Programming in Higher-Order Logic

Lecture 2

Alwen Tiu

The Australian National University

ANU Logic Summer School
Outline

- First-order intuitionistic logic.
- Uniform provability and abstract logic programming.
- First-order Horn clauses.
- Completeness of goal directed search for first-order Horn clauses.
Using λ-calculus as a term language

- We shall extend propositional logic with
  - predicates (or relations) over simply typed λ-terms;
  - quantification over λ-terms.
- Thus this extended logic can specify classes of syntactic structures that can be encoded via λ-terms.
- In fact, even formulae themselves (propositional, first-order or higher-order) can be represented as λ-terms.
Example: representing natural numbers

- Introduce a base type $nt$ to denote natural numbers.
- Two constructors:
  - $z : nt$, represents the number zero.
  - $s : nt \rightarrow nt$, represents the successor function.
- Examples: the term $(s (s z))$ represents the number 2, $(s (s (s z)))$ represents the number 3, and so on.
Example: representing lists

- Introduce a base type $lst$ to denote lists of natural numbers.
- Two constructors:
  - $nil : lst$, representing an empty list.
  - $cons : nt \to list \to list$, constructing a list from a natural number and another list.
- Examples: the list 1,2,3 is represented as the $\lambda$-term

\[(cons\ 1\ (cons\ 2\ (cons\ 3\ nil)))\].
Example: representing binary trees

- Introduce a base type $bt$ to denote binary trees.
- Two constructors:
  - $emp : bt$, representing an empty tree.
  - $node : nt \to bt \to bt \to bt$, representing an internal node, decorated with a natural number and has two child nodes.
- Example: the term

  $$ (\text{node } 1 (\text{node } 2 \text{ emp emp}) (\text{node } 3 \text{ emp emp})) $$

  represents a binary tree with three nodes.
Example: encoding untyped $\lambda$-terms

- Introduce a base type $tm$ to denote untyped $\lambda$ terms.
- Two constructors:
  - $app : tm \rightarrow tm \rightarrow tm$, representing applications.
  - $abs : (tm \rightarrow tm) \rightarrow tm$, representing abstraction (notice the type).
- Example: the (object) $\lambda$-term ($\lambda x \lambda y . x \; y$) is represented as the (meta) typed $\lambda$-term:

  $$abs \: (\lambda x . abs \: (\lambda y . app \; x \; y)).$$

- Notice that object-level $\lambda$-abstraction are encoded using meta-level $\lambda$-abstraction. Therefore, object-level $\alpha$-equivalence is captured faithfully at the meta-level.
Example: representing logical formulae

- Introduce a type $o$ to represent formulae and $\iota$ to represent individuals. (This is Church’s original notation)
- Propositional operators:
  \[
  \bot, \top : o \\
  \land, \lor, \rightarrow : o \rightarrow o \rightarrow o
  \]
  For example, $(A \land B)$ is represented as $(\land A B)$.
- Quantifiers:
  \[
  \forall, \exists : (\iota \rightarrow o) \rightarrow o.
  \]
  For example, $\forall x. p(x) \land q(x)$ is represented as
  \[
  \forall (\lambda x. \land (p \ x \ q \ x)).
  \]
A first-order intuitionistic logic

Formulae are simply typed $\lambda$-terms of type $o$, formed using the following constructors:

- **Logical constants**:
  - $\bot, \top : o$
  - $\land, \lor, \rightarrow : o \rightarrow o \rightarrow o$
  - $\forall_\tau, \exists_\tau : (\tau \rightarrow o) \rightarrow o$.

For now, we restrict the type $\tau$ in the quantifiers to one which does not contain $o$.

- **Predicate symbols**, e.g.,
  - $p : \tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n \rightarrow o$.

Again, for now, the type $\tau_i$ is restricted to not contain the type $o$.

- **Other non-logical constants**, depending on the applications, e.g., lists, trees, natural numbers, etc.
Sequent calculus with explicit signatures

- We add an explicit typing environment $\Sigma$ (also called a *signature*) to sequents:
  $$\Sigma; \Gamma \rightarrow C.$$  
- The formulae in $\Gamma$ and $C$ must all be of type $o$, under the typing environment $\Sigma$, and are all in *normal forms*.
- We shall use the infix notation when writing down binary connectives, e.g., $A \land B$ instead of $\land A \ B$.
- Lambda abstraction in quantifiers are left implicit, e.g., we write $\forall x.B \ x$ instead of $\forall (\lambda x.B \ x)$. 

