Programming in Higher-Order Logic

Lecture 5

Alwen Tiu

The Australian National University

ANU Logic Summer School

Outline

- A logical notion of modules and abstraction.
- Hereditary Harrop formulae.
- Structured operational semantics (SOS).
- Encoding SOS in λProlog.

Recursion over structures with bindings

- Consider the encoding of untyped λ terms in Horn logic:

$$app : tm \to tm \qquad\qquad abs : (tm \to tm) \to tm.$$

  Suppose we want to write a predicate that recognize an untyped λ-term, how do we do it?
- The first problem is that of recognizing a variable: is x a λ-term? A related problem is recognizing an abstraction: how do we know that $abs(\lambda x.t)$ is an encoding of an untyped λ term?
- In this encoding, we cannot form a formal statement that says "x is a variable" (and there is a good logical reason to this).

Hypothetical and generic judgments

- Recursion over binders can be captured using a form of *generic* and *hypothetical* judgments, e.g.:

  > *To recognize abs*$(\lambda x.t)$*, we show that, given a* new constant c*, and* assuming that c is an encoding of an untyped λ-term, *the term* $t[x \mapsto c]$ *is an encoding of an untyped* λ*-term.*

- The constant c must be a new constant, hence embodying the notion of genericity: the statement holds not by virtue of a particular value of c, but independently of it.
- This notion of genericity is built-in in the sequent calculus, via the introduction rule for universal quantifier.
- Formally, the above hypothetical and generic judgments can be stated as

$$term\ (abs\ (\lambda x.t)) \equiv \forall x(term\ x \Rightarrow term\ t).$$

Information hiding and modular programming

- Incorporating implication and universal quantification allows us to capture a notion of modules and abstraction.
- Recall the proof rules for $\forall$ and $\Rightarrow$ on the right-hand side of sequents:

$$\frac{\Sigma, c : \tau; \Gamma \longrightarrow G[x \mapsto c]}{\Sigma; \Gamma \longrightarrow \forall_\tau x.G} \ \forall_R, c \text{ new} \qquad \frac{\Sigma; \Gamma, D \longrightarrow G}{\Sigma; \Gamma \longrightarrow D \Rightarrow G} \ \Rightarrow_R$$

- $\forall$ can be used to hide the details of c.
- $\Rightarrow$ can be used to augment the "program" Γ with a new program D (i.e., loading a module D).

Hereditary Harrop formulae

- We consider an extension of Horn clauses which allows universal quantifier and implication in goal formule.
- This class of formulae is called *hereditary Harrop (HH) formulae*:

$$
\begin{array}{l}
D ::= A_r \mid G \Rightarrow A_r \mid \forall x.D \mid D \wedge D \\
G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \forall x.G \mid \exists x.G \mid D \Rightarrow G.
\end{array}
$$

  Here A is an atomic formula and A_r is a *rigid* atomic formula.
- The D-formulae are called program clauses and the G-formulae are goal formulae.
- We call the fragment of higher-order logic restricted to HH formulae as *HH-logic*.

An idealized interpreter for HH-logic

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 this interpreter:

- **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.
- **AUGMENT** Reduce $\Sigma; \mathcal{P} \longrightarrow B_1 \Rightarrow B_2$ to $\Sigma; \mathcal{P}, B_1 \longrightarrow B_2$.
- **GENERIC** Reduce $\Sigma; \mathcal{P} \longrightarrow \forall_\tau x.B$ to $\Sigma, c : \tau; \mathcal{P} \longrightarrow B[c/x]$, where c is a "new constant".
- **TRUE** The $\Sigma; \mathcal{P} \longrightarrow \top$ is provable immediately.

Classical logic and modular programming

- For the Horn fragment, classical and intuitionistic provability co-incides, but not so for the HH fragment.
- Goal-directed search for HH formulae is not complete w.r.t. classical logic, e.g.,

  $$(p \Rightarrow q) \vee p$$

  is a classical tautology but does not have uniform provability. That is, it is not the case that $p \Rightarrow q$ is provable or p is provable.
- So logic programming based on classical logic does not naturally support modular programming.

