Programming in Higher-Order Logic

Lecture 4

Alwen Tiu

The Australian National University

ANU Logic Summer School

Outline

- Higher-order unification.
- Examples of higher-order programs: map, function lists, lazy lists, etc.
- Encoding structures with binders.

Higher-order unification

- The *higher-order unification problem* is the problem of finding a substitution θ, given λ terms s and t, such that $s\theta =_{\beta\eta} t\theta$. It is undecidable.
- M.g.u doesn't always exist even when the unification problem is solvable.
- Example: Let f and M be a function symbol and a variable of type $\alpha \to \alpha$. Consider the following unification problem:

$$\lambda x.M\ (f\ x) =^{?} \lambda x.f\ (M\ x).$$

[$f^k x$ means k applications of f to x.]
Then any substitution of the form

$$\theta_k = [M \mapsto \lambda x.f^k x], \qquad k \geq 0,$$

is a unifier, and it is not an instance of any other unifier.

Complete set of unifiers

Let $\mathcal{S}$ be a unification problem. Then a complete set of unifiers (CSU) for $\mathcal{S}$ is a set of substitutions $\mathcal{U}$ such that every $\sigma \in \mathcal{U}$ is a unifier for $\mathcal{S}$ and such that for every unifier θ of $\mathcal{S}$, there exists a unifier $\theta' \in \mathcal{U}$ and a substitution ρ such that

$$\theta = \theta' \circ \rho.$$

For example, for the unification problem in the previous slide:

$$\lambda x.M\ (f\ x) =^? \lambda x.f\ (M\ x).$$

the set

$$\mathcal{U} = \{\theta_k \mid k \geq 0\},$$

where $\theta_k = [M \mapsto \lambda x.f^k x]$, is a CSU for the above unification problem.

Some terminology

- Let t be the λ-normal term:

$$\lambda x_1 \cdots \lambda x_m.u\ t_1 \cdots t_n$$

  where u is a variable or a constant.
  The term u is called the head of t; t is a rigid term if u is a constant or in $\{x_1, \ldots, x_m\}$, otherwise it is a flexible term.
- A unification problem is $s =^? t$ is a rigid-rigid problem if both s and t are rigid terms; it is a flex-rigid if one of s and t is a flexible term and the other is is a rigid term; it is a flex-flex problem if both terms are flexible.

Huet's unification algorithm

It is a semi-decision procedure for generating complete set of unifiers. It consists of iterations of two simplication steps:

- The SIMPL procedure: it simplifies the rigid-rigid problems.
  Given a unification problem $\mathcal{S}$, it either returns a new unification problem containing only flex-rigid/flex-flex pairs, or terminates with failure.
- The MATCH procedure: it computes a set of substitutions that may form "initial segments" of unifiers.
  It consists of an *imitation* step and a *projection step* for simplifying flex-rigid problems.
  This step generates a tree of possible simplifications.

Huet's algorithm does not deal with flex-flex case, so it is really a *pre-unification* algorithm (but one could get lucky and solves the problem before reaching flex-flex cases).

The SIMPL procedure

Let $\mathcal{S}$ be the input unification problem. SIMPL($\mathcal{S}$) =

- If $\mathcal{S}$ is empty then return *Success*.
- If there is $(s =^? t) \in \mathcal{S}$ such that

$$s = \lambda x_1 \cdots \lambda x_n.u\ s_1 \cdots s_k \qquad t = \lambda x_1 \cdots \lambda x_n.v\ t_1 \cdots t_l$$

  and $u \neq v$ then return *Fail*.
- Otherwise, let $\mathcal{S}'$ be the union of $\mathcal{S}$ and

$$\{\lambda x_1 \cdots \lambda x_n.s_1 =^? \lambda x_1 \cdots \lambda x_n.t_1, \cdots, \lambda x_1 \cdots \lambda x_n.s_1 =^? \lambda x_1 \cdots \lambda x_n.t_1\}$$