Inference rules

- The propositional rules are essentially the same, e.g.,

\[
\frac{\Sigma; \Gamma, A \rightarrow C}{\Sigma; \Gamma, A \land B \rightarrow C} \land_L
\]

- The first-order rules:

\[
\frac{\Sigma \vdash t : \tau \quad \Sigma; \Gamma, B[x \mapsto t] \rightarrow C}{\Sigma; \Gamma, \forall x. B \rightarrow C} \forall_L \quad \frac{\Sigma, y : \tau; \Gamma \rightarrow B[x \mapsto y]}{\Sigma; \Gamma \rightarrow \forall x. B} \forall_R
\]

\[
\frac{\Sigma, y : \tau; \Gamma, B[x \mapsto y] \rightarrow C}{\Sigma; \Gamma, \exists x. B \rightarrow C} \exists_L \quad \frac{\Sigma \vdash t : \tau \quad \Sigma; \Gamma \rightarrow B[x \mapsto t]}{\Sigma; \Gamma \rightarrow \exists x. B} \exists_R
\]

- In \(\forall_R\) and \(\exists_L\), the variable \(y\) does not occur in \(\Sigma\). It is called an eigenvariable.

- Implicit in \(\forall_L\) and \(\exists_R\) is that the formulae are normalized after applying the substitutions.
Empty types

- A type $\tau$ is non-empty under a typing environment $\Sigma$ if there is a term $t$ such that $\Sigma \vdash t : \tau$.
- We do not assume that all types are non-empty. Consequently, the sequent

$$\Sigma; \Gamma \rightarrow \exists_\tau x. \top$$

may not be derivable, if $\tau$ is empty.
- A base type corresponds to the notion of a domain in classical first-order logic.
- In contrast to our intuitionistic logic, in classical logic, the domain is assumed to be non-empty. Hence $\exists x. \top$ is valid in classical logic.
A simple example

Consider the problem of the “mortality of Socrates:”

*We all know that all men are mortal. We also know that Socrates is a man. Therefore, by modus ponens, Socrates is mortal. Q.E.D.*

Let $ps$ be a base type denoting the set of persons. Let $\Sigma$ be the typing environment

$$\Sigma = \{\text{socrates} : ps, \text{man} : ps \rightarrow o, \text{mortal} : ps \rightarrow o\}$$

Then the following sequent is provable:

$$\Sigma; \forall_{ps \times \text{man} \times} \text{mortal} \times, \text{man socrates} \rightarrow \text{mortal socrates}$$
Substitution of eigenvariables

Eigenvariables capture the meaning of universality; they act as placeholders for arbitrary values.
If we have a derivation of

\[ \Sigma, x : \tau; \Gamma \rightarrow C \]

then for every term \( t \) such that \( \Sigma \vdash t : \tau \), we also have a derivation of

\[ \Sigma; \Gamma[x \mapsto t] \rightarrow C[x \mapsto t] \]

To prove this, we also need to show that \( \Sigma, x : \tau \vdash s : \tau_1 \) implies
\( \Sigma \vdash s[x \mapsto t] : \tau_1 \)
Cut elimination

**Theorem**

*If a sequent $\Sigma; \Gamma \rightarrow C$ is provable then it is provable without using the cut rule.*

\[
\begin{align*}
\Sigma, y : \tau; \Gamma & \rightarrow B y \\
\Sigma ; \Gamma & \rightarrow \forall x . B x \\
\Sigma ; \Gamma , \forall x . B x & \rightarrow C \\
\Sigma ; \Gamma & \rightarrow C \\
\end{align*}
\]

reduces to (using substitution of eigenvariables)

\[
\begin{align*}
\Sigma ; \Gamma & \rightarrow B t \\
\Sigma ; \Gamma , B t & \rightarrow C \\
\Sigma ; \Gamma & \rightarrow C \\
\end{align*}
\]
Specifying computation as proof search

- Logic programming is just a way of encoding computation as proof search.
- Sequents are used to encode states of computation. A computation state consists of:
  - a signature $\Sigma$ containing the non-logical constants that the computation will involve.
  - a logic program $P$, which is a multiset of $\Sigma$-formulas that specifies the meaning of the constants in $\Sigma$, and
  - a query or goal $G$, which is a $\Sigma$-formula.
- Computation is a the process of attempting to prove the sequent $\Sigma; P \rightarrow G$. If successful, the resulting proof could be returned, e.g., in forms of answer substitutions.
Problems with proof search

Given a sequent, there are potentially many directions to explore:

- We can use the cut rule.
- Structural rules: unrestricted contraction/weakening.
- Left/right introduction rules.
- Finding instantiations of variables in quantifier rules.

Some obvious reduction in search space:

- By cut-elimination, we need not consider the cut rule.
- Structural rules can be absorbed into initial and introduction rules.

More difficult problems:

- Variable instantiations: use unification algorithm.
- Left/right choices of intro rules: goal-directed proof search.
An idealized interpreter for goal-directed search

An idealized interpreter has three components: signature $\Sigma$, a set of $\Sigma$-formulas $\mathcal{P}$ (program) and a $\Sigma$-formula $G$ (goal). The state of this idealized interpreter is denoted by the sequent $\Sigma; \mathcal{P} \rightarrow G$.

