Programming in Higher-Order Logic

Lecture 3

Alwen Tiu

The Australian National University

ANU Logic Summer School

Outline

- First-order unification.
- Overview of λProlog and examples of first-order programs.
- Overview of proof theory for higher-order logic.
- Higher-order Horn clauses.
- Uniform provability for higher-order Horn clauses.

How to deal with variable instantiations

- In proving existential goal or instantiating program clauses, replace the bound variables with *logic variables*, which will be instantiated lazily.

$$\frac{\Sigma; \Gamma \longrightarrow B[Y/x]}{\Sigma; \Gamma \longrightarrow \exists x.B}\ \exists_R \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma; \mathcal{P} \xrightarrow{D[Y/x]} A}{\Sigma; \mathcal{P} \xrightarrow{\forall_\tau x.D} A}\ \forall_L, \quad Y \text{ new}$$

- Logic variables are a technical device for implementation; they are not part of the logical system.
- Logic variables are instantiated when applying the identity rule, e.g., to prove

$$\Sigma; \Gamma, p\ a \longrightarrow p\ Y$$

  using the *id* rule, the logic variable Y is instantiated with the constant a.
- This reduces to solving the syntactic equation: $p\ a = p\ Y$ which is also called a *unification problem*.

Unification problems

- A *unification problem* S is a set of pairs of terms (possible containing logic variables)
$$S = \{s_1 =^? t_1, \cdots, s_n =^? t_n\}.$$
- A solution for S is a substitution θ (from logic variables to terms) s.t. $s_i\theta = t_i\theta$. Such a θ is called a *unifier* for S.
- The set of unifiers can be infinite, but there is a most general one if unifiers exist.
- θ is a *most general unifier* (mgu) for S if for every a unifier σ of S, there exists a substitution ρ such that $\sigma = \theta \circ \rho$.
- Note: eigenvariables are treated like constants in unification.

First-order unification

For now, we restrict the unification problems to the first-order ones:

- No λ-abstractions.
- All variables are of base types.
- All function symbols are fully applied. This means, if $f : \tau_1 \to \cdots \tau_n \to \tau$, where τ is a base type, then every occurrence of f in the problem must be applied to n terms, e.g., $f\ t_1 \cdots t_n$.

Martelli-Montanari algorithm for first-order unification

Input: $S = \{s_1 =^? t_1, \dots, s_n =^? t_n\}$.
Output: $\perp$ (not unifiable), or an mgu θ.
Initial: $\theta = \emptyset$.

1. If $S = \emptyset$ then return θ.
2. If $t =^? t \in S$ then let $S := S \setminus \{t =^? t\}$ and repeat Step 1.
3. If $X =^? t \in S$ or $t =^? X \in S$ then
   - If $X \in FV(t)$, return $\perp$.
   - Otherwise, let $\theta := \theta \circ [X \mapsto t]$ and let

     $$S := (S \setminus \{X =^? t, t =^? X\})[X \mapsto t]$$

     and repeat Step 1.
4. If $f\ u_1 \cdots u_k =^? g\ v_1 \cdots v_l \in S$ and $f \neq g$ then return $\perp$.
5. If $f\ u_1\ \cdots\ u_k =^? f\ v_1 \cdots v_k \in S$ then

   $$S := (S \setminus \{f\ u_1\ \cdots\ u_k =^? f\ v_1 \cdots v_k\}) \cup \{u_1 =^? v_1, \dots, u_k =^? v_k\}.$$

   and repeat Step 1.

Example: solving a unification problem

$$
\begin{array}{c}
\{f\ X\ (g\ Y\ b) =^? f\ b\ (g\ X\ Y)\}, \quad \theta = \emptyset \\
\Downarrow \\
\{X =^? b,\ g\ Y\ b =^? g\ X\ Y\}, \quad \theta = \emptyset \\
\Downarrow \\
\{g\ Y\ b =^? g\ b\ Y\}, \quad \theta = [X \mapsto b] \\
\Downarrow \\
\{Y =^? b,\ b =^? Y\}, \quad \theta = [X \mapsto b] \\
\Downarrow \\
\{\}, \quad \theta = [X \mapsto b, Y \mapsto b]
\end{array}
$$

λProlog: an overview

- λProlog is a logic programming language based on higher-order intuitionistic logic.
- It's richer than the Horn fragment, but we shall use it to do some examples of first-order programming.
- Concrete syntax:

$\wedge$	`,` (comma)	$\vee$	`;` (semicolon)
$\Rightarrow$	`=>`	$\Leftarrow$	`:-` (reverse implication)
$\forall$	`pi`	$\exists$	`sigma`
$\lambda x.t$	`x\t`		

- Example: $\exists x.p\ x \wedge q\ x$ is written `(sigma x\p x, q x)`.