  Return SIMPL($\mathcal{S}'$).
- If there are no rigid-rigid pairs left, reorient any remaining flex-rigid pairs: Return

$$\{t =^? s \mid (s =^? t) \in \mathcal{S} \text{ rigid-flex}\}$$
$$\cup \{s =^? t \mid (s =^? t) \in \mathcal{S} \text{ flex-rigid or flex-flex}\}.$$

The MATCH procedure: Imitation

Let F be a variable of type $\sigma_1 \to \cdots \to \sigma_k \to \tau$, where τ is a base type. Suppose we are given a flex-rigid pair $s =^? t$, where

$$s = (\lambda x_1 \cdots \lambda x_m.F\ s_1 \cdots s_k) \text{ and } t = (\lambda x_1 \cdots \lambda x_m.c\ t_1 \cdots t_n),$$

and a set of variables V (containing the free variables of s and t):
If $c \in \{x_1, \ldots, x_m\}$ then $IMIT(s, t, V) = \emptyset$. Otherwise,

$$IMIT(s, t, V) = \{[F \mapsto \lambda y_1 \cdots \lambda y_k.(c\ (H_1\ y_1 \cdots y_k) \cdots (H_n\ y_1 \cdots y_k))]\}$$

where $H_1, \ldots, H_n$ are new variables distinct from V, $x_1, \ldots, x_m$ and $y_1, \ldots, y_k$, of the appropriate types.

The MATCH procedure: Projection

Let F be a variable of type $\sigma_1 \to \cdots \to \sigma_k \to \tau$, where τ is a base type. Suppose we are given a flex-rigid pair $s =^? t$, where

$$s = (\lambda x_1 \cdots \lambda x_m.F\ s_1 \cdots s_k) \text{ and } t = (\lambda x_1 \cdots \lambda x_m.c\ t_1 \cdots t_n),$$

and a set of variables V (containing the free variables of s and t):
If σ_i, where $1 \leq i \leq k$, is of the form $\tau_1 \to \cdots \to \tau_n \to \tau$, then

$$PROJ_i(s, t, V) = \{[F \mapsto \lambda y_1 \cdots \lambda y_k.(y_i\ (H_1\ y_1 \cdots y_k)\ \cdots\ (H_n\ y_1 \cdots y_k))]\}$$

were $H_1, \ldots, H_n$ are new variables.
Otherwise, $PROJ_i(s, t, V) = \emptyset$. Let

$$PROJ(s, t, V) = \bigcup\{PROJ_i(s, t, V) \mid 1 \leq i \leq k\}.$$

Then define:

$$MATCH(s, t, V) = IMIT(s, t, V) \cup PROJ(s, t, V)$$

Constructing Match trees

The main unification algorithm is just an exhaustive search, represented as a search tree below: Nodes are either a *Success* node, a *Failure* node, or a unification problem $\mathcal{S}$.
Successors of a node $\mathcal{S}$:

- A *Success* or *Failure* node has no successor.
- If $\mathcal{S}$ has a rigid-rigid or rigid-flex pair, then $SIMPL(\mathcal{S})$ is the successor of $\mathcal{S}$.
- Otherwise, if $\mathcal{S}$ has a flex-rigid pair, say $s =^? t$, then $\mathcal{S}\theta$ is a successor of $\mathcal{S}$, for any $\theta \in MATCH(s, t, FV(s, t))$.

Example: higher-order unification

Use Huet's algorithm to solve the following problems:

- $\mathcal{S}_1 = \{\lambda x \lambda y.g\ x\ (\lambda z.y) =^? \lambda x \lambda y.g\ y\ (\lambda z.y)\}$.
- $\mathcal{S}_2 = \{\lambda x.M =^? \lambda x.x\}$.
- $\mathcal{S}_3 = \{F\ a =^? a\}$.
- $\mathcal{S}_4 = \{\lambda x.M\ (f\ x) =^? \lambda x.f\ (M\ x)\}$.
- $\mathcal{S}_5 = \{\lambda f \lambda x.M\ (f\ x) =^? \lambda f \lambda x.f\ (M\ x)\}$.

