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.
  - $s : nt \to nt$, 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 λ-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

$$(node\ 1\ (node\ 2\ emp\ emp)\ (node\ 3\ emp\ emp))$$

represents a binary tree with three nodes.

Example: encoding untyped λ-terms

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

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

- Notice that object-level λ-abstraction are encoded using meta-level λ-abstraction. Therefore, object-level α-equivalence is captured faithfully at the meta-level.

Example: representing logical formulae

- Introduce a type o to represent formulae and ι to represent individuals. (This is Church's original notation)
- Propositional operators:

$$\begin{aligned} \bot, \top &: o \\ \wedge, \vee, \Rightarrow &: o \to o \to o \end{aligned}$$

  For example, $(A \wedge B)$ is represented as $(\wedge\ A\ B)$.
- Quantifiers:

$$\forall, \exists : (\iota \to o) \to o.$$

  For example, $\forall x.p(x) \wedge q(x)$ is represented as

$$\forall\ (\lambda x. \wedge\ \ (p\ x)\ (q\ x)).$$

A first-order intuitionistic logic

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

- *Logical constants*:

  $$\bot, \top : o \qquad \wedge, \vee, \Rightarrow : o \to o \to o \qquad \forall_\tau, \exists_\tau : (\tau \to o) \to o.$$

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

- *Predicate symbols*, e.g.,

  $$p : \tau_1 \to \tau_2 \to \cdots \to \tau_n \to o.$$

  Again, for now, the type τ_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 Σ (also called a *signature*) to sequents:
$$\Sigma; \Gamma \longrightarrow C.$$
- The formulae in Γ and C must all be of type o, under the typing environment Σ, and are all in *normal forms*.
- We shall use the infix notation when writing down binary connectives, e.g., $A \wedge B$ instead of $\wedge\ A\ B$.
- 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.,

$$\frac{\Sigma; \Gamma, A \longrightarrow C}{\Sigma; \Gamma, A \wedge B \longrightarrow C} \wedge_L$$

- The first-order rules:

$$\frac{\Sigma \vdash t : \tau \quad \Sigma; \Gamma, B[x \mapsto t] \longrightarrow C}{\Sigma; \Gamma, \forall_\tau x.B \longrightarrow C} \forall_L \qquad \frac{\Sigma, y : \tau; \Gamma \longrightarrow B[x \mapsto y]}{\Sigma; \Gamma \longrightarrow \forall_\tau x.B} \forall_R$$

$$\frac{\Sigma, y : \tau; \Gamma, B[x \mapsto y] \longrightarrow C}{\Sigma; \Gamma, \exists_\tau x.B \longrightarrow C} \exists_L \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma; \Gamma \longrightarrow B[x \mapsto t]}{\Sigma; \Gamma \longrightarrow \exists_\tau x.B} \exists_R$$

- In $\forall_R$ and $\exists_L$, the variable y does not occur in Σ. 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 τ is non-empty under a typing environment Σ 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

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

  may not be derivable, if τ 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 Σ be the typing environment

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

Then the following sequent is provable:

$$\Sigma; \forall_{ps} x.man\ x \Rightarrow mortal\ x,\ man\ socrates \longrightarrow mortal\ socrates$$

Substitution of eigenvariables

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

$$\Sigma, x : \tau; \Gamma \longrightarrow C$$

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

$$\Sigma; \Gamma[x \mapsto t] \longrightarrow C[x \mapsto t]$$

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 $\Sigma; \Gamma \longrightarrow C$ is provable then it is provable without using the cut rule.

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

reduces to (using substitution of eigenvariables)

$$
\dfrac{\begin{array}{c}\vdots \\ \Sigma; \Gamma \longrightarrow B\,t\end{array} \qquad \begin{array}{c}\vdots \\ \Sigma; \Gamma, B\,t \longrightarrow C\end{array}}{\Sigma; \Gamma \longrightarrow C}\ cut
$$

Specifiying 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 Σ containing the *non-logical constants* that the computation will involve.
  - a *logic program* P, which is a multiset of Σ-formulas that specifies the meaning of the constants in Σ, and
  - a *query* or *goal* G, which is a Σ-formula.
- Computation is a the process of attempting to prove the sequent $\Sigma; P \longrightarrow G$. 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 Σ, a set of Σ-formulas $\mathcal{P}$ (program) and a Σ-formula G (goal). The state of this idealized interpreter is denoted by the sequent $\Sigma; \mathcal{P} \longrightarrow G$.

