Programming in Higher-Order Logic

Lecture 1

Alwen Tiu

The Australian National University

ANU Logic Summer School
Logic and programming

- The idea of using logic as a *declarative* programming language begun in the 60’s, motivated mainly by theorem proving and AI applications.
- The underlying view can perhaps be best summarised by the following equation (due to Bob Kowalski):

  \[
  \text{Algorithm} = \text{Logic} + \text{Control}
  \]

- Procedural interpretation of logic programs is basically (depth-first) search for proofs.
- Its best known manifestation is the programming language Prolog.
Programming = Logic + ...

- The idea of a purely declarative programming language didn’t quite work.

Programming = Logic + Control + I/O + Modules

+ Concurrency + Object-oriented + . . . .

- An important (and perhaps ambitious) goal is to bring it down to:

Programming = Logic

- This course is about an attempt at reducing the former to the latter.

- Specifically, it’s about a *purely logical* extension of Prolog with: higher-order programming, modules and abstraction.
Outline of the course

1. Intuitionistic logic, proof theory, sequent calculus, $\lambda$-calculus.
2. First-order intuitionistic logic, uniform provability, first-order Horn clauses, abstract logic programming.
3. First-order unification, higher-order Horn clauses, overview of $\lambda$Prolog.
4. Higher-order unification, examples of higher-order programming.
5. Hereditary Harrop formulae, hypothetical and generic judgments, module and abstract data types, encoding of a functional language.
Course website

Course materials (slides, notes, program codes) will be made available on the following website:
http://users.rsise.anu.edu.au/~tiu/teaching/lss/
An abstract view of logic programming

- Generally speaking, logic programming is nothing but a specific instance of theorem proving: Given a set of program clauses $\Gamma$, show that a query $F$ follows from $\Gamma$.
- The difference lies in a particular view of logical connectives as instructions for search.
- To prove $\Gamma \vdash A \land B$ in the logic programming way means to prove $\Gamma \vdash A$ and $\Gamma \vdash B$.
- In other words, logic programming can be seen as goal-directed proof search.
- An important requirement is that goal-directed proof search must be complete.
Modular programming and data abstraction

- Given the view of connectives as search instruction, how does one encode a notion of modules and abstract datatypes?
- Abstraction – quantifiers (∃ or ∀), e.g., ∀ can be used to quantify over a range of structures satisfying certain abstract properties.
- Modules – implication. \( A \Rightarrow B \) means “load \( A \) and use it to prove \( B \)”. 
Classical logic and scope extrusion

- In classical logic, we have:

\[ P \lor (Q \Rightarrow R) \equiv Q \Rightarrow (P \lor R) \quad P \lor (\forall x. Q) \equiv \forall x. (P \lor Q) \]

when \( x \) is not free in \( P \).

- Call these properties “scope extrusion”.

- Scope extrusion clearly violates the principle of modularity and abstraction.

- So to capture these notions logically (in a goal-directed manner), we turn to a different logic, *intuitionistic logic*, in which scope extrusion doesn’t hold.
Intuitionistic logic is a branch of logic that originated from the constructivist approach to the foundations of mathematics in the late 19th century. Its conception was generally attributed to Brouwer.

It is characterised by the rejection of the principle of proof by contradiction and the excluded middle principle, and its insistence on constructive proofs.

Proofs and proof constructions play a central role.

Its semantics logic can be given in terms of possible-worlds interpretation (a.k.a. Kripke semantics). One can encode intuitionistic logic into modal logic S4.
Classical vs. intuitionistic truth

Disjunction property

- **Classical:** $A \lor B$ ("$A$ or $B$") is true if $A$ is true or $B$ is true.
- **Intuitionistic:** $A \lor B$ is true if you can show me a proof of $A$ or a proof of $B$.

