

Higher Order Logic

The typed λ-calculus provides the theoretical foundation for a variety of technical areas. First, it is the foundation of many strongly typed programming languages. Second, it can serve as the foundation of higher order logics and is a natural and highly expressive extension of first order logic. Third, the typed λ-calculus serves as a theoretical foundation for certain theories of the syntactic and semantic structure of natural languages such as English. The typed λ-calculus is a simple yet non-trivial formal structure which arises in a wide variety of contexts.

1 Fundamental Definitions

The typed λ-calculus is a logic — it has propositions, models and a way of assigning a truth value to each proposition in each model. The typed λ-calculus also has type expressions and terms and a way of assigning a meaning to each type expression and each term in each model. We will first focus on the type expressions. The semantic value of a type expression (in a model) is a set. There is a set of *basic type symbols* and "arrow types" built out of basic type symbols. If A and B are basic type symbols then $A \rightarrow B$ is a type expression and denotes the set of all functions from the set denoted by A to the set denoted by B. If we write $f : \tau$, read as f has type τ, we mean that f denotes an element of the set denoted by the type expression τ. For example, the expression $f : A \rightarrow B$ indicates that f denotes a function from the set A to the set B. More formally, the syntax and semantics of type expressions is defined as follows.

Definition: A *type expression* is either a type symbol or an arrow type of the form $\tau_1 \times \tau_2 \times \cdots \times \tau_n \rightarrow \sigma$ where $\tau_1, \tau_2, \cdots, \tau_n$ and σ

are type expressions.

To define the semantics of type expressions we assume that a model $\mathcal{M}$ assigns a meaning to each type symbols as a set. A model of the λ-calculus provides other information as well, but the semantics of type expressions depends only the meaning of the type symbols. Let $\mathcal{M}$ be a model that contains an interpretation of each type symbol as a set.

> **Definition:** For each type symbol A, the semantic value $\mathcal{V}(A,\ \mathcal{M})$ is the set that $\mathcal{M}$ assigns to A. For each type expression of the form $\tau_1 \times \tau_2 \times \cdots \times \tau_n \rightarrow \sigma$, the semantic value $\mathcal{V}(\tau_1 \times \tau_2 \times \cdots \times \tau_n \rightarrow \sigma,\ \mathcal{M})$ is the set of all functions from the set $\mathcal{V}(\tau_1,\ \mathcal{M}) \times \mathcal{V}(\tau_2,\ \mathcal{M}) \times \cdots \times \mathcal{V}(\tau_n,\ \mathcal{M})$ to the set $\mathcal{V}(\sigma,\ \mathcal{M})$, i.e., the set of function f such that f takes n arguments, one from each of the sets $\mathcal{V}(\tau_1,\ \mathcal{M}), \mathcal{V}(\tau_2,\ \mathcal{M}), \cdots, \mathcal{V}(\tau_n,\ \mathcal{M})$, and returns an element of the set $\mathcal{V}(\sigma,\ \mathcal{M})$.

For example, if N is a type symbol then the type expression $N \times N \rightarrow N$ denotes the set of functions that take two elements of N and return an element of N. If $\mathcal{M}$ is a model such that $\mathcal{V}(N,\ \mathcal{M})$ is the set of natural numbers then $\mathcal{V}(N \times N \rightarrow N,\ \mathcal{M})$ is the set of functions that take two natural numbers and return a natural number. If $\mathcal{V}(N,\ \mathcal{M})$ is the set of natural numbers, then the addition function on natural numbers is a member of $\mathcal{V}(N \times N \rightarrow N,\ \mathcal{M})$. Arrow types denote function spaces — a function space is just a set of functions.