Type declaration

- The type of logical formulae is denoted by `o`.
- We can declare new base types and compound types, using the keyword `kind`.

```
kind nt type.
kind list type -> type.
```

- The arrow `->` in type declaration is used to form *type constructors*, i.e., functions to form new types from existing types.
- `list` can be used to form new base types, e.g., `list nt` denotes a list of elements of type `nt`.
- This means we can encode polymorphic lists in λProlog.
- Another example of a type constructor is the built-in function-type constructor: $\rightarrow$.

Function and predicate declaration

- Constants, function symbols and predicate symbols are declared using `type`.
- For example, lists are formed using two (data) constructors:

```
nil : list A.
:: : A -> (list A) -> (list A).
```

  Here `A` denotes a *type variable*. It can be instantiated to any type.
- Some typing information, such as types of logic variables and quantifiers, can be left implicit; λProlog is equipped with a type inference algorithm.
- The datatype `list` is built-in, so there's no need to explicitly declare it.

Example: appending lists

Lists are built in data structures in λProlog, and can be written using a more familiar notation, e.g., `[1,2,3,4]`.
Appending a list to another list:

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

append nil L L.
append (X::L) M (X::N) :- append L M N.
```

Example: appending lists (2)

Some example queries:

```
?- append [1,2,3] [4,5] L.

The answer substitution:

L = 1 :: 2 :: 3 :: nil
```

“Input” and “output” can be exchanged:

```
?- append L [4,5] [1,2,3,4,5].

The answer substitution:

L = 1 :: 2 :: 3 :: nil
```

Example: enumerating all solutions

We can ask λProlog to generate all possible solutions:

```
?- append L M [1,2,3].

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

More solutions (y/n)? y
The answer substitution:
M = 2 :: 3 :: nil
L = 1 :: nil

More solutions (y/n)? y
The answer substitution:
M = 3 :: nil
L = 1 :: 2 :: nil
...
```

Example: ordering of clauses and termination

λProlog selects the matching clauses in the order of their appearance. Ordering of clauses can affect termination.
Consider the append clause, but with the order reversed:

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

append (X::L) M (X::N) :- append L M N.
append nil L L.
```

Then the following query will not terminate:

```
?- append L R R.
```

In the original program, this query would give an answer: `L = nil`.

Example: reversing lists

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

reverse nil nil.
reverse (X::L) R :- reverse L S, append S (X::nil) R.
```

Example queries:

```
?- reverse [1,2,3] L.

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

More solutions (y/n)? y

no (more) solutions
```

Example: difference-lists

- There is a more "efficient" way of appending lists, by utilising unification and using a data structure called *difference-lists*.
- A *different list* is a pair of lists. The first component of the pair is a list and the second component is a logic variable representing the tail of the first list.
- Types and constructors for difference-lists:

  ```
  kind dlist type -> type.

  type dl (list A) -> (list A) -> (dlist A).
  ```

- Example: the list `[1,2,3]` is represented as the difference-list

  `(dl [1,2,3 | X] X)`

  where X is a logic variable.

Example: appending difference-lists

```
type append-dl (dlist A) -> (dlist A) -> (dlist A) -> o.

append-dl (dl L R) (dl R V) (dl L V).
```

Example query:

```
?- append-dl (dl [1,2|X] X) (dl [3,4|Y] Y) L.

The answer substitution:
L = dl (1 :: 2 :: 3 :: 4 :: Y) Y
Y = Y
X = 3 :: 4 :: Y
```

Example: converting difference lists to lists

```
type dlist2list (dlist A) -> (list A) -> o.
type list2dlist (list A) -> (dlist A) -> o.

dlist2list (dl L nil) L.
list2dlist L (dl R T) :- append L T R.
```

Example queries:

```
?- dlist2list (dl [1,2,3|X] X) L.