Existential property

- **Classical:** $\exists x. A(x)$ ("there exists $x$ such that $A(x)$ holds") is true if assuming the non-existence of such an object leads to contradiction.
- **Intuitionistic:** $\exists x. A(x)$ is true if you can construct an object $t$ and a proof of $A(t)$.
A sequent calculus for intuitionistic logic

- An *intuitionistic sequent* is an expression of the form

  \[ \Gamma \rightarrow A \]

  where \( \Gamma \) is a *multi-set* of formulae, and \( A \) is a formula. Its intuitive reading is that “\( A \) follows from \( \Gamma \)”.

- We call \( \Gamma \) the *antecedent* (or *hypothesis*) and \( A \) the *succedent* of the sequent.

- We shall first look at the propositional case, i.e., Gentzen’s *LJ*. 

The identity and cut rules

\[ \Gamma, A \rightarrow A \quad \text{id} \quad \Gamma \rightarrow A \quad A, \Delta \rightarrow B \quad \text{cut} \]

- The cut rule expresses transitivity of logical reasoning (modus ponens).
- The central theorem of sequent calculus is the so-called cut elimination: the cut rule is redundant.
- Cut elimination has several important consequences, as we’ll see later.
Logical rules

- These are rules that compose proofs of smaller statements into proofs of a bigger compound statement.
- Logical rules are divided into two parts: the left-introduction rules and the right-introduction rules.
- Example:

\[
\frac{A, \Gamma \rightarrow C}{A \land B, \Gamma \rightarrow C} \quad \land_L \quad \frac{\Gamma \rightarrow A \quad \Gamma \rightarrow B}{\Gamma \rightarrow A \land B} \quad \land_R
\]
Complete list of logical rules

\[ \bot \rightarrow C \quad \bot^L \]
\[ \Gamma \rightarrow \top \quad \top^R \]

\[ \frac{A, \Gamma \rightarrow C}{A \land B, \Gamma \rightarrow C} \quad \land^L \]
\[ \frac{B, \Gamma \rightarrow C}{A \land B, \Gamma \rightarrow C} \quad \land^L \]
\[ \frac{\Gamma \rightarrow A}{\Gamma \rightarrow A \land B} \quad \land^R \]

\[ \frac{A, \Gamma \rightarrow C}{A \lor B, \Gamma \rightarrow C} \quad \lor^L \]
\[ \frac{\Gamma \rightarrow A}{\Gamma \rightarrow A \lor B} \quad \lor^R \]
\[ \frac{\Gamma \rightarrow B}{\Gamma \rightarrow A \lor B} \quad \lor^R \]

\[ \frac{\Gamma \rightarrow A}{A \Rightarrow B, \Gamma \rightarrow C} \quad \Rightarrow^L \]
\[ \frac{\Gamma, A \rightarrow B}{\Gamma \rightarrow A \Rightarrow B} \quad \Rightarrow^R \]
Structural rules

- These rules are used to manage the antecedent part of a sequent.

\[
\Gamma, A, A \rightarrow \Delta \\
\Gamma, A \rightarrow \Delta
\]

\[c_L\]

\[
\Gamma \rightarrow \Delta \\
\Gamma, A \rightarrow \Delta
\]

\[w_L\]

- The *contraction rule* allows removal of redundant hypotheses in a sequent. Reading the rule bottom up, it says that a logical proposition is an inexhaustible resource; we can use it as many times as we like.

- The weakening rule is not really needed. It can be absorbed into other rules.
Example: a derivation in sequent calculus

\[
\begin{array}{c}
A, B \rightarrow A & \text{id} & A, B \rightarrow B & \text{id} \\
\hline
A, B \rightarrow A \land B & \land_R \\
A, B \land C \rightarrow A \land B & \land_L \\
A \land C, B \land C \rightarrow A \land B & \land_L \\
A \land C, B \land C \rightarrow (A \land B) \lor C & \lor_R \\
\end{array}
\]