In addition to type expressions, the typed λ-calculus has terms. Each term of the typed λ-calculus is associated with a type expression called the type of that term. If f is a term of type τ then we write $f : \tau$ indicating that the semantic value of the term f must be a member of the semantic value of the type τ. For example, we would write $+ :\ N \times N \rightarrow N$ to indicate that the addition symbol denotes a function that maps two elements of N to an element of N. The terms of the typed λ-calculus can be defined as follows. Each term is associated with a unique type expression called the type of that term.

Definition: A term of the typed λ-calculus is one of the following.

- A variable or a constant symbol of any type (we assume infinitely many variables and constants of each type). A variable of type τ will be denoted as x^τ and a constant of type τ as c^τ.
- A λ-expression of the form $(\lambda\ (x_1^{\tau_1}, \cdots, x_n^{\tau_n})\ w)$ where $x_1^{\tau_1}$, $x_2^{\tau_n}$, $\cdots$, $x_n^{\tau_n}$ are distinct variables of type $\tau_1, \tau_2, \cdots, \tau_n$ respectively and w is any term. The type of this term is $\tau_1 \times \cdots \tau_n \rightarrow \sigma$ where σ is the type of the term w.
- An application of the form $(f\ s_1, \cdots s_n)$ where f and each s_i are terms and the type of f has the for $\tau_1 \times \cdots \tau_n \rightarrow \sigma$ where τ_i is the type of s_i. This application term has type σ.

For example, if $+$ is a constant of type $N \times N \rightarrow N$, and x and y are variables of type N, then $(+\ x\ y)$ is an application term of type N. The expression $(+\ x\ (+\ x\ y))$ is also a term of type N. However, the expression $(+\ x\ +)$ is not a term of the typed λ-calculus because the function symbol $+$ must take a term of type N as its second argument. We say that the expression $(+\ x\ +)$ is not well typed, or that it fails to type-check.

In addition to constants of various types, and applications built out of these constants, the typed λ-calculus includes λ-expressions. If x is a variable of type N, and $+$ is a constant of type $N \times N \rightarrow N$, then $(\lambda\ (x)\ (+\ x\ (+\ x\ x)))$ is a term of type $N \rightarrow N$.

A model of the λ-calculus provides an interpretation of each type symbol as a set plus an interpretation of each variable and constant as an element of its associated type expression.

Definition: A model $\mathcal{M}$ of the typed λ-calculus provides the following information.

- An interpretation of each type symbol as a set.

- An interpretation of each variable and constant symbol such that for each variable x^τ of type τ we have that $\mathcal{V}(x^\tau,\ \mathcal{M})$ is an element of $\mathcal{V}(\tau,\ \mathcal{M})$. A similar statement holds for constants.

Given a model we can now define the meaning of each term. For each term w of type τ we have that $\mathcal{V}(w,\ \mathcal{M})$ is an element of the set $\mathcal{V}(\tau,\ \mathcal{M})$. In the following definition let $\mathcal{M}$ be a model of the typed λ-calculus.

Definition:

- If $(f\ s_1\ \cdots\ s_n)$ is an application term then $\mathcal{V}((f\ s_1\ \cdots\ s_n),\ \mathcal{M})$ is the value of the function $\mathcal{V}(f,\ \mathcal{M})$ applied to the arguments $\mathcal{V}(s_1,\ \mathcal{M})$, $\cdots$, $\mathcal{V}(s_n,\ \mathcal{M})$.
- If $(\lambda\ (x_1^{\tau_1} \cdots x_n^{\tau_n})\ w)$ is a λ-term then $\mathcal{V}((\lambda\ (x_1^{\tau_1} \cdots x_n^{\tau_n})\ w),\ \mathcal{M})$ is the function that maps the elements $d_1, \cdots d_n$ of the sets $\mathcal{V}(\tau_1,\ \mathcal{M})$, $\cdots$, $\mathcal{V}(\tau_n,\ \mathcal{M})$ respectively to $\mathcal{V}(w,\ \mathcal{M}[x_1^{\tau_1} := d_1, \cdots, x_n^{\tau_n} := d_n])$ where $\mathcal{M}[x_1^{\tau_1} := d_1, \cdots, x_n^{\tau_n} := d_n]$ is the model identical to $\mathcal{M}$ except that it interprets each $x_i^{\tau_i}$ as the element d_i.

