z}$ is some listing of the set of elements common to $\bar{y}$ and $\bar{z}$.

Simplification Transformations

These rules eliminate common rigid structure at the top level in terms:

- *Removing Abstractions*

  $$\langle \mathcal{Q}(\lambda(n, s) = \lambda(n, t) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}(s = t :: E), \theta\rangle$$

- *Descending Under Rigid Heads*

  $$\langle \mathcal{Q}(a(s_1, \ldots, s_n) = a(t_1, \ldots, t_n) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}(s_1 = t_1 :: \ldots :: s_n = t_n :: E), \theta\rangle$$

  if a is a lambda bound or universal variable.

  Note: Failure occurs implicitly if heads are different.

Flexible-Rigid Transformation

An incremental substitution is posited to reduce the difference between the two terms:

$$\langle \mathcal{Q}_1 \exists f\, \mathcal{Q}_2 (f(\overline{y}) = a(t_1, \ldots, t_n) :: E), \theta \rangle \longrightarrow$$
$$\langle \mathcal{Q}_1 \exists h_1 \ldots \exists h_n \exists f\, \mathcal{Q}_2 (h_1(\overline{y}) = t_1 :: \ldots :: h_n(\overline{y}) = t_n :: \theta'(E)), \theta' \circ \theta \rangle$$

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 is quantified in $\mathcal{Q}_1$ and $a' = a$, or
  - a appears in $\overline{y}$ and $a' = a{\downarrow}\overline{y}$.

Note: Once again, failure is implicit if the conditions are not satisfied.

Flexible-Flexible Transformation (Same Variable)

Here, a substitution must be posited that prunes away arguments that are not identical in the same places:

$$\langle \mathcal{Q}_1 \exists f\, \mathcal{Q}_2 (f(y_1, \ldots, y_n)) = f(z_1, \ldots, z_n)) :: E), \theta\rangle$$
$$\longrightarrow \langle \mathcal{Q}_1 \exists h \exists f\, \mathcal{Q}_2 (\theta'(E)), \theta' \circ \theta\rangle$$

where

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

Flexible-Flexible Transformation (Different Variables)

- *No Intervening Universal Quantifiers*
  Preserve only those universal variables that are in both argument lists:

$$\langle \mathcal{Q}_1 \exists f\, \mathcal{Q}_2 \exists g\, \mathcal{Q}_3 (f(\overline{y}) = g(\overline{z}) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}_1 \exists h \exists f\, \mathcal{Q}_2 \exists g\, \mathcal{Q}_3 (\theta'(E)), \theta' \circ \theta\rangle$$

  for $\theta = \{\langle f, \lambda(|\overline{y}|, h(\overline{u}))\rangle, \langle g, \lambda(|\overline{z}|, h(\overline{v}))\rangle\}$
  where $\overline{u} = \overline{w}{\downarrow}\overline{y}$ and $\overline{v} = \overline{w}{\downarrow}\overline{z}$ for $w = \overline{y} \cap \overline{z}$

- *Raising Transformation*
  Bring quantifiers together through a substitution that encodes permitted dependencies:

$$\langle \mathcal{Q}_1 \exists f\, \mathcal{Q}_2 \exists g\, \mathcal{Q}_3 (f(\overline{y}) = g(\overline{z}) :: E), \theta\rangle \longrightarrow \langle \mathcal{Q}_1 \exists f \exists h\, \mathcal{Q}_2 \exists g\, \mathcal{Q}_3 (f(\overline{y}) = h(\overline{w} + \overline{z}) :: \theta'(E), \theta' \circ \theta\rangle$$

  where $\overline{w}$ is a listing of the variables quantified universally in $\mathcal{Q}_2$, and $\theta' = \{\langle g, h(\overline{w})\rangle\}$.

Inefficiencies in the Naive Algorithm

- Raising Transformation
  - Maintaining and examining the quantifier prefix
  - Introducing arguments that have to be pruned later
- Legitimacy check for rigid head in flex-rigid case
  - requires prefix examination
  - depends also on size of argument list for flexible term
- Incremental substitution generation in flexible-rigid case
  - unnecessary term construction
  - repeated occurs check

Eliminating the Quantifier Prefix

Quantifier prefix is used for the following:

- Distinguishing existential and universal variables
- Checking quantification order in flexible-rigid transformation
- Effecting the raising transformation

Eliminating the Quantifier Prefix

Quantifier prefix is used for the following:

- Distinguishing existential and universal variables

  Store type tags with variables

- Checking quantification order in flexible-rigid transformation

- Effecting the raising transformation

Eliminating the Quantifier Prefix

Quantifier prefix is used for the following:

- Distinguishing existential and universal variables

  Store type tags with variables

- Checking quantification order in flexible-rigid transformation

  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

Eliminating the Quantifier Prefix

Quantifier prefix is used for the following:

- Distinguishing existential and universal variables

  Store type tags with variables

- Checking quantification order in flexible-rigid transformation

  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 such that $l_h = l_f$, 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 to realize this can be generated "on-the-fly," solely from looking at $\overline{y}$ and $\overline{z}$.