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

?- list2dlist [1,2,3] L.

The answer substitution:
L = dl (1 :: 2 :: 3 :: _T1) _T1
```

Higher-order programming

- *Higher-order programming* generally refers to a programming technique where functions or procedures can be treated as data and passed around as arguments of another function/procedure.
- In the context of logic programming, this means that one can use use predicates or even formulae as arguments of another predicate.
- It allows programming techniques such as *continuation passing* (CPS), encoding of *lazy* data structures (e.g., streams), etc.
- We use an intuitionistic variant of Church's higher-order logic as the foundation.

Some examples of higher-order progamming

Higher-order logic programming can be used to specify the following problems, simply and declaratively:

- Given a predicate of one argument and a list, check that every (or some) element of that list satisfies the predicate.
- Given a predicate of two arguments and two lists, check that corresponding elements of these two lists are related by the given predicate.
- Given a predicate and two lists, check that all of the elements of the second list satisfy the given predicate and are members of the rst list.
- Given a predicate of two arguments, construct its transitive closure.
- etc..

Intuitionistic higher-order logic

- To simplify terminology, we shall often refer to terms and formulae uniformly as simply formulae.
- We have the same set of logical constants as in the first-order case, but without the restrictions on the types of quantifiers and predicates.
- The notion of *atomic formula* is slightly different: it is a λ normal form

$$\lambda x_1 \cdots \lambda x_m . u \ F_1 \cdots F_n$$

  where u is either a variable or a non-logical constant.
- Quantified formulae: in $\forall_\tau x.P\ x$ and $\exists_\tau x.P\ x$ τ can be any type, e.g., it can be of type o (i.e., a formula).
- The rules are exactly the same as first-order rules.

Cut elimination

Proving cut eliminaton is significantly more difficult than the first-order case, e.g., in transforming

$$\dfrac{\dfrac{\dfrac{\vdots}{\Sigma, y : \tau; \Gamma \longrightarrow B\,y}}{\Sigma; \Gamma \longrightarrow \forall_\tau x. B\,x}\ \forall_R \quad \dfrac{\Sigma \vdash t : \tau \quad \dfrac{\vdots}{\Sigma; \Gamma, B\,t \longrightarrow C}}{\Sigma; \Gamma, \forall_\tau x. B\,x \longrightarrow C}\ \forall_L}{\Sigma; \Gamma \longrightarrow C}\ cut$$

into

$$\dfrac{\dfrac{\vdots}{\Sigma; \Gamma \longrightarrow B\,t} \quad \dfrac{\vdots}{\Sigma; \Gamma, B\,t \longrightarrow C}}{\Sigma; \Gamma \longrightarrow C}\ cut$$

the size of the cut formula $B\,t$ can be bigger than $\forall x. B\,x$, since t itself can be any formula, e.g., it can be the formula $\forall x. B\,x$ itself.

Higher-order Horn clauses

- The class of positive formulae, $\mathcal{PF}$, is the smallest collection of formulae such that:
  - Each variable and each constant other than $\Rightarrow$ and $\forall$ is in $\mathcal{PF}$.
  - The formula $\lambda x.A$ and $(A\ B)$ are in $\mathcal{PF}$ if A and B are in $\mathcal{PF}$.
- The Positive Herbrand Universe, $\mathcal{H}^+$, is the collection of all λ-normal formulae in $\mathcal{PF}$.
- A higher-order goal formula is a formula of type o in $\mathcal{H}^+$.
- A positive atom is an atomic goal formula.
- A higher-order program clause is any formula of the form

$$\forall x_1 \cdots \forall x_n.(G \Rightarrow A)$$

where G is a goal formula and A is a rigid positive atom.