For example, if x is a variable of type N, and $+$ is a constant of type $N \times N \rightarrow N$, then $(\lambda\ (x)\ (+\ x\ (+\ x\ x)))$ denotes an element of the function space denoted by $N \rightarrow N$. If N is interpreted as the set of natural numbers, and $+$ is interpreted as addition on the natural numbers, then the above λ-term denotes the function that maps x to $3x$.

2 Equations and Inference Rules

Since the typed λ-calculus has terms, one can write equations and each equation is either true or false, in the standard way, in any given model. Equations are the only propositions of the simply typed λ-calculus and thus inference is necessarily a matter of deriving equations from equations. In most applications of the simply typed λ-calculus, the syntax and semantics of the language is more important than the inference rules for deriving equations. However, for the sake of completeness, a set of inference rules is presented here.

There are three standard inference rules for the λ-calculus. These rules are fundamental to the theory of computer programming languages and it is often useful to know these rules by name. The first rule, α-conversion, states that the meaning of a λ-term is independent of the choice of names for the bound variables. The rule of α-conversion requires a host of technical conditions. The the α-conversion rule, $w[x_1, \cdots, x_n]$ is a term that may or may not contain occurrences of the distinct variables $x_1, \cdots, x_n$ and $w[y_1, \cdots, y_n]$ is the result of simultaneously replacing $x_1, \cdots, x_n$ by the distinct variables $y_1, \cdots y_n$ where each y_i is a variable of the same type as x_i, no y_i appears free in $w[x_1, \cdots, x_n]$, and with appropriate renaming of bound variables inside w to avoid the capture of the introduced occurrences of the variables y_i.

α-conversion $\quad (\lambda\ (x_1\ \cdots\ x_n)\ w[x_1, \cdots x_n]) = (\lambda\ (y_1\ \cdots\ y_n)\ w[y_1, \cdots y_n])$

The second rule is called β-conversion. It allows a function application to be "reduced" in the case where the function being applied is a λ-term. In this rule, as in the α-conversion rule, $w[s_1, \cdots, s_n]$ denotes the result of simultaneously replacing all free occurrences of $x_1, \cdots, x_n$ in $w[x_1, \cdots, x_n]$ by the terms $s_1, \cdots, s_n$ respectively with appropriate renaming of bound variables in $w[x1, \cdots, x_n]$ to avoid the capture of free variables in $s_1, \cdots, s_n$.

β-conversion $\quad ((\lambda\ (x_1\ \cdots\ x_n)\ w[x_1, \cdots, x_n])\ s_1\ \cdots\ s_n) = w[s_1, \cdots, s_n]$

The third rule, η-conversion, expresses an "extensionality" condition on λ-terms. In this rule f is a λ-term such that none of the bound variables $x_1, \cdots, x_n$ appear free in f.

η-conversion $\quad (\lambda\ (x_1\ \cdots\ x_n)\ (f\ x_1\ \cdots\ x_n)) = f$

Of course the standard inference rules for equality are also sound. Recall that the substitution of equals for equals can only be performed in compositional expressions. In the typed λ-calculus applications are compositional (the meaning of an application is determined by the meaning of its subterms). However, λ-terms are not compositional and the substitution of equals for equals inside a λ-term is unsound. For example, the premise that $(+\ x\ x) = 2$ does not allow one to infer that $(\lambda\ (x)\ (+\ x\ x)) = (\lambda\ (x)\ 2)$. However, the following sequent rule of generalization can be used to derive equivalences between λ-terms. In the following rule, the variables $x_1, \cdots, x_n$ must not appear free in Σ.