Desirable operational behaviors of goal-directed search:

- **AND** Reduce $\Sigma; \mathcal{P} \longrightarrow B_1 \wedge B_2$ to $\Sigma; \mathcal{P} \longrightarrow B_1$ and $\Sigma; \mathcal{P} \longrightarrow B_2$.
- **OR** Reduce $\Sigma; \mathcal{P} \longrightarrow B_1 \vee B_2$ to either $\Sigma; \mathcal{P} \longrightarrow B_1$ or $\Sigma; \mathcal{P} \longrightarrow B_2$.
- **INST** Reduce $\Sigma; \mathcal{P} \longrightarrow \exists_\tau x.B$ to $\Sigma; \mathcal{P} \longrightarrow B[t/x]$, for some Σ-term t.
- **TRUE** The $\Sigma; \mathcal{P} \longrightarrow \top$ 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:

$$\Sigma; p \vee q \longrightarrow q \vee p$$

$$\Sigma; r\ a \vee r\ b \longrightarrow \exists x. r\ x$$

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:

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_\tau x.G$$
$$D ::= A \mid G \Rightarrow A \mid D \wedge D \mid \forall_\tau x.D$$

A denotes an atomic formula. D-formulas are called *program clauses* and G-formulas are called *goal formulas*.

Theorem

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

Actually, for this fragment, classical and intuitionistic provability coincides.

Rule permutation

A common technique to prove completeness of goal-directed search is via *rules permutation.* For example:

$$\dfrac{\begin{array}{c}\vdots\\ \Sigma;\Gamma \longrightarrow A\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Sigma;\Gamma,B \longrightarrow C\end{array} \quad \begin{array}{c}\vdots\\ \Sigma;\Gamma,B \longrightarrow D\end{array}}{\Sigma;\Gamma,B \longrightarrow C \wedge D}\ \wedge_R}{\Sigma;\Gamma, A \Rightarrow B \longrightarrow C \wedge D}\ \Rightarrow_L$$

is transformed into

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Sigma;\Gamma \longrightarrow A\end{array} \quad \begin{array}{c}\vdots\\ \Sigma;\Gamma,B \longrightarrow C\end{array}}{\Sigma;\Gamma, A \Rightarrow B \longrightarrow C}\ \Rightarrow_L \qquad \dfrac{\begin{array}{c}\vdots\\ \Sigma;\Gamma \longrightarrow A\end{array} \quad \begin{array}{c}\vdots\\ \Sigma;\Gamma,B \longrightarrow D\end{array}}{\Sigma;\Gamma, A \Rightarrow B \longrightarrow D}\ \Rightarrow_L}{\Sigma;\Gamma, A \Rightarrow B \longrightarrow C \wedge D}\ \wedge_R$$

Dealing with atomic goals: Backchaining

$$\frac{\Sigma; \mathcal{P} \xrightarrow{D} A}{\Sigma; \mathcal{P} \longrightarrow A}\ decide \qquad \frac{}{\Sigma; \mathcal{P} \xrightarrow{A} A}\ id$$

$$\frac{\Sigma; \mathcal{P} \xrightarrow{D_1} A}{\Sigma; \mathcal{P} \xrightarrow{D_1 \wedge D_2} A}\ \wedge_L \qquad \frac{\Sigma; \mathcal{P} \xrightarrow{D_2} A}{\Sigma; \mathcal{P} \xrightarrow{D_1 \wedge D_2} A}\ \wedge_L$$

$$\frac{\Sigma; \mathcal{P} \longrightarrow G \quad \Sigma; \mathcal{P} \xrightarrow{D} A}{\Sigma; \mathcal{P} \xrightarrow{G \Rightarrow D} A}\ \Rightarrow_L \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma; \mathcal{P} \xrightarrow{D[t/x]} A}{\Sigma; \mathcal{P} \xrightarrow{\forall_\tau x.D} A}\ \forall_L$$

- A denotes an atomic formula.
- The "focussed" sequent $\Sigma; \mathcal{P} \xrightarrow{D} A$ is the same as the sequent

$$\Sigma; \mathcal{P}, D \longrightarrow A$$

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* Γ *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.*