Higher Order Logic in Semantics

Stephen.Pulman@comlab.ox.ac.uk

Higher Order Logic Overview

- The Lambda Calculus
- λ-notation
- α-reduction, β-reduction, η-reduction
- The simple theory of types
- Types, self-reference and paradox

Higher order logic overview

The basic well formedness rules of the logic are as follows. They may seem unfamiliar to those who have already been exposed to first order predicate calculus, but the reason for this format, as well as an alternative, more readable syntax, will be given later.

term ::= constant | var | term(term) | λvar. term

constant ::= logical-constant | non-logical-constant

logical-constant ::= $\forall$| $\exists$| $\wedge$ | $\vee$ | $\neg$ | $\rightarrow$

($\forall$= 'for all', $\exists$= 'there exists')

non-logical-constant :: = likes | sleeps | John | Mary etc.

var ::= P | Q | R | x | y | x etc.

This logic has 'terms', rather than 'sentences'. As well as obeying these syntactic rules, each term must have a 'type', and obey certain rules about types. The basic types we shall assume to begin with are e (entity), and t (truth value). We can construct complex types by the following rule: if 'α' and 'β' are types then $\langle\alpha, \beta\rangle$ is a type. (Read 'from α to β'). Constants can be of any type.

The type of a term tells us the range of things it can legitimately denote. Thus if the type of a term is e, we know that it can denote only something which is an entity. (What we choose to allow as entities is another matter). If the type of a term is ⟨e,e⟩ then we know it can only denote something which when given an entity returns an entity. (The type is read 'type entity to entity'). That must be a function: a function from entities to entities. An example of such a function might be 'mother-of': a function which when applied to a person or animal (assuming these are entities) returns the mother, of that person or animal. Types can be as complicated as we like: we shall fairly soon encounter something of type

⟨⟨e, t⟩,⟨⟨e, t⟩,t⟩⟩
(entity to truth value) to ((entity to truth value) to truth value)

Logical constants are also treated as functions, with the usual definitions: the connective $\neg$ is of type ⟨t,t⟩, and $\neg$(P) is true if *P* is false, false otherwise. The connectives $\wedge$, $\vee$, and $\rightarrow$ are of type ⟨t,⟨t,t⟩⟩. Their syntax observes that of all other terms:

¬(sleeps(John)),
[∧ (sleeps (John))](snores(John))
etc.

However, it is easier to read the binary connectives if they are in infix form, and so we will write things like:

[∧ (sleeps (John))] (snores(John))

as the easier to read:

sleeps(John) ∧ snores(John).

We also use several other notational conventions to make expressions easier to read and write: instead of [[[**a,b**],**c**],**d**] we write [**a,b,c,d**], or use 'relational notation' **a(b,c,d)** (usually only when the type of the whole expression is t). If we have several lambda expressions: λx.λy.λz.[[[...]...]...] we sometimes write this as λxyz.[[[...]...]...]. If you know the types of the elements of such expressions there will never be any ambiguity about what their structure is.

The other connectives have their usual interpretation: $P \wedge Q$ is true only if both P and Q are; $P \vee Q$ is true only if at least one of P and Q is true, and $P \rightarrow Q$ is true unless P is true and Q is false.

We will now illustrate some examples of logical forms. By the first and second clauses of the first rule above any constant or variable is a term. Many such terms are not of interest to us: a single quantifier, for example, is a syntactically well formed term, but corresponds to no clear intuitive interpretation. The class of terms we are usually interested in are those of type *t*, corresponding roughly to sentences. We can build such terms by the third clause of the first rule, which says that two terms concatenated form a term. However, this is not quite all there is to this: the semantic interpretation of such a combination is as 'function application', where the left hand term is a function, and the right hand term its argument. Since we also require all terms to be 'well-typed', not every such concatenation will count as well formed. For a term like this to be well typed, the subterms must be well typed, and the left most term must have a type of the form $\langle\alpha,\beta\rangle$, and the rightmost term be of type α. The type of the new term formed will be type β. So in general if we have a term, say *Z*, which consists of a function *X* applied to an argument *Y*, then we can express the relationships between the various types involved as:

where *Z* is of form *X(Y)*,
type(X) = ⟨type(Y),type(Z)⟩

It is easy to see that given any two of these types we can work out what the third one is. When applying a function to an argument, one can think of the resulting type being obtained by 'cancelling out': $\langle\alpha,\beta\rangle$ and α give β.