Completeness of goal-directed search for HH logic

Theorem

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

Example: germs in a jar

Consider a program encoding the following knowledge:

- A jar is sterile if every germ in it is dead.
- A germ in a heated jar is dead.
- A particular jar is heated.

```
sterile J :-
   pi x\ germ x => in x J => dead x.

dead X :-
   sigma J\ heated J , germ X, in X J.

heated j.
```

Now consider the query "Is there a sterile jar?"

Example: reversing a list

Use an accumulator to implement "reverse".

```
rev1 L K :-
  pi rv\
  (
    (pi M\ rv nil M M),
    (pi X\ pi M\ pi N\ pi R\ rv (X::M) R N :- rv M (X::R) N)
  )
  => rv L nil K.
```

The "implementation" (the predicate `rv`) is hidden via universal quantification and is loaded only when proving `reverse`.

Example: another implementation of "reverse"

Consider the following program:

```
rv (X::L) R :- rv L (X::R).
```

Start from (`rv L nil`) and do a series of backchaining to get to the formula (`rv nil K`). If this can be done then `K` is the reverse of `L`.
This idea is used in the following `reverse` program:

```
rev2 L K :-
   pi rv\
      (pi X\ pi P\ pi Q\ rv (X :: P) Q :- rv P (X :: Q))
      => rv nil K
      => rv L nil.
```

Example: reasoning about `reverse`

We show that reverse is symmetric. Suppose that `(rev2 L K)` is provable. Then

$$\vdash \forall rv.(\forall X \forall P \forall Q. rv\ (X :: P)\ Q \Leftarrow rv\ P\ (X :: Q)) \Rightarrow rv\ nil\ K \Rightarrow rv\ L\ nil$$

Now, instantiate rv with $\lambda x \lambda y. \neg(rv\ y\ x)$.
Then we have

$$(\forall X \forall P \forall Q. \neg(rv\ (X :: P)\ Q) \Leftarrow \neg(rv\ P\ (X :: Q))) \Rightarrow \neg(rv\ nil\ K) \Rightarrow \neg(rv\ L\ nil)$$

Hence, by the contrapositive law, we have

$$(\forall X \forall P \forall Q. rv\ P\ (X :: Q) \Leftarrow rv\ (X :: P)\ Q) \Rightarrow (rv\ L\ nil \Rightarrow rv\ nil\ K)$$

We can now universally generalize on rv to get to `(rev2 K L)`.
Note: this is "classical" reasoning, but it is valid intuitionistically (why?).

Structured operational semantics

- *Structured operational semantics* [Plotkin'81, Kahn'87] is an abstract description of computation systems, formulated as a deduction system.
- The idea is to specify the meaning of programming constructs without reference to implementation details of the underlying machineries (compiler, abstract machine, etc).
- The so-called *big step semantics* specifies the evaluation of a functional program to a value (i.e., a result of computation), e.g., the "if-then-else" construct can be specified as:

$$\frac{b \Downarrow \mathrm{true} \quad s \Downarrow v}{(\mathrm{if}\ b\ \mathrm{then}\ s\ \mathrm{else}\ t) \Downarrow v} \qquad \frac{b \Downarrow \mathrm{false} \quad t \Downarrow v}{(\mathrm{if}\ b\ \mathrm{then}\ s\ \mathrm{else}\ t) \Downarrow v}$$

- These evaluation rules can be encoded as program clauses, e.g.,

$$\forall b \forall s \forall t \forall v ((b \Downarrow \mathrm{true} \wedge s \Downarrow v) \Rightarrow (\mathrm{if}\ b\ \mathrm{then}\ s\ \mathrm{else}\ t) \Downarrow v)$$

A mini functional language

- Consider a functional language based on simply typed λ-calculus, extended with arithmetic and boolean operators, and a *fixed point* operator, for encoding recursion. Call this language miniFP.
- Type expressions of miniFP:

$$\tau ::= \mathrm{integer} \mid \mathrm{bool} \mid \tau_1 \rightarrow \tau_2.$$

- In addition to the standard λ calculus operators, we have special operators:
  - "If-then-else" operator.
  - Equality on integers and boolean values; the greater-than relation.
  - Arithmetic operators: $+$, $-$ and $\times$.
- We do not allow partial applications of special operators, e.g., a term like $(+\ m)$ (but we do allow $\lambda n.(+\ m\ n)$).