A shorter derivation:

\[
\begin{array}{c}
C, B \land C \rightarrow C & \text{id} \\
\hline
A \land C, B \land C \rightarrow C & \land_L \\
A \land C, B \land C \rightarrow (A \land B) \lor C & \lor_R \\
\end{array}
\]
Example: the need for the contraction rule

\[
\begin{align*}
& \frac{p \rightarrow p}{\text{id}} \\
& \frac{p \rightarrow p \land (p \Rightarrow q)}{\lor R} \\
& \frac{(p \lor (p \Rightarrow q)) \Rightarrow q, p \rightarrow q}{\rightarrow R} \\
& \frac{(p \lor (p \Rightarrow q)) \Rightarrow q \rightarrow p \Rightarrow q}{\lor R} \\
& \frac{(p \lor (p \Rightarrow q)) \Rightarrow q \rightarrow p \lor (p \Rightarrow q)}{\lor R} \\
& \frac{(p \lor (p \Rightarrow q)) \Rightarrow q, (p \lor (p \Rightarrow q)) \Rightarrow q \rightarrow q}{\text{id}} \\
& \frac{(p \lor (p \Rightarrow q)) \Rightarrow q \rightarrow q}{\text{cL}}
\end{align*}
\]
Cut elimination

The cut rule is very useful in composition of proofs (i.e., reading the rule top-down):

\[
\frac{\Gamma \rightarrow B \quad B, \Gamma \rightarrow A}{\Gamma \rightarrow A} \text{ cut}
\]

In searching for a proof of a sequent, we apply the inference rules bottom up. In this case, cut poses a big problem: which formula \( B \) should we use?

**Theorem (Cut elimination)**

*If a formula \( A \) is provable in LJ, then it is provable without using the cut rule.*
Example: a cut reduction

\[\vdash \ldots \Gamma \rightarrow A \Gamma \rightarrow B \rightarrow A \land B \rightarrow \Gamma \rightarrow A \land B, \Gamma \rightarrow C \rightarrow C \land R \rightarrow A, \Gamma \rightarrow C \rightarrow C \land L \rightarrow \ldots\]

This can be transformed to:

\[\vdash \ldots \Gamma \rightarrow A \rightarrow A \land B, \Gamma \rightarrow C \rightarrow C \rightarrow \Gamma \rightarrow C \rightarrow C \land L \rightarrow \ldots\]
Consequences of cut elimination

- **Syntactic consistency**: The formula $\bot$ is not derivable.
- **Subformula property**: If a sequent $\Gamma \rightarrow C$ is provable, then every sequent appearing in the proof uses only subformulae of $\Gamma$ and $C$.
- **Separation property**: Define a *fragment* of a logic by the set of connectives allowed in its formulae. Let $\mathcal{L}$ be a fragment of intuitionistic logic. Then the proof system obtained from $LJ$ by restricting its logical rules to those connectives in $\mathcal{L}$ is a complete proof system for $\mathcal{L}$.
First-order intuitionistic logic

- In encoding computation systems, we need to encode and manipulate computation objects, such as data structures, programs, states, etc.
- We need to extend propositional logic to allow logical statements that depend on individual objects.
- We shall first look at a *first-order* extension, where one can quantify over individual objects (programs, datas, etc), but not propositions.
- But first, we need a language to represent first-order structures.
- We shall use Church’s *simply typed* λ-calculus.
Developed by Alonzo Church in the 30’s as a language for representing computable functions.

Functions are represented using a “nameless” notation, e.g.,

\[ f(x) = x + 1 \text{ is represented as } \lambda x.(x + 1). \]

The notation \( \lambda x \) is called a \( \lambda \)-abstraction.