A decidable fragment

- A λ normal term t is a higher-order pattern term if for every free variable M, every occurrence of M in t is applied to *distinct bound variables* in t.
- Some examples:

$$(\lambda x \lambda y.M\ x\ y) \qquad f\ (\lambda x.N\ x)(\lambda y.g\ y) \qquad \lambda x \lambda f.f\ (f\ x)$$

  and some non-examples:

$$(\lambda x.M\ x\ x) \qquad (\lambda x \lambda y.M\ (\lambda x.x)\ N)$$

- A unification problem $\mathcal{S}$ is a higher-order pattern unification problem if for every $s =^? t \in \mathcal{S}$, s and t are higher-order pattern terms.
- Higher-order pattern unification is decidable [Miller 91] and an m.g.u. exists if the unification problem is sovable.

Higher-order pattern unification

- Many theorem proving problems in higher-order logic involves only higher-order pattern unification.
- Higher-order pattern unification can be solved by simply applying the SIMPL and MATCH procedures repeatedly.
- In the MATCH procedure, only one of imitation and projection steps needs to be applied, e.g., in the flex-rigid problem

$$(\lambda x \lambda y.M\ x\ y) =^? (\lambda x \lambda y.f\ x\ (g\ y))$$

only the imitation step will lead to a solution.

- Similarly, in

$$(\lambda x \lambda f.M\ x\ f) =^? (\lambda x \lambda f.f\ (f\ x))$$

only the projection step needs to be applied (i.e., projecting M on its second argument).

Higher-order features of λProlog

- The most recent implementation, the Teyjus system, implements only the higher-order pattern unification, not the full unification.
- Non-higher-order-pattern unification are handled partially, with the remaining flex-rigid and flex-flex pairs returned, e.g.,

```
?- (x\ f (M (f x))) = (x\ f x).

The answer substitution:
M = M

The remaining disagreement pairs list:
<M (f #1), #1>
```

- "Flexible goals" are not handled, e.g., a goal like $\exists X.X$ will generate runtime error, although syntactically it is allowed.

Example: predicate quantification

- Given a list L, a list R and a binary predicate R, check that the i-th element of L is related by R to the i-th element of R, for every i:

```
type mappred  (A -> B -> o) -> (list A) -> (list B) -> o.

mappred P nil nil.
mappred P (X :: L) (Y :: K) :- P X Y, mappred P L K.
```

- Given a list L and a unary predicate P, check that P holds for every element of L:

```
type forevery (A -> o) -> list A -> o.

forevery P nil.
forevery P (X :: L) :- P X, forevery P L.
```

Example: function quantification

Given a list L and a one-argument function F, apply F to every element of L:

```
type  mapfun   (A -> B) -> list A -> list B -> o.

mapfun F nil nil.
mapfun F (X :: L) ((F X) :: K) :- mapfun F L K.
```

Note that since F is just a simply typed term, applying F to X is just β-reduction. More complicated function evaluation mechanisms have to be encoded explicitly.

```
?- mapfun (x\ x + 1) [1,2,3] L.

The answer substitution:
L = 1 + 1 :: 2 + 1 :: 3 + 1 :: nil

?- mapfun (x\ x + 1) [1,2,3] L, mappred (x\ y\ y is x) L R.

The answer substitution:
R = 2 :: 3 :: 4 :: nil
L = 1 + 1 :: 2 + 1 :: 3 + 1 :: nil
```