The term language of miniFP

Terms:

$$
\begin{array}{rl}
s, t & ::= C \mid A \mid B \mid x \mid \lambda x.t \mid (s\ \ t) \mid \mathrm{fix}\ x.t \mid \mathrm{if}\ B\ \mathrm{then}\ s\ \mathrm{else}\ t \\
C & ::= n \mid \mathrm{true} \mid \mathrm{false} \\
A & ::= (s + t) \mid (s - t) \mid (s \times t) \\
B & ::= \mathrm{true} \mid \mathrm{false} \mid (s = t) \mid (s > t) \mid (s \wedge t) \mid (s \vee t) \mid (\neg s).
\end{array}
$$

where n is an integer (.., -2,-1,0,1,2,...) .
Not all terms formed using this grammar are allowed; we shall use a type system to restrict allowed terms.
A term is a value if it is either an integer, a boolean value, or a λ-abstraction.

The big-step operational semantics (1)

- The judgment $t \Downarrow v$ means t evaluates to value v.
- The "initial" rule (values evaluate to values):

$$\frac{}{v \Downarrow v} \; v \text{ is a value}$$

- Rules for arithmetic:

$$\frac{s \Downarrow m \quad t \Downarrow n}{(s * t) \Downarrow v} \; v = m * n$$

  where $*$ is either $+$, $-$ or $\times$.
- Equality:

$$\frac{s \Downarrow v_1 \quad t \Downarrow v_2}{(\Downarrow s = t) v}$$

  where v_1 and v_2 are both integers or both booleans, and v is true if v_1 and v_2 are of the same value, otherwise it is false.

The big-step operational semantics (2)

- Greater-than:

$$\frac{s \Downarrow m \quad t \Downarrow n}{s > t \Downarrow v}$$

where m and n are integers and v is true if m is greater than n, otherwise it is false.

- If-then-else:

$$\frac{b \Downarrow \mathrm{true} \quad s \Downarrow v}{(\mathrm{if}\ b\ \mathrm{then}\ s\ \mathrm{else}\ t) \Downarrow v} \qquad \frac{b \Downarrow \mathrm{false} \quad t \Downarrow v}{(\mathrm{if}\ b\ \mathrm{then}\ s\ \mathrm{else}\ t) \Downarrow v}$$

- Boolean operators:

$$\frac{s \Downarrow v_1 \quad t \Downarrow v_2}{(s * t) \Downarrow v} \qquad \frac{s \Downarrow v'}{\neg s \Downarrow v}\ v = \neg v'$$

The binary operator $*$ can be $\wedge$ or $\vee$ and the value v in this case is obtained by taking, resp., the conjunction and disjunction of v_1 and v_2.

The big-step operational semantics (3)

- Application:

$$\frac{s \Downarrow (\lambda x.r) \quad t \Downarrow u \quad r[x \mapsto u] \Downarrow v}{(s\ t) \Downarrow v}$$

- Fixed point:

$$\frac{t[x \mapsto \mathrm{fix}\ f.t] \Downarrow v}{(\mathrm{fix}\ f.t) \Downarrow v}$$

Example: factorial

The factorial program can be defined via fixed point:

$$\text{fix } f.(\lambda n.\text{if } (n > 0) \text{ then } (f\ (n-1)) \times n \text{ else } 1).$$

Now derive the following:

$$(\text{fix } f.(\lambda n.\text{if } (n > 0) \text{ then } (f\ (n-1)) \times n \text{ else } 1))\ 3 \Downarrow 6$$

Type system for miniFP

- The evaluation relation $t \Downarrow v$ may not be defined for all term t and value v, but only for those which are well-typed, e.g.,

$$(\lambda x.x) = (\lambda x.x)$$

  cannot be evaluated.
- The type structures for miniFP are essentially those of simply typed λ-calculus, but with polymorphism for abstractions, equality and fixed points.
- The typing judgment is of the form

$$\Gamma \vdash t : \tau$$

  where Γ is a typing environment (pairs of variables and types).