Problem with goal-directed search

Instantiation of higher-order variables can introduce non-Horn programs and goals in proof search:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q\ a \longrightarrow q\ a}\ id}{q\ a \longrightarrow \exists Z.q\ Z}\ \exists_R \quad \dfrac{\dfrac{}{q\ b \longrightarrow q\ b}\ id}{q\ b \longrightarrow \exists Z.q\ Z}\ \exists_R}{q\ a \vee q\ b \longrightarrow \exists Z.q\ Z}\ \vee_L}{\longrightarrow (q\ a \vee q\ b) \Rightarrow \exists Z.q\ Z}\ \Rightarrow_R \quad \dfrac{\dfrac{}{p\ a \longrightarrow p\ a}\ id}{p\ a \longrightarrow \exists Y.p\ Y}\ \exists_R}{\dfrac{((q\ a \vee q\ b) \Rightarrow \exists Z.q\ Z) \Rightarrow p\ a \longrightarrow \exists Y.p\ Y}{\forall X(X \Rightarrow p\ a) \longrightarrow \exists Y.p\ Y}\ \forall_L}\ \Rightarrow_L
$$

We need to transform the offending subproofs into a proof with only Horn goals and program clauses.

Transforming non-Horn formulae

- Let *pos* be a mapping defined as:
  - If F is a constant or a variable:

$$pos(F) = \begin{cases} \lambda x \lambda y.\top, & \text{if } F \text{ is } \Rightarrow, \\ \lambda x.\top, & \text{if } F \text{ is } \forall, \\ F, & \text{otherwise.} \end{cases}$$

  - $pos((F_1\ F_2)) = (pos(F_1)\ pos(F_2))$.
  - $pos(\lambda x.F) = \lambda x.pos(F)$.
- Define a mapping *pc* on goal formulae as follows: if F is a formula then $pc(F)$ is the λ normal form of $pos(F)$.
- If G is a Horn goal formula, then $pc(G) = G$.

Implicational formulae

- *Implicational formula*: A formula F is an implicational formula if it is of the form

$$\forall \vec{x}.(H \Rightarrow A)$$

  where A is a rigid atom.

- *Transforming program clauses*: Define two mappings pos_i and pc_i on implicational formulae as follows:
  - If F is $H \Rightarrow A$, then $pos_i(F) = pos(H) \Rightarrow pos(A)$.
  - If F is $\forall x.F'$ then $pos_i(F) = \forall x.pos_i(F')$.
  - If F is an implicational formula, then $pc_i(F)$ is the λ-normal form of $pos_i(F)$.
- If F is a Horn program clause, then $pc_i(F) = F$.

Transforming non-Horn sequents

- Define a mapping pc_s to sequents as follow: Given a sequent $\mathcal{S}$:

$$\Sigma; B_1, \ldots, B_n \longrightarrow C$$

  $pc_s(\mathcal{S})$ is the sequent:

$$\Sigma; pc_o(B_1), \ldots, pc_o(B_n) \longrightarrow pc(C)$$

  where $pc_o(B_i) = pc_i(B_i)$ if B_i is an implicational formula, otherwise $pc_o(B_i) = pc(B_i)$.
- Notice that if Γ and C are, respectively, Horn program clauses and a Horn goal, then $pc_s(\mathcal{S}) = \mathcal{S}$.

Lemma (Horn-transformation)

If $\Sigma; \Gamma \longrightarrow C$ is provable then $pc_s(\Sigma; \Gamma \longrightarrow C)$ is also provable.

Completeness of goal directed proof search

Theorem

If the sequent $\Sigma; \Gamma \longrightarrow C$ is provable, where Γ is a set of higher-order program clauses and C is a higher-order goal formula, then there is a uniform proof for $\Sigma; \Gamma \longrightarrow C$.

Proof.

By rule permutation and the "Horn-transformation" lemma. For example, consider the non-uniform proof considered previously. After Horn-transformation, we have:

$$\dfrac{\dfrac{}{\longrightarrow \top} \qquad \dfrac{\dfrac{}{p\ a \longrightarrow p\ a}}{p\ a \longrightarrow \exists Y.p\ Y}}{\dfrac{\top \Rightarrow p\ a \longrightarrow \exists Y.p\ Y}{\forall X(X \Rightarrow p\ a) \longrightarrow \exists Y.p\ Y}}$$

by replacing $(q\ a \vee q\ b) \Rightarrow \exists Z.q\ Z$ with $\top$. We can then use rule permutations to turn this into a uniform proof. □