Let us assume we have a constant *sleeps* of type ⟨e,t⟩, and a constant *John* of type *e*. The interpretation of *sleeps* is as a function from things of type *e* to things of type *t*: that is, from things denoting entities to things denoting truth values. Intuitively we can think of *sleeps* as a function which, when applied to an appropriate argument, an entity, tells you whether or not (true or false) that entity sleeps. The concatenation of these two terms, *sleeps(John)* is also a term, by the well-formedness rule, and it is also well-typed. The resulting term is of type *t*.

Consider now a constant such as *likes*. Intuitively this is a two place verb, requiring a subject and an object. But our formation rule appears not to allow such a possibility. However, consider what type *likes* should have. It should somehow combine with two things of type *e* to form something of type *t*. But our rules for forming complex types appear to have the same restriction to binary objects as our syntactic formation rule (not an accident!). The type we need must be something like ⟨e,⟨e,t⟩⟩: something which takes an entity type and makes something of type entity to truth value. Thus if we form a term *[likes (John)](Mary)* we will have something which has the type structure

⟨⟨⟨e,⟨e,t⟩⟩,e⟩,e⟩

Two 'cancellings out' give us a sentence: *likes* applies to *John* to give us something of type ⟨e,t⟩. This in turn applies to *Mary* to give us something of type *t*.

It is most natural to interpret *John* as the subject and *Mary* as the object, here, and we can regard *[likes (John)](Mary)* as saying the same thing as does *likes(John, Mary)* in more familiar logical notation. Certainly *likes(John,Mary)* is more readable, but we should, in terms of our logic, think of it simply as a convenient way of writing *[likes (John)](Mary)*. (However, if we want to give 'likes' a meaning that we will be able to apply first to an object NP to give a VP, and then have the VP apply to the subject NP, then we should translate the English word as the expression: λy.λx.likes(x,y)).

The fourth clause of the first formation rule forms 'lambda terms', sometimes called 'lambda abstracts'. Given a term like *sleeps(x)* it allows us to form a term *λx.sleeps(x)*. The intuitive interpretation of this is as a function characterising or denoting 'the set of those x such that sleeps is true of x'. We can form lambda terms from any term: all of the following are valid:

λP.P(John), where type(P) = ⟨e,t⟩: the function characterising the set of those P such that P are true of John

λQ.not(Q), where type(Q)=t: the function characterising the set of those Q such that it is not the case that Q

λR.λy.R(y) where the function characterising the set of those R which are such that they are true of y, in the inner lambda term. The inner term's interpretation is the function characterising the set of those y such that R is true of y.

The rules for typing of lambda terms are simple: in λA.B, where A is of type α and B is of type β, the lambda term is of type $\langle\alpha,\beta\rangle$. Thus if, as in the first examples, x is a variable of type e and *sleeps(x)* is a term of type t, the lambda term λx.*sleeps(x)* is a term of type $\langle$e,t$\rangle$. It follows from the other cases of the first formation rule that a term like *[λx.sleeps(x)](John)* is also a term: clearly, it is also well-typed. (Read it '(the function characterising) the set of x such that x sleeps, includes John').

We include in our higher order logic a rule of 'lambda conversion' (actually several: this one is strictly called β-reduction). Given a well-typed term of the form *[$\lambda X.Y$](Z)*, we can form a logically equivalent term by replacing any instances of X in Y by Z. Given e.g. *[λx.sleeps(x)](John)*, we can form the equivalent term *sleeps(John)* by replacing every occurrence of x in *sleeps(x)* by *John*. Here are some further examples:

([λP (P John)] sleeps) = (sleeps John)

([λP (sleeps John)] snores) = (sleeps John)

(there were no occurrences of P)

([λQ (not Q)] (sleeps John)) = (not (sleeps John))

(([λR λy (R y))] sleeps) John) = [λy (sleeps y)](John)

= (sleeps John)

Equally, of course, we can regard lambda-conversion as operating the other way round. Given a term like *[likes(John)] (Mary)* we can form the equivalent complex term *[λx.[likes (John)](x)](Mary)* .