The typing rules (1)

The rules for simply typed λ-calculus, plus the following:

$$\frac{}{\Gamma \vdash \mathrm{true} : \mathrm{bool}} \qquad \frac{}{\Gamma \vdash \mathrm{false} : \mathrm{bool}} \qquad \frac{n \text{ is an integer}}{\Gamma \vdash n : \mathrm{integer}}$$

$$\frac{\Gamma \vdash s : \mathrm{bool} \quad \Gamma \vdash t : \mathrm{bool}}{\Gamma \vdash s = t : \mathrm{bool}} \qquad \frac{\Gamma \vdash s : \mathrm{integer} \quad \Gamma \vdash t : \mathrm{integer}}{\Gamma \vdash s = t : \mathrm{bool}}$$

$$\frac{\Gamma \vdash s : \mathrm{integer} \quad \Gamma \vdash t : \mathrm{integer}}{\Gamma \vdash s > t : \mathrm{bool}} \qquad \frac{\Gamma \vdash s : \mathrm{integer} \quad \Gamma \vdash t : \mathrm{integer}}{\Gamma \vdash s * t : \mathrm{integer}} \; * \in \{+, -, \times\}$$

The typing rules (1)

$$\frac{\Gamma \vdash s : \mathrm{bool} \quad \Gamma \vdash t : \mathrm{bool}}{\Gamma \vdash s \bullet t : \mathrm{bool}} \; \bullet \in \{\wedge, \vee\} \qquad \frac{\Gamma \vdash s : \mathrm{bool}}{\Gamma \vdash \neg s : \mathrm{bool}}$$

$$\frac{\Gamma \vdash b : \mathrm{bool} \quad \Gamma \vdash s : \alpha \quad \Gamma \vdash t : \alpha}{\Gamma \vdash \mathrm{if}\ b\ \mathrm{then}\ s\ \mathrm{else}\ t : \alpha}$$

$$\frac{\Gamma, f : \alpha \vdash t[x \mapsto f] : \alpha}{\Gamma \vdash \mathrm{fix}\ x.t : \alpha} \; f \text{ new}$$

Exercise: show that the factorial program

$$\mathrm{fix}\ f.(\lambda n.\mathrm{if}\ (n > 0)\ \mathrm{then}\ (f\ (n-1)) \times n\ \mathrm{else}\ 1)$$

has type $\mathrm{integer} \rightarrow \mathrm{integer}$.

Encoding miniFP in λProlog: types and terms

```
kind tm type. % syntactic category for miniFP terms
kind ty type. % syntactic category for miniFP types

type bool ty. % boolean type
type integer ty. % integer type
type arr ty -> ty -> ty. % arrow type constructor

type tt, ff tm.   % true and false
type i int -> tm. % integers

% abstraction and application
type abs (tm -> tm) -> tm. % abstraction
type @ tm -> tm -> tm.  % application
infixl @ 120.     % @ is infix, and associate to the left.
```

Encoding miniFP in λProlog: types and terms

```
% equality and greater-than
type eq, gt tm -> tm -> tm.

% if-then-else
type ite tm -> tm -> tm -> tm.

% boolean
type and, or tm -> tm -> tm.
type neg tm -> tm.

% arithmetic
type sum, minus, times tm -> tm -> tm.

% fixed point operator
type fix (tm -> tm) -> tm.
```

Encoding of the operational semantics

```
% values
eval tt tt.
eval ff ff.
eval (i I) (i I).
eval (abs M) (abs M).

% applications
eval (M @ N) V :- eval M (abs F), eval N T, eval (F T) V.

% fixed points
eval (fix F) V :- eval (F (fix F)) V.

% special constants
eval (sum M N) (i V) :-
  eval M (i V1), eval N (i V2), V is V1 + V2.
eval (minus M N) (i V) :-
  eval M (i V1), eval N (i V2), V is V1 - V2.
eval (times M N) (i V) :-
  eval M (i V1), eval N (i V2), V is V1 * V2.
```

......

