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.

$$\text{Programming} = \text{Logic} + \text{Control} + \text{I/O} + \text{Modules}$$

$$+ \text{Concurrency} + \text{Object-oriented} + \ldots.$$

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

$$\text{Programming} = \text{Logic}$$

- 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, λ-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 λProlog.
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 Γ, show that a *query* F follows from Γ.
- The difference lies in a particular view of logical connectives as *instructions* for search.
- To prove $\Gamma \vdash A \wedge B$ 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 ($\exists$ or $\forall$), e.g., $\forall$ 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:

$$P \vee (Q \Rightarrow R) \equiv Q \Rightarrow (P \vee R) \qquad P \vee (\forall x.Q) \equiv \forall x.(P \vee Q)$$

  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

- Intuitionistic logic is a branch of logic that orignated 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: $A \vee B$ ("A or B") is true if A is true or B is true.
- Intuitionistic: $A \vee B$ 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

$$\Gamma \longrightarrow A$$

  where Γ is a *multi-set* of formulae, and A is a formula. Its intuitive reading is that "A follows from Γ".
- We call Γ 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

$$\frac{}{\Gamma, A \longrightarrow A}\ id \qquad \frac{\Gamma \longrightarrow A \quad A, \Delta \longrightarrow B}{\Gamma, \Delta \longrightarrow B}\ cut$$

- 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:

$$\frac{A, \Gamma \longrightarrow C}{A \wedge B, \Gamma \longrightarrow C} \wedge_L \qquad \frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B} \wedge_R$$

Complete list of logical rules

$$\frac{}{\bot \longrightarrow C}\ \bot_L \qquad \frac{}{\Gamma \longrightarrow \top}\ \top_R$$

$$\frac{A, \Gamma \longrightarrow C}{A \wedge B, \Gamma \longrightarrow C}\ \wedge_L \qquad \frac{B, \Gamma \longrightarrow C}{A \wedge B, \Gamma \longrightarrow C}\ \wedge_L \qquad \frac{\Gamma \longrightarrow A \quad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge_R$$

$$\frac{A, \Gamma \longrightarrow C \quad B, \Gamma \longrightarrow C}{A \vee B, \Gamma \longrightarrow C}\ \vee_L \qquad \frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\ \vee_R \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\ \vee_R$$

$$\frac{\Gamma \longrightarrow A \quad B, \Gamma \longrightarrow C}{A \Rightarrow B, \Gamma \longrightarrow C}\ \Rightarrow_L \qquad \frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \Rightarrow B}\ \Rightarrow_R$$

Structural rules

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

$$\frac{\Gamma, A, A \longrightarrow \Delta}{\Gamma, A \longrightarrow \Delta}\ c_L \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, A \longrightarrow \Delta}\ w_L$$

- 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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \longrightarrow A}\,id \quad \dfrac{}{A, B \longrightarrow B}\,id}{A, B \longrightarrow A \wedge B}\,\wedge_R}{A, B \wedge C \longrightarrow A \wedge B}\,\wedge_L}{A \wedge C, B \wedge C \longrightarrow A \wedge B}\,\wedge_L}{A \wedge C, B \wedge C \longrightarrow (A \wedge B) \vee C}\,\vee_R$$

A shorter derivation:

$$\dfrac{\dfrac{\dfrac{}{C, B \wedge C \longrightarrow C}\,id}{A \wedge C, B \wedge C \longrightarrow C}\,\wedge_L}{A \wedge C, B \wedge C \longrightarrow (A \wedge B) \vee C}\,\vee_R$$

Example: the need for the contraction rule

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \longrightarrow p}\, id}{p \longrightarrow p \vee (p \Rightarrow q)}\, \vee_R \quad \dfrac{}{q, p \longrightarrow q}\, id}{(p \vee (p \Rightarrow q)) \Rightarrow q, p \longrightarrow q}\, \Rightarrow_L}{(p \vee (p \Rightarrow q)) \Rightarrow q \longrightarrow p \Rightarrow q}\, \Rightarrow_R}{(p \vee (p \Rightarrow q)) \Rightarrow q \longrightarrow p \vee (p \Rightarrow q)}\, \vee_R \quad \dfrac{}{(p \vee (p \Rightarrow q)) \Rightarrow q, q \longrightarrow q}\, id}{(p \vee (p \Rightarrow q)) \Rightarrow q, (p \vee (p \Rightarrow q)) \Rightarrow q \longrightarrow q}\, \Rightarrow_L$$

$$\dfrac{(p \vee (p \Rightarrow q)) \Rightarrow q, (p \vee (p \Rightarrow q)) \Rightarrow q \longrightarrow q}{(p \vee (p \Rightarrow q)) \Rightarrow q \longrightarrow q}\, c_L$$