We have to be careful with variables when dealing with lambda terms, and make sure that we only substitute in for the 'right' variables: the simplest way to do this is to make sure that each quantifier or lambda is associated with a typographically distinct variable. We assume that where two expressions differ only in their choice of variables, they are logically equivalent. We further assume that where P has a complex type $\langle\alpha,\beta\rangle$, then $P = \lambda x.P(x)$, and vice versa, where x will be of type α. (These latter two assumptions amount to the α and η rules of other presentations).

The semantics of lambda-conversion, informally, is as follows: let A be the domain of things of the type of X, and B the domain of things of the type of Y. Now, a term of the form $\lambda X.Y$ is that function from A to B such that whenever it is applied to something from

A, the result is equal to what you would get by evaluating Y, with that thing substituted for every occurrence of X in Y. In the system we have outlined, we can regard lambda-conversion as a rule of inference: with a different set of primitives, it might have been a theorem.

The reason we are operating with a somewhat counterintuitive and deeply bracketed syntax here, in which every complex (non-lambda) term is a binary concatenation, is to make the semantics and the type system more easily understandable. The simple method of forming terms by concatenation makes the corresponding interpretation as function application easier, and the same thing goes for the computation of typing consistency. (Any implementation using a logic of this type would be well advised to stick to this simple syntax, too).

But why have types at all? The answer is simple: in any logical system, we must be careful to avoid inconsistency: if we can deduce both P and $\neg(P)$ from the same set of axioms, something has gone drastically wrong (and we will be able to deduce anything at all. (How? Assume both P and $\neg(P)$. Try to prove an arbitrary Q. By the usual rules for disjunction, if we have P then we can also conclude $P \vee Q$, since if P is true, then the disjunction will be true whatever the truth value of Q. But $P \vee Q$ is equivalent to $\neg(P) \rightarrow Q$). And from $\neg(P) \rightarrow Q$ and our other assumption, $\neg(P)$, we can conclude Q).

The fourth clause of the first formation rule allows us to form terms which would be capable of leading to such inconsistency, if we did not have the constraint that terms must be well typed. Consider the well known paradoxes of the type that arise when we have a system powerful enough to describe notions like ‘the set of all sets that are not members of themselves’. This sounds like a sensible set: we can point to sets that have other sets as members (e.g. the set containing all possible sets of teaspoons) and to sets that don’t contain themselves (the set of all chairs does not contain itself: it contains only chairs), and it doesn’t seem too unlikely that we could have a set consisting of all the latter types of set. But paradox arises when we ask whether this set is a member of itself: if it is, then it isn’t a set that doesn’t contain itself, and so doesn’t satisfy our description; if it isn’t, then it is a set that doesn’t contain itself and so it should be in there somewhere. It seems that we cannot consistently have such a set. However, if we ignore the requirement that terms should be well-typed, nothing so far prevents us from forming a term that denotes exactly such a set.

What are we asking for exactly when contemplating such a set? Let P = ‘not a member of itself’ (i.e. $\lambda x.\neg member(x,x)$). Then $\lambda x.P(x)$ denotes the set of such objects. Then $[\lambda x.P(x)]\ (\lambda x.P(x))$ is the expression corresponding to the paradoxical question. But this expression, like any of the form $X(X)$, cannot possibly be well typed: the rules require, in a concatenation like this, that the left hand member of the pair be of type $\langle\alpha,\beta\rangle$, where α is the type of the right hand side member of the pair (and β is any other type). If both are identical expressions this will be impossible: something cannot have one type in one context and a different type in another. Thus the requirement that terms be well typed avoids (this kind of) paradox.