Example: function-lists

- This is similar to difference-list, but instead of attaching a logic variable to the "tail" of a list, we use λ-abstraction.
- The list $[a_1, \ldots, a_n]$ is represented as the function

$$\lambda x.(a_1 :: \cdot :: a_n :: x).$$

  Thus a function list is of type: `(list A -> list A)`.
- The empty list is represented as the identity function $\lambda x.x$.
- Appending function-lists via β-reduction:

```
type fappend (list A -> list A) ->
             (list A -> list A) -> (list A -> list A) -> o.

fappend L R (z\L (R z)).
```

Example: recursion over function lists

- Check whether a function list is well-formed:

```
isflist (x\x).
isflist (x\A :: (L x)) :- isflist L.
```

- Reversing a function list:

```
frev (z\z) (z\z).
frev (z\ A :: (L z)) (z\RL (A :: z)) :- frev L RL.
```

- Converting a list to a function list:

```
list2flist nil (z\z).
list2flist (A::L) (z\ A :: (FL z)) :- list2flist L FL.
```

Example: encoding streams

- In *lazy* functional languages, one can represent a (possibly infinite) list such that the elements of the list are constructed only when needed.
- For example, the sequence of natural numbers can be generated in a typical lazy functional language via the function:

  ```
  f(n) = (n :: f(n+1))
  ```

  That is, as the list generated by executing `f(0)`. The function call in the tail of the list (`f(n+1)`) is suspended until the tail is queried.
- In an eager language, calling `f(0)` will lead infinite loop.
- We shall see a λProlog version of lazy lists; encoded via suspended higher-order predicates.

Example: data structures for streams

- Cells: containing either a value (a *forced cell*) or a suspended computation (a *delayed cell*).

```
kind cell type -> type.
type fcell A -> (cell A).          % forced cells
type dcell (A -> o) -> (cell A). % delayed cells
```

  The delayed cell takes a higher-order unary predicate.

- Streams:

```
kind strm type -> type.
type empty (strm A).
type stream A -> (cell (strm A)) -> (strm A).
```

  The constructor `stream` constructs a stream from an element and a cell containing a stream.

Example: operations on streams

- Taking the head and the tail of a stream:

```
head (stream V S) V.
tail (stream V S) T :- getcell S T.

getcell (fcell V) V.
getcell (dcell P) V :- P V.
```

  `getcell` "executes" the suspended predicate in the delayed cell, and passes its output to `V`.

- The 'list-like' interface to lazy streams:

```
snil empty.
scons X P (stream X (dcell P)).
```

Example: some simple streams

- Natural numbers, even and odd numbers:

```
% A sequence with I increment.
inc X I T :- Y is X + I, scons X (S\ inc Y I S) T.

nat S :- inc 0 1 S.
even S :- inc 0 2 S.
odd S :- inc 1 2 S.
```

- Fibonacci sequence:

```
fb X Y S :-
   Z is X + Y, scons Z (T\ fb Y Z T) S.

fib X Y (stream X (fcell (stream Y (fcell T)))) :-
   fb X Y T.
```

Example: querying streams

- Merging two streams:

```
merge empty S S.
merge S empty S.
merge (stream A S) (stream B T) (stream A (fcell W)) :-
   tail (stream A S) S', tail (stream B T) T',
   scons B (Z\ merge S' T' Z) W.
```

- Querying a finite segment of a stream:

```
take 0 S nil.
take N S (V :: L) :-
  N > 0, M is N - 1, head S V, tail S T, take M T L.
```

Encoding structures with binders

Recall the language of untyped λ-terms:

$$s, t ::= x \mid \lambda x.t \mid (s\ t)$$

A direct encoding would represent variables explicitly, e.g., as a string. For example, consider the following constructors:

$$\mathit{var} : \mathit{string} \to \mathit{tm} \qquad \mathit{app} : \mathit{tm} \to \mathit{tm} \to \mathit{tm} \qquad \mathit{abs} : \mathit{string} \to \mathit{tm} \to \mathit{tm}.$$