Function application, such as \( f(5) \), is represented via application

\[ (\lambda x.(x + 1)) \ 5 \]

Functions are computed via \( \beta \)-reduction rule: replace a \( \lambda \) abstracted variable with a term, e.g.,

\[ (\lambda x.(x + 1)) \ 5 \rightarrow_{\beta} 5 + 1. \]
Untyped $\lambda$-calculus

- The language of $\lambda$-terms:

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

Here $x$ denotes a variable.
- The variable $x$ in $\lambda x.t$ binds the occurrences of $x$ in $t$.
- The term $(s \ t)$ is an application, which applies the function $s$ to a term $t$.
- Applications associate to the left, e.g., $(((f \ x) \ y) \ z)$ is written simply as $(f \ x \ y \ z)$.
- Application has higher precedence than $\lambda$-abstraction, e.g., $(\lambda x.f \ x)$ is really $(\lambda x.(f \ x))$. 
Free variables

- Free variables of a $\lambda$-term is defined as follows:
  - $FV(x) = \{x\}$
  - $FV(s \ t) = FV(s) \cup FV(t)$
  - $FV(\lambda x. t) = FV(t) \setminus \{x\}$

- The variable $x$ in $\lambda x. t$ is said to be *bound*.

- The names of bound variables are not important and can be renamed without changing the meaning of a $\lambda$-term (but one has to be careful to avoid “capture” of other free names).
Substitutions

- A basic operation on $\lambda$-terms is substitution of free variables in the terms with other terms.
- A substitution $\theta$ is a mapping from variables to terms such that the domain of $\theta$, i.e., the set $\{x \mid \theta(x) \neq x\}$ is finite.
- The range of a substitution is $\text{ran}(\theta) = \{\theta(x) \mid x \in \text{dom}(\theta)\}$.
- Substitutions are often enumerated as a list of mappings, e.g., as in $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$. 
Applying substitutions to terms

- A substitution can be lifted to a mapping from terms to terms as follows (we write $t\theta$ for application of substitution to terms):
  1. $x\theta = \theta(x)$
  2. $(s \ t)\theta = (s\theta) (t\theta)$
  3. $(\lambda x. t)\theta = \lambda x. (t\theta)$ if $x$ is not free in $\text{dom}(\theta)$ and $\text{ran}(\theta)$.
  4. $(\lambda x. t)\theta = \lambda y. (t[x \mapsto y]\theta)$, if $x$ is free in $\text{dom}(\theta)$ or $\text{ran}(\theta)$, and $y$ is a new variable not free in both $\text{dom}(\theta)$ and $\text{ran}(\theta)$.

- The side condition in (4) is needed to prevent accidental capture of free variables in the range of substitution, which will lead to inconsistency of the $\lambda$-calculus.

- Composition of substitutions, denoted with $\circ$, is defined as

$$t(\theta \circ \rho) = (t\theta)\rho.$$
The reduction rules

\( \alpha \)-conversion \( \lambda x. t \to_{\alpha} \lambda y. t[x \mapsto y] \), provided \( y \notin FV(t) \).

\( \beta \)-reduction \( (\lambda x.s) t \to_{\beta} s[x \mapsto t] \).

\( \eta \)-reduction \( (\lambda x.t\ x) \to_{\eta} t \), \( x \notin FV(t) \).

- These rules can be applied anywhere inside a term.
- Terms which are \( \alpha \)-convertible are considered syntactically equal, e.g., \( \lambda x.x \) is equal to \( \lambda y.y \).
- \( s \to_{\lambda} t \) means \( s \) can be converted to \( t \) in one step using either \( \beta \) or \( \eta \) reduction.
- \( s \to_{\lambda}^{*} t \) means \( s \) can be converted to \( t \) in zero or more steps.
- \( s =_{\lambda} t \) means that there are \( t_1, \ldots, t_n \) such that

\[
\begin{align*}
s & \to_{\lambda}^{*} t_1 \leftarrow_{\lambda} t_2 \to_{\lambda}^{*} \cdots \to_{\lambda}^{*} t_n \leftarrow_{\lambda} t.
\end{align*}
\]
Capture-avoiding substitution and consistency