Encoding of the type system

- In encoding the typing judgment $\Gamma \vdash t : \tau$, we encode implicitly the typing environment Γ as hypotheses in a sequent.

```
type typeof tm -> ty -> o.
```

- For example, the typing judgment $x : \alpha, y : \beta \vdash t : \tau$ corresponds to provability of the sequent

$$\mathit{typeof}\ \ x\ \alpha, \mathit{typeof}\ \ y\ \beta \longrightarrow \mathit{typeof}\ \ t\ \tau$$

- Rules that augment the typing environment, e.g.,

$$\frac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta}$$

are encoded via hypothetical and generic judgment, e.g.,

```
typeof (abs M) (arr A B) :- pi x\ typeof x A => typeof (M x) B.
```

Example: factorial

```
type prog string -> tm -> o.

% Factorial
prog "fact"
(fix fact\ abs n\
  ite (gt n (i 0))
      (times n (fact @ (minus n (i 1))))
      (i 1)
 ).
```

An example query:

```
[minifp] ?- prog "fact" F, eval (F @ (i 5)) V.

The answer substitution:
V = i 120
F = fix (W1\ abs (W2\ ite (gt W2 (i 0)) ......
```

Example: another factorial program

```
% Tail-recursive factorial: the recursive call to "fact" is the
% "last" call.

prog "trec-fact"
(fix fact\ abs n\ abs m\
  ite (eq n (i 0))
      m
      (fact @ (minus n (i 1)) @ (times n m))
).
```

An example query:

```
[minifp] ?- prog "trec-fact" F, eval (F @ (i 5) @ (i 1)) V.
```

```
The answer substitution:
V = i 120
F = fix (W1\ abs (W2\ abs (W3\ ite (eq W2 (i 0)) .......
```

Tail-recursion

- A tail-recursive program is a recursive program where the recursive call happens after all other function calls.
- More precisely:
  - A program is tail-recursive if it contains no recursive calls.
  - A program that consists solely of a recursive call (with possibly modified arguments) is tail-recursive.
  - A program is tail-recursive if its body consists of a conditional (if-then-else) where the test contains no recursive calls and where the left and right branches satisfy the tail-recursiveness requirement.
- Tail-recursive programs can be turned into iterative programs with no need to save call stacks.

Example: recognizing tail-recursion

```
type trec (tm -> tm) -> o.

trec (f\ f).
trec (f\ M).
trec (f\ abs (M f)) :- pi x\ trec (f\ M f x).
trec (f\ (M f) @ N) :- trec M.
trec (f\ ite B (P f) (Q f)) :- trec P, trec Q.

tailrec (fix F) :- trec F.
```

Example: meta-reasoning about logic programs

Theorem (Type Preservation)

If $\vdash \mathrm{eval}\ P\ V$ *and* $\vdash \mathrm{typeof}\ P\ T$ *then* $\vdash \mathrm{typeof}\ V\ T$.

Proof.

By structural induction on uniform proofs and cut elimination. □

Summary

- Logic programming as goal-directed search: Completeness of goal-directed search as a defining criteria.
- λ-tree syntax: A uniform notation for representing syntactic structures.
- Meta-reasoning about properties of programs can benefit from the meta theory of logic, e.g., cut elimination, substitution lemma.

Other extensions

- Using *linear logic* as the logical foundation.
- Encoding a notion of *resources* via linear implication and linear conjunction (the $\otimes$ operator). Example: the Lolli programming language [Miller & Hodas], based on intuitionistic linear logic.
- Concurrent goals: using multiple-conclusion sequents to model concurrent computation. Example: the Forum programming language [Miller].

Focussed proofs and encoding of computation

- Uniform proofs are a special case of *focussed proofs*.
- The basic idea is to structure proofs into two phases: the *asynchronous* phase and the *synchronous* phase.
- All of linear/intuitionistic/classical logic admit a complete focussed proof search strategy.
- Focussed proofs may not be goal-directed, but one still retain the correspondence:

$$\textit{Computation} = \textit{Focussed proof search}$$

- As in the case with goal-directed search, representation of (models of) computation as focussed proofs allows for a logical analysis of computation using logical tools.

Reasoning about computation

- Mechanized reasoning about (logic) programming languages:
  - Bedwyr (INRIA, U. Minnesotta, ANU): an automated reasoning tool for proving simple properties of Horn logic programs.
  - Abella (U. Minnesotta): an interactive theorem prover that can reason about properties of λProlog programs.
- Meta-logical frameworks for reasoning about computation: need expressive proof principles, e.g., induction and co-induction, support for datatypes for binders, etc.