Cut elimination

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

$$\frac{\Gamma \longrightarrow B \quad B, \Gamma \longrightarrow A}{\Gamma \longrightarrow A}\ cut$$

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

$$\dfrac{\dfrac{\vdots}{\Gamma \longrightarrow A} \quad \dfrac{\vdots}{\Gamma \longrightarrow B}\ \wedge_R \quad \dfrac{\dfrac{\vdots}{A, \Gamma \longrightarrow C}}{A \wedge B, \Gamma \longrightarrow C}\ \wedge_L}{\Gamma \longrightarrow C}\ cut$$

This can be transformed to:

$$\dfrac{\dfrac{\vdots}{\Gamma \longrightarrow A} \quad \dfrac{\vdots}{A, \Gamma \longrightarrow C}}{\Gamma \longrightarrow C}\ cut$$

Consequences of cut elimination

- Syntactic consistency: The formula $\bot$ is not derivable.
- Subformula property: If a sequent $\Gamma \longrightarrow C$ is provable, then every sequent appearing in the proof uses only subformulae of Γ 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.

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., $f(x) = x + 1$ is represented as $\lambda x.(x + 1)$.
- The notation λx is called a λ-abstraction.
- Function application, such as $f(5)$, is represented via application

$$(\lambda x.(x + 1))\ 5$$

- Functions are computed via β-*reduction* rule: replace a λ abstracted variable with a term, e.g.,

$$(\lambda x.(x + 1))\ 5 \rightarrow_\beta 5 + 1.$$

Untyped λ-calculus

- The language of λ-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 occurrencs 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 λ-abstraction, e.g., $(\lambda x.f\ x)$ is really $(\lambda x.(f\ x))$.

Free variables

- Free variables of a λ-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 λ-term (but one has to be careful to avoid "capture" of other free names).

Substitutions

- A basic operation on λ-terms is substitution of free variables in the terms with other terms.
- A substitution θ is a mapping from variables to terms such that the *domain* of θ, i.e., the set

$$\{x \mid \theta(x) \neq x\}$$

  is finite.
- The *range* of a substitution is

$$ran(\theta) = \{\theta(x) \mid x \in dom(\theta)\}.$$

- 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 $dom(\theta)$ and $ran(\theta)$.
  4. $(\lambda x.t)\theta = \lambda y.(t[x \mapsto y]\theta)$, if x is free in $dom(\theta)$ or $ran(\theta)$, and y is a new variable not free in both $dom(\theta)$ and $ran(\theta)$.
- 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 λ-calculus.
- Composition of substitutions, denoted with $\circ$, is defined as

$$t(\theta \circ \rho) = (t\theta)\rho.$$

The reduction rules

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

β-reduction $(\lambda x.s)\ t \to_\beta s[x \mapsto t]$.

η-reduction $(\lambda x.t\ x) \to_\eta t$, $x \notin FV(t)$.

- These rules can be applied anywhere inside a term.
- Terms which are α-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 β or η 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

$$s \to_\lambda^* t_1 \leftarrow_\lambda^* t_2 \to_\lambda^* \cdots \to_\lambda^* t_n \leftarrow_\lambda^* t.$$

Capture-avoiding substitution and consistency

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

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

Then we obviously have:

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

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

A divergent term

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

$$\Omega \equiv (\lambda x.x\,x)\ (\lambda x.x\,x)$$

Notice that applying the β-rule, we have

$$\Omega \to_\beta \Omega.$$

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

Simply typed λ-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* λ-terms are λ-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 τ is a type.

Typing judgments

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x_\alpha . t : \alpha \to \beta}\ x \text{ not free in } \Gamma$$

$$\frac{\Gamma \vdash s : \alpha \to \beta \quad \Gamma \vdash t : \alpha}{\Gamma \vdash (s\ t) : \beta}$$

Theorem (Type preservation)

If $\Gamma \vdash s : \tau$ and $s \longrightarrow^* t$, then $\Gamma \vdash t : \tau$.

Properties of simply typed λ-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 $s \to_\lambda^* s_1$ and $s \to_\lambda^* s_2$ then there exists a term t such that $s_1 \to_\lambda^* t$ and $s_2 \to_\lambda^* t$.

Unique Normal Form. Every well-typed term has a unique normal form.

λ normal form

- A term t is in λ-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 τ is a base type (i.e., no arrows in it).
- Exercise: Every simply typed term t can be converted into λ normal form.
- In the next lecture, we shall see that *all* first-order structures can be encoded as a λ-term in λ-normal form.