The quantifiers $\forall$ and $\exists$ are also functions: we define them as constants of type $\langle\langle\alpha,t\rangle,t\rangle$, where α is any type. What this means is that they are functions which can only apply to things which are equivalent to terms of the form λX.Y, where *X* is whatever type α is, and *Y* is of type *t*. The lambda term will thus be of type $\langle\alpha,t\rangle$. So we will be able to form things like:

$\forall$*(sleeps)* or $\forall$*(λx. sleeps(x))*

- everything sleeps, and:

$\exists$*(λx. [likes (John)](x))*

- John likes something. We define the truth conditions of $\forall$ and $\exists$ to be as follows (assuming the obvious type requirements):

$\forall$(P) is true iff P(x) is true for every value of x
$\exists$(P) is true iff P(x) is true for some value of x

Intuitively, quantifiers are functions of predicates, where these can be of arbitrary complexity, and predicates not just of individuals, but of almost anything. A term made by applying the universal quantifier to such a predicate will be true if and only if that predicate is true of every thing of the appropriate type, and a term made by applying the existential quantifier to a predicate will be true provided there is at least one thing that the predicate is true of.

For example, in a domain where John and Mary are the only individuals,

$\forall$*(λx. sleeps(x))*

will be true if both

[λx. sleeps(x)](John), i.e. *sleeps(John)*, and
[λx. sleeps(x)] (Mary), i.e. *sleeps(Mary)*

are true. Note that although in these examples, the term that the quantifier applies to is of type $\langle$e,t$\rangle$, this is not necessary: our definition allows us to form things like:

$\forall$*(λP. P(John))* (everything (of type $\langle$e,t$\rangle$) is true of John)
$\exists$*(λR. $\neg$(R))* (there is some false propositional term)
$\forall$*(λS. S(sleeps))* (everything (of type $\langle\langle$e,t$\rangle$,t$\rangle$ is true of 'sleeps')

(The type assignments here presuppose that the type of the body of the lambda expression is t).

The way we have written expressions involving quantifiers may be unfamiliar, but reflects their logical properties in the system we have developed. To make these expressions easier to read, however, we can adopt a convention that expressions like:

$\forall(\lambda X.\ Y)$
$\exists(\lambda X.\ Y)$

can also be written as:

$\forall X.\ Y$
$\exists X.\ Y$

meaning exactly the same thing.

In this logic, we are not restricted to quantifying over individuals: that is what we mean by saying the logic is higher order: we can quantify over any type of term. Less powerful logics can be got by placing restrictions on what we have at present. If we restrict $\forall$ and $\exists$ to apply only to expressions of type $\langle$e,t$\rangle$, then our logic will be more or less equivalent to standard first order predicate calculus. If we do not use λ, $\forall$, $\exists$, or terms of any type other than t, or those built from these with the connectives then we will have something more or less equivalent to propositional calculus. Conversely, we can define other well known types of logic in terms of our higher order logic. If we add an extra type, w, (possible worlds) then we can single out a set of terms of type $\langle$w,t$\rangle$, functions from worlds to truth values, which can serve as the meanings of sentences. These terms might be things like

$\lambda x.\ [sleeps(John)](x)$

where x is of type w. These terms might represent contextually dependent sentence meanings which do not form something which can be evaluated for truth until it is applied to a particular world. When such a term is applied to a particular world, say world *w23*, we will get things like:

$[sleeps(John)](w23)$

i.e. 'John sleeps in world 23'. The truth of 'John sleeps' will then vary depending on which world (or time, or context) we are referring to. We can go on to define necessity and possibility operators as follows, where p is a term of type $\langle$w,t$\rangle$:

$$\mathit{Nec(p)} = \forall x.\ \mathit{p(x)}$$
$$\mathit{Poss(p)} = \exists x.\ \mathit{p(x)}$$

The proposition p is necessarily true if it is true in all possible worlds, and it is possibly true if there is at least one world in which it is actually true.

References

D. Dowty, R. Wall, S. Peters 1981 Introduction to Montague Semantics, Dordrecht: D. Reidel

Bob Carpenter 1998 Type Logical Semantics, MIT Press. (first couple of chapters)

James Allen, 1996, Natural Language Understanding, Chap 8 and 9.

D Jurafsky and J Martin, 2008, Speech and Language Processing (Second Edition, Chapters 17 and 18.