CSci 8980, Fall 2012
Specifying and Reasoning About Computational Systems
Higher-Order Logic Programming

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Fall 2012

The Lambda Calculus and Higher-Order Logic

Some important perspectives for the present setting

- We will take a logical rather than a computational view of lambda calculus
  - Equality between terms will be the primary notion
  - Reduction will mainly be a vehicle for determining equality
- Lambda calculus will help in building a higher-order logic
  - lambda terms will figure as the arguments of predicates
  - lambda terms will also be used to construct complex formula-valued expressions that are quantified over
- Lambda terms will mainly be used to *represent* objects
  - The calculus strength must be calibrated to analyzing syntax structure
  - Stronger (non-syntactic, computational) properties of objects will be expressed via the relational framework

Building a Higher-Order Logic

The lambda calculus provides a primitive understanding of functionality

Can we build a logic of predicates, connectives and quantifiers over this?

The General Idea

- Designate particular constants from $\mathcal{C}$ to represent connectives and quantifiers
- Build in an interpretation of these symbols through sequent calculus or natural deduction rules

The thought is quite seductive: if it works we would get a *foundation* for mathematics

Untyped Lambda Calculus Logic is Inconsistent

Unfortunately, the idea *does not* work

In particular, any logical system that includes the following rules

$$\frac{X \qquad X =_{\lambda\beta\eta} Y}{Y}\ EQ$$

$$\frac{\begin{matrix}[X]\\ \vdots \\ Y\end{matrix}}{X \supset Y} \supset I \qquad\qquad \frac{X \qquad X \supset Y}{Y} \supset E$$

is inconsistent

These rules are so basic that the project seems doomed

Curry's Paradox

Curry showed that it is possible to construct a derivation for any formula M in such a logic

By the properties of the untyped lambda calculus, we know that there is a term X such that $X =_{\lambda\beta\eta} X \supset M$

We then construct the following proof π of $X \supset M$

$$\dfrac{\dfrac{X \qquad \dfrac{X \qquad X =_{\lambda\beta\eta} X \supset M}{X \supset M}\,EQ}{M}\,\supset E}{X \supset M}\,\supset I$$

We now complete the proof of M as follows

$$\dfrac{\dfrac{\pi \quad X \supset M =_{\lambda\beta\eta} X}{X}\,EQ \qquad \pi}{M}\,\supset E$$

The Simple Theory of Types

Church's resolution to this problem was to introduce types

The main formal purpose of types was to recognize a functional hierarchy

More specifically

- we have a type for propositions and possibly several atomic types
- we form function types from other known types

These types restrict the kinds of lambda terms we can form and thereby avoid the paradoxes

The resulting logic is not foundational but still is good for formalization

Following this recipe results in a class of logics that we will generically call the *Simple Theory of Types*

Building a Typed Higher-Order Logic

The logic is constructed by giving the type o a special status

Terms of this type are identified as *formulas* from which we form sequents

We also distinguish specific constants over the type o used to represent connectives and quantifiers

$\vee, \wedge, \supset$	of type `o -> o -> o`
$\top, \bot$	of type `o`
$\Pi_\alpha, \Sigma_\alpha$	of type `(α -> o) -> o`

Here Π_α and Σ_α represent a *family* of constants, one for each type α

These constants are called *generalized quantifiers*

We will often omit the type subscript when we want to talk of these constants collectively

Representing the Familiar Quantifiers

Church's proposal: Handle binding effect through abstraction, leaving only the predication aspect to be treated specially

For example, consider representing $\forall x\ P(x)$

- First convert $P(x)$ into a "set" by abstracting over the x

  Specifically $\lambda x\ P(x)$ represents the set of things of which P is true

- Then make the quantifier a predication over this set

  $(\Pi\ (\lambda x\ P(x)))$, representing $\forall x\ P(x)$, effectively says "P is true of everything"

Similar idea works for $\exists x\ P(x)$ and also for other (non-logical) binding operators we will want to encode

In these cases, Π and Σ represent the *universal set* and the *existential set* recognizers

Realizing Substitution via β-Conversion

Advantage of such a representation: Substitution is completely and correctly realized through application and β-conversion

For example, we have the following correspondence

$$\begin{tikzcd}
\forall x\ B(x) \arrow[r, "\text{represented by}"] \arrow[d, "\text{instantiated by } t"] & \Pi\ P \quad (P = \lambda x[B(x)]) \arrow[d, "\text{applied to } [t]"] \\
B(t) \arrow[r, "\text{represented by}"] & (P\ [t])
\end{tikzcd}$$

Scope respecting substitution is implicit in the λ-conversion rules

Sequent Calculus for Higher-Order Logic

Our sequents will now contain formulas from the new language

Beyond this, the only change to the LK/LJ rules is to build in λ-conversion

Can be done in one of two ways

- Assume that matching of formulas to schemas incorporates λ-conversion
- Add one more rule of the form

$$\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta}\ \lambda$$

where Γ and Γ' and Δ and Δ' differ only by λ-conversion

We will assume the former approach

Note that Church's logic has additional "mathematical" axioms that we do not want or use in our setting