Desirable operational behaviors of goal-directed search:

- **AND** Reduce $\Sigma; \mathcal{P} \rightarrow B_1 \land B_2$ to $\Sigma; \mathcal{P} \rightarrow B_1$ and $\Sigma; \mathcal{P} \rightarrow B_2$.
- **OR** Reduce $\Sigma; \mathcal{P} \rightarrow B_1 \lor B_2$ to either $\Sigma; \mathcal{P} \rightarrow B_1$ or $\Sigma; \mathcal{P} \rightarrow B_2$.
- **INST** Reduce $\Sigma; \mathcal{P} \rightarrow \exists \tau x. B$ to $\Sigma; \mathcal{P} \rightarrow B[t/x]$, for some $\Sigma$-term $t$.
- **TRUE** The $\Sigma; \mathcal{P} \rightarrow \top$ is provable immediately.

Note: these reductions do not consider the logic program or the signature at all.

Behaviors of logical connectives cannot be modified by logic programs.
Completeness problems

Completeness of goal-directed search

Do we get all the theorems via goal-directed search?

No. Counterexample:

\[ \Sigma; p \lor q \rightarrow q \lor p \]

\[ \Sigma; r a \lor r b \rightarrow \exists x. r x \]

What restrictions should be made about program formulas to guarantee completeness of goal-directed proof search?
Uniform provability

Goal-directed search is formalized via the notion of uniform provability:

**Definition (Uniform provability)**

A cut-free intuitionistic proof is a *uniform proof* if every sequent in the proof with a non-atomic succedent is the conclusion of a right-introduction rule.

**Definition (Abstract logic programming)**

Let $\vdash$ be a provability relation in some logic. Let $\mathcal{D}$ be a set of formulas denoting program clauses and let $\mathcal{G}$ be a set of formulas denoting goal formulas in an intended logic programming. The triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ is an abstract logic programming language if and only if for every finite subset $\mathcal{P}$ of $\mathcal{D}$ and for every $G \in \mathcal{G}$,

$$\Sigma; \mathcal{P} \vdash G$$

if and only if $\Sigma; \mathcal{P} \longrightarrow G$ has a uniform proof.
First-order Horn clauses

Syntax of first-order Horn clauses:

\[
G ::= \top \mid A \mid G \land G \mid G \lor G \mid \exists \tau x. G \\
D ::= A \mid G \Rightarrow A \mid D \land D \mid \forall \tau x. D
\]

A denotes an atomic formula. \(D\)-formulas are called \textit{program clauses} and \(G\)-formulas are called \textit{goal formulas}.

**Theorem**

Let \(\vdash_{IL}\) denote intuitionistic provability. Then \(\langle D, G, \vdash_{IL} \rangle\) is an abstract programming language.

Actually, for this fragment, classical and intuitionistic provability coincides.
A common technique to prove completeness of goal-directed search is via *rules permutation*. For example:

\[
\begin{align*}
\Sigma; \Gamma \to A & \quad \Sigma; \Gamma, B \to C \\
\Sigma; \Gamma, A \to B & \to C \land D \quad \Rightarrow_L \\
\Sigma; \Gamma \to A & \quad \Sigma; \Gamma, B \to D \\
\Sigma; \Gamma, A \to B & \to C \land D \quad \Rightarrow_L \\
\-end{align*}
\]

is transformed into

\[
\begin{align*}
\Sigma; \Gamma \to A & \quad \Sigma; \Gamma, B \to C \\
\Sigma; \Gamma, A \to B & \to C \land D \quad \Rightarrow_L \\
\Sigma; \Gamma \to A & \quad \Sigma; \Gamma, B \to D \\
\Sigma; \Gamma, A \to B & \to D \quad \Rightarrow_L \\
\end{align*}
\]
Dealing with atomic goals: Backchaining

\[ \Sigma; \mathcal{P} \xrightarrow{D} A \]
\[ \Sigma; \mathcal{P} \xrightarrow{} A \quad \text{decide} \]
\[ \Sigma; \mathcal{P} \xrightarrow{D_1} A \]
\[ \Sigma; \mathcal{P} \xrightarrow{D_1 \land D_2} A \quad \land_L \]
\[ \Sigma; \mathcal{P} \xrightarrow{D_2} A \]
\[ \Sigma; \mathcal{P} \xrightarrow{D_1 \land D_2} A \quad \land_L \]
\[ \Sigma; \mathcal{P} \xrightarrow{G} \]
\[ \Sigma; \mathcal{P} \xrightarrow{D} A \quad \Rightarrow_L \]
\[ \Sigma; \mathcal{P} \xrightarrow{G \Rightarrow D} A \]
\[ \Sigma; \mathcal{P} \xrightarrow{D[t/x]} A \quad \forall_L \]
\[ \Sigma; \mathcal{P} \xrightarrow{\forall_{\tau \times D}} A \]

- $A$ denotes an atomic formula.
- The “focussed” sequent $\Sigma; \mathcal{P} \xrightarrow{D} A$ is the same as the sequent $\Sigma; \mathcal{P}, D \xrightarrow{} A$ except that we can only apply the rules to $D$. 
Completeness for Horn clauses with backchaining

**Theorem**

Let $\Sigma; \Gamma \longrightarrow C$ be a sequent where $\Gamma$ is a set of Horn program clauses and $C$ is a Horn goal formula. If $\Sigma; \Gamma \longrightarrow C$ is derivable then it is derivable using the right-introduction rules and the backchaining rules.