Modified Flex-Flex (Different Variables) Rule

Let $\overline{y}\Uparrow g$ denote a listing of the set

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

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

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

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

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

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
- Variable binding involves recursive descent through term towards generating
  - a substitution term, and
  - possible substitutions for embedded variables
- Normalization is performed on-demand using explicit substitutions

Example

Consider the unification problem

$$\exists x \forall a \forall b \forall c \exists y \forall d (b(x(a,d)) = b(a(y)) :: nil)$$

After labelling of variables and dropping of the prefix this becomes

$$(b_{c(1)}(x_{v(0)}(a_{c(1)}, d_{c(2)})) = b_{c(1)}(a_{c(1)}(y_{v(1)})) :: nil)$$

After simplification applied to the (first) equation, we get

$$(x_{v(0)}(a_{c(1)}, d_{c(2)}) = a_{c(1)}(y_{v(1)}) :: nil)$$

Variable binding must now be applied to the equation to generate a unifier.

Example (Continued)

Variable binding unravels as follows:

$$\Big\downarrow$$

$$x_{v(0)}(a_{c(1)}, d_{c(2)}) = a_{c(1)}(y_{v(1)})$$

Example (Continued)

Variable binding unravels as follows:

$$\downarrow \quad \uparrow \quad \{\langle x, \lambda(2, \qquad\qquad))\rangle\} +$$

$$x_{v(0)}(a_{c(1)}, d_{c(2)}) = a_{c(1)}(y_{v(1)})$$

$$\downarrow$$

$$mksubst(x_{v(0)}, [a_{c(1)}, d_{c(2)}], a_{c(1)}(y_{v(1)}))$$

Example (Continued)

Variable binding unravels as follows:

$$\downarrow\uparrow \quad \{\langle x, \lambda(2, 2(\qquad\qquad)))\rangle\} +$$

$$x_{v(0)}(a_{c(1)}, d_{c(2)}) = a_{c(1)}(y_{v(1)})$$

$$\downarrow\uparrow \quad 2(\qquad\qquad)$$

$$mksubst(x_{v(0)}, [a_{c(1)}, d_{c(2)}], a_{c(1)}(y_{v(1)}))$$

$$\downarrow$$

$$mksubst(x_{v(0)}, [a_{c(1)}, d_{c(2)}], y_{v(1)})$$

Example (Continued)

Variable binding unravels as follows:

$$\downarrow\uparrow \quad \begin{array}{l} \{\langle x, \lambda(2, 2(h_{v(0)}(2)))\rangle\} + \\ \{\langle y, h_{v(0)}(a_{c(1)})\rangle\} \end{array}$$

$$x_{v(0)}(a_{c(1)}, d_{c(2)}) = a_{c(1)}(y_{v(1)})$$

$$\downarrow\uparrow \quad \begin{array}{l} 2(h_{v(0)}(2)) \\ \{\langle y, h_{v(0)}(a_{c(1)})\rangle\} \end{array}$$

$$mksubst(x_{v(0)}, [a_{c(1)}, d_{c(2)}], a_{c(1)}(y_{v(1)}))$$

$$\downarrow\uparrow \quad \begin{array}{l} h_{v(0)}(2) \\ \{\langle y, h_{v(0)}(a_{c(1)})\rangle\} \end{array}$$

$$mksubst(x_{v(0)}, [a_{c(1)}, d_{c(2)}], y_{v(1)})$$

Comparison with Other Algorithms

Two existing styles of algorithms:

- Based on an explicit *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]
  - 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
- Has a significant impact on performance in the *Teyjus* system
- Compilation of aspects beyond first-order like simplification are being examined
- Relevance of explicit substitutions needs to be better understood:
  - seems useful for delaying reduction substitution, but
  - do graftable metavariables really offer a benefit?