Thus, the term $\lambda x \lambda f.(f\ x)$ would be encoded as:

$$\mathit{abs}\ \text{“}x\text{”}\ (\mathit{abs}\ \text{“}f\text{”}\ (\mathit{app}\ (\mathit{var}\ \text{“}f\text{”})\ (\mathit{var}\ \text{“}x\text{”}))).$$

But with this encoding, α-equivalence needs to be defined explicitly as well, since $\lambda x.x$ and $\lambda y.y$ have different encodings:

$$\mathit{abs}\ \text{“}x\text{”}\ (\mathit{var}\ \text{“}x\text{”}) \text{ and } \mathit{abs}\ \text{“}y\text{”}\ (\mathit{var}\ \text{“}y\text{”}).$$

λ-tree syntax

- A different style of encoding uses λ-*tree syntax*.
- Variables are not explicitly encoded via constructors. Abstraction in the object terms is encoded as meta-level abstraction: λ-calculus).

$$app : tm \to tm \to tm \qquad abs : (tm \to tm) \to tm.$$

- We get two things for free:
  - α-equivalence in the object syntax co-incides with α-equivalence in the meta-level. For example, the encodings of $\lambda x.x$ and $\lambda y.y$ are

    $$(abs\ (\lambda x.x)) \qquad \text{and} \qquad (abs\ (\lambda y.y))$$

    which are α-equivalent terms at the meta level.
  - Capture-avoiding substitution in the object level is captured by β-reduction at the meta level. For example, if $t = (\lambda x.s)$ then to substitute x with u, we simply use

    $$(\lambda x.s)\ u.$$

Example: encoding first-order formulae

Consider encoding formulae of classical first-order logic:

```
kind form type.  % Syntactic category for formulae.
kind tm type.    % Syntactic category for (untyped) terms.

type const string -> tm. % constants

type atm string -> (list tm) -> form. % atomic formulae
type and form -> form -> form.        % conjunction
type or  form -> form -> form.        % disjunction
type imp form -> form -> form.        % implication
type all (tm -> form) -> form.        % universal quantifier
type some (tm -> form) -> form.       % existential quantifier

% Some operators can be written with infix notation
infixr and 120.
infixr or  120.
infixr imp 120.
```

Example: prenex normal form

Every first-order classical formula can be transformed into an equivalent formula in prenex normal form (pnf), i.e., formulae of the form

$$Q_1 x_1 \cdots Q_n x_n . F$$

where each Q_i is either a $\forall$ or an $\exists$ and F is a quantifier free formula.
The transformation uses the following logical equivalence (where Q can be $\forall$ or $\exists$):

$$\begin{gathered}
(Qx.G) \vee F \equiv F \vee (Qx.G) \equiv Qx.(F \vee G) \\
(Qx.G) \wedge F \equiv F \wedge (Qx.G) \equiv Qx.(F \wedge G) \\
F \Rightarrow (\forall x.G) \equiv \forall x.(F \Rightarrow G) \\
(\forall y.F) \Rightarrow G \equiv \exists y.(F \Rightarrow G) \\
F \Rightarrow (\exists x.G) \equiv \exists x.(F \Rightarrow G) \\
(\exists y.F) \Rightarrow G \equiv \forall y.(F \Rightarrow G)
\end{gathered}$$

In all of the above: x is not free in F and y is not free in G.

Example: prenex normal form (2)

```
pnf (atm X L) (atm X L).
pnf (A and B) C :-
    pnf A F, pnf B G, merge (F and G) C.
pnf (A or B) C :-
    pnf A F, pnf B G, merge (F or G) C.
.....
```

```
merge ((all x\B x) or C) (all D) :-
      pi x\ merge ((B x) or C) (D x).
merge ((some x\B x) or C) (some D) :-
      pi x\ merge ((B x) or C) (D x).
.....
```

The side condition that "x is not free in C" is satisfied trivially: C is not in the scope of x and substitution is capture-avoiding.