Let $s$ and $t$ be two $\lambda$ terms.
Let $x$ and $y$ be two variables which do not appear in $s$ and $t$. Without the capture avoiding condition, we have

$$\begin{align*}
(\lambda x \lambda y . x) &= \lambda x . ((\lambda z \lambda y . z) \ x) & \text{by } \beta\text{-rule} \\
&= \lambda y . ((\lambda z \lambda y . z) \ y) & \text{\(\alpha\)-rule} \\
&= \lambda y . ((\lambda y . z) [z \mapsto y]) & \text{\(\beta\)-rule} \\
&= \lambda y . (\lambda y . y) & \text{substitution} \\
&= \lambda y \lambda x . x & \text{\(\alpha\)-conversion}
\end{align*}$$

Then we obviously have:

$$s =_\lambda (\lambda x \lambda y . x) \quad s \ t =_\lambda (\lambda y \lambda x . x) \quad s \ t =_\lambda t.$$

That is, we can prove the equality of arbitrary terms!
A divergent term

- The untyped $\lambda$-calculus is Turing equivalent, since we can represent all recursive functions in the calculus.
- In particular, there are “non-terminating” $\lambda$-term, e.g., the following term:

  $$Ω ≡ (\lambda x.x \ x) \ (\lambda x.x \ x)$$

  Notice that applying the $\beta$-rule, we have

  $$Ω →_\beta Ω.$$ 

- Since we want to use $\lambda$-calculus to represent syntactic structures, we do not need the full power of $\lambda$-calculus.
- We shall use a “typed” version of $\lambda$-calculus, for which reduction always terminates and every term has a unique “normal form”.
Simply typed $\lambda$-calculus

- A *type* is essentially a set of terms.
- A *simple type* can be either a *base type* (i.e., a set of values, e.g., integers, strings, etc.) or a *function type*. Formally,

$$\alpha, \beta := a \mid \alpha \to \beta$$

where $a$ is a base type.
- The arrow $\to$ is a *type constructor*.
- A *simply typed* $\lambda$-terms are $\lambda$-terms where the variables are decorated with simple types.

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

- A *typing environment* is a set of pairs of variables and types, i.e., pairs of the form $x : \tau$ where $\tau$ is a type.
Typing judgments

We consider only well-typed terms. This is determined by the following type inference rules. In the typing judgment $\Gamma \vdash t : \tau$, $\Gamma$ is a typing environment.

\[
\begin{align*}
\Gamma, x : \tau & \vdash x : \tau \\
\Gamma, x : \alpha & \vdash t : \beta \\
\Gamma & \vdash \lambda x_\alpha . t : \alpha \rightarrow \beta & x \text{ not free in } \Gamma \\
\Gamma & \vdash s : \alpha \rightarrow \beta & \Gamma & \vdash t : \alpha \\
\Gamma & \vdash (s \ t) : \beta
\end{align*}
\]

Theorem (Type preservation)

If $\Gamma \vdash s : \tau$ and $s \rightarrow^* t$, then $\Gamma \vdash t : \tau$. 
Properties of simply typed $\lambda$-calculus

**Strong Normalisation.** Every well-typed term can be reduced, in finitely many steps, to a normal term which cannot be reduced further. The order of reductions does not matter.

**Church-Rosser** If $s \rightarrow^*_\lambda s_1$ and $s \rightarrow^*_\lambda s_2$ then there exists a term $t$ such that $s_1 \rightarrow^*_\lambda t$ and $s_2 \rightarrow^*_\lambda t$.

**Unique Normal Form.** Every well-typed term has a unique normal form.
A term $t$ is in $\lambda$-normal form (also called $\beta\eta$ long normal form) if it can be written in the form

$$\lambda \vec{x}.(u \ t_1 \cdots t_m)$$

where $u$ is either a constant or a variable of type $\sigma_1 \to \cdots \to \sigma_m \to \tau$ and $\tau$ is a base type (i.e., no arrows in it).

Exercise: Every simply typed term $t$ can be converted into $\lambda$ normal form.

In the next lecture, we shall see that all first-order structures can be encoded as a $\lambda$-term in $\lambda$-normal form.