Predicate Variables and Substitution

Even though the logic looks very similar to first-order logic, its behaviour can be wildly different

The main source of difference is predicate variables and what substituting for them can do to the formula structure

For example consider the formula

$$\forall p\ ((p\ A) \supset (c\ A))$$

Instantiating the quantifier with $\lambda z((q\ z) \wedge \forall x\ ((z\ x) \supset (q\ x)))$ yields

$$((q\ A) \wedge \forall x\ ((A\ x) \supset (q\ x))) \supset (c\ A)$$

In other words, a substitution instance can have a vastly different logical structure from the original formula

This difference has a significant impact on the meta-theory and on matters such as proof search

Logic Programming in Higher-Order Logic

We would like to extend *fohh* in such a way that

- useful higher-order features become available
- the search related interpretation is preserved
- determining predicate substitutions can be done in sensible computational steps

To realize these objectives we have to restrict predicate quantification and the kinds of instantiations to be considered

The key questions become

- what atoms can appear at the heads of clauses, and
- what logical symbols can appear in predicate arguments

The Need to Restrict Predicate Quantification

Program clauses in the reduced form of *fohh* have the structure

$$\forall x_1 \ \ldots \forall x_n \ G \supset A$$

where A is an atomic formula

In the higher-order setting, A could be rigid or flexible

Letting it have a flexible head is problematic

- A clause of the form

  $$\forall p \ G \supset (p \ t_1 \ \ldots \ t_m)$$

  adds meaning to *all* relations; this is at least anti-modular

- Such clauses can easily lead to inconsistent programs

  For example, consider the following

  $\forall p \, (q \supset p) \quad \wedge \quad q$

  From this, *any* formula can be derived

We shall therefore require clause heads to be rigid atoms

The Need for Restricting Predicate Arguments

Logical symbols can appear in terms in higher-order logic

Such symbols can end up instantiating predicate variables and thereby affecting top-level formula structure

This can potentially impact the completeness of goal-directed search

For example, consider the "goal formula"

$$\exists Q \, \forall p \, \forall q \, ((r \, (p \supset q)) \supset (r \, (Q \, p \, q))) \wedge \\ (Q \, (t \vee s) \, (s \vee t))$$

This formula is provable but its proof requires solving the goal

$$(t \vee s) \supset (s \vee t)$$

for which no uniform proof exists

Disallowing $\perp$ and $\supset$ in arguments provides a static approach to ruling out such problems

Higher-Order Hereditary Harrop Formulas

A *positive* atom is a (higher-order) atomic formula of the form

$$(P \ t_1 \ \ldots \ t_n)$$

in whose arguments the symbols $\supset$ and $\perp$ do not appear

The atom is *flexible* if P is a variable and *rigid* otherwise

Let $\mathcal{G}_3$ and $\mathcal{D}_3$ be the G and D formulas given by

$$\begin{array}{lll} G & ::= & \top \mid A \mid G \vee G \mid G \wedge G \mid \exists x \ G \mid \forall x \ G \mid D \supset G \\ D & ::= & A_r \mid G \supset D \mid D \wedge D \mid \forall x \ D \end{array}$$

where A stands for a positive atom and A_r for a rigid atomic formula

The *language of higher-order hereditary Harrop formulas* or *hohh* is the triple $\langle \mathcal{D}_3, \mathcal{G}_3, \vdash_I \rangle$

Showing *hohh* is an ALPL

Let a *positive term* be any λ-term in which the symbols $\supset$ and $\perp$ do not appear

Key Lemma

Let $\mathcal{P}$ be an *hohh* program and let G be an *hohh* goal such that $\mathcal{P} \vdash G$ has an intuitionistic proof

Then $\mathcal{P} \vdash G$ must have such a proof in which in any occurrence of the rules

$$\frac{\Gamma, \forall x \ B, B[t/x] \vdash \Delta}{\Gamma, \forall x \ B \vdash \Delta} \ \forall L \qquad \frac{\Gamma \vdash B[t/x]}{\Gamma \vdash \exists x \ B} \ \exists R$$

the term t is restricted to being positive

Thus it is enough to consider proofs in which all sequents have the "normal form" that makes the first-order proof work

The proof of the lemma relies on a proof transformation

Computation via Proof Search for *hohh*

We can describe a reduced proof system for *hohh* that for the most part preserves the nature of computation for *fohh*:

- The top-level connective of a non-atomic goal determines the next step
- A generalized backchaining works for rigid atomic goals

Of course, in constructing instances, we will now have to consider *typed* λ*-terms* over a given signature

One new thing in the higher-order setting is that we have also to consider *flexible atomic goals*

The meta-theory provides a simple answer here:

- delay the solving of such goals
- eventually, when only flexible goals remain, use a “universal set substitution”

In specification logic applications, though, we will be interested in situations where such flexible goals *will not* arise