Equality Generalization
$$\frac{\Sigma \vdash s = w}{\Sigma \vdash (\lambda\ (x_1\ \cdots\ x_n)\ s) = (\lambda\ (x_1\ \cdots\ x_n)\ w)}$$

The four inference rules, together with the traditional rules for equality are sound and complete for the types λ-calculus. Although the inference relation is undecidable, the set of valid equations (equations that are true in all models) is decidable. The decision procedure is based on the fact that the typed λ-calculus is *strongly normalizing*. This means that given any λ-term one can compute a unique normal form for that λ-term such that an equation is valid if and only if the two terms involved have the same normal form.

3 Higher Order Logic

The syntax and semantics of the typed λ-calculus can be used as the foundation for a wide variety of different logics. To transform the typed λ-calculus into a different logic one merely identifies "distinguished" or "logical" constants and specifies fixed meanings for those constants. For example, one might specify the type N and the symbol $+$ of type $N \times N \rightarrow N$ as distinguished constants. One can transform the typed λ-calculus into a different logic by restricting the models so that one only considers models in which N is interpreted as the set of natural numbers and in which $+$ is interpreted as addition. This restriction on models gives a new logic with new computational properties. For example, the equation

$$(\lambda\ (x\ y)\ (+\ x\ y)) = (\lambda\ (x\ y)\ (+\ y\ x))$$

is not valid (true in all models) in the typed λ-calculus, but it is valid in the different logic in which N and $+$ are required to have their standard meanings.

Higher order logic is derived from the typed λ-calculus by selecting distinguished symbols and restricting the set of models so that each distinguished symbol is guaranteed to have a certain standard meaning. First, we select a distinguished type symbol B (for *Boolean*) and require that in every model B denotes the two element set $\{\mathbf{T}, \mathbf{F}\}$. Given this distinguished type, any term of B will either denote $\mathbf{T}$ or $\mathbf{F}$ and hence any term of type B can be viewed as a proposition. After specifying that B denotes the set $\{\mathbf{T}, \mathbf{F}\}$, we select certain Boolean operators of type $B \to B$ and $B \times B \to B$ and require that these Boolean operations have their intended meaning. Finally, for each type τ we select a distinguished operator that allows for quantification over the type τ.

Definition: *Higher order logic* is that logic whose propositions are the terms of type B in the typed λ-calculus and whose models are those models of the typed λ calculus satisfying the following conditions.

- The type B denotes the set $\{\mathbf{T}, \mathbf{F}\}$.
- The Boolean operations $\neg$, $\vee$, $\wedge$, and $\to$, all have their standard meaning as operations on the type B.
- For each type τ the constant $=_\tau$, of type $\tau \times \tau \to B$, denotes the predicate such that, when applied to elements d_1 and d_2 of the type denoted by τ, returns $\mathbf{T}$ if and only if d_1 is the same element as d_2.
- For each type τ the constant $\forall_\tau$, of type $(\tau \to B) \to B$, denotes the operator such that $(\forall_\tau\ P)$ equals $\mathbf{T}$ if and only if the "predicate" P (which must be of type $\tau \to B$) is true on all elements of the type denoted by τ.

Higher order logic, as defined above, is just a restriction on the models of the typed λ-calculus so that certain distinguished symbols must always have their "intended meaning". Higher order logic is a generalization of first order logic. More precisely, any proposition of first order logic can be translated, in a meaning-preserving way, into a proposition of higher order logic. To define this translation let D be any type symbol other than the Boolean type B. In any model of higher order logic, as in any model of the typed λ-calculus, D denotes some set. We can think of the set denoted by D as being analogous to the semantic domain of a model of first order logic. One can show that any first order formula about the objects in D can be written as a formula of higher order logic. Higher order logic, like first order logic, has predicate symbols. In higher order logic, a monadic predicate on D is simply a constant symbol of type $D \rightarrow B$. A binary predicate on D is simply a constant of type $D \times D \rightarrow B$. In higher order logic, a proposition is simply a term of type B. If P is a binary predicate on type D, and c and d are constants of type D, then $(P\ c\ d)$ is a term of type B, i.e., a proposition of higher order logic. More generally, any atomic formula of first order logic can be translated into higher order logic simply by treating first order constants as constants of type D, first order functions as constants of type $D \times \cdots \times D \rightarrow D$, and first order predicates as constants of type $D \times \cdots \times D \rightarrow B$. Boolean combinations of atomic formulas can be constructed using the Boolean constants of type $B \rightarrow B$ and $B \times B \rightarrow B$. Let $\Phi(x)$ be the translation of some proposition of first order logic into a proposition of higher order logic where the variable x of type D appears free in $\Phi(x)$. We would like to show that there is a proposition of higher order logic that is equivalent to the first order proposition $\forall x \Phi(x)$. To do this we first construct the λ-term $(\lambda\ (x)\ \Phi(x))$ of type $D \rightarrow B$. We then use the constant $\forall_D$ of type $(D \rightarrow B) \rightarrow B$ to construct the proposition $(\forall_D\ (\lambda\ (x)\ \Phi(x)))$. This proposition denotes $\mathbf{T}$ if and only if the first order proposition $\forall x \Phi(x)$ denotes $\mathbf{T}$. Since existential quantification can be expressed in terms of universal quantification and negation, we have now given a way of translating any first order proposition into higher order logic. For any type τ we can write $\forall_\tau x\ \Phi(x)$ as an abbreviation for $(\forall_\tau\ (\lambda\ (x)\ \Phi(x)))$ and $\exists_\tau x\ \Phi(x)$ as an abbreviation for $\neg\forall_\tau x\ \neg\Phi(x)$.

4 The Expressive Power of Higher Order Logic

The translation of first order propositions into higher order propositions shows that higher order logic can express any first order proposition. The converse does not hold — there are propositions of higher order logic that can not be expressed in first order logic. For example, higher order logic, unlike first order logic, can express the notion of transitive closure. To see this, let R and T be two relation symbols, i.e., two constants of type $D \times D \rightarrow B$. We want to write a proposition of higher order logic that is true if and only if T is the transitive closure of R.

To state that T is the transitive closure of R we first construct a λ-predicate that takes a relation and returns **T** if and only if that relation is transitive. I will use the symbol **TRANS** as an abbreviation for this λ-predicate. In this predicate the argument S has type $(D \times D \rightarrow B) \rightarrow B$.

$$\mathbf{TRANS} \equiv (\lambda\ (S)\ \forall_D x \forall_D y \forall_D z\ (S\ x\ y)\ \wedge\ (S\ y\ z) \rightarrow (S\ x\ z))$$

Now we construct a λ-predicate that expresses the notion that one relation is a "subrelation" of another. In the following predicate S and W are each relations, i.e., have type $D \times D \rightarrow B$. I will call this λ-predicate **SUBREL**.

$$\mathbf{SUBREL} \equiv (\lambda\ (S\ W)\ \forall_D x \forall_D y\ (S\ x\ y) \rightarrow (W\ x\ y))$$

Given these two higher order λ-predicates we can now state that T is the transitive closure of R with the following set of propositions.

- $(\mathbf{SUBREL}\ R\ T)$
- $(\mathbf{TRANS}\ T)$
- $\forall_{D \times D \rightarrow B} S\ \ [(\mathbf{SUBREL}\ R\ S) \wedge (\mathbf{TRANS}\ S) \rightarrow (\mathbf{SUBREL}\ T\ S)$

The above propositions state that T contains R, T is transitive, and T is the least relation satisfying these two properties. The first two propositions — that T contains R and is transitive, can both be stated in first order logic. The third proposition necessarily involves quantification over relations and can not be stated in first order logic.

5 Problems

A future version of the notes will contain problems on higher order combinators.