Crash Course: Type theory and λ-abstraction

Ted Sider
Higher-order metaphysics

1. Third-order logic and beyond

With predicates of predicates we can symbolize sentences like:

> Sally and John have exactly the same virtues
>
> $\forall X(\boldsymbol{V}X \rightarrow (Xs \leftrightarrow Xj))$

Syntax:

> In addition to the ordinary kind of predicates (both constant and variable), there are *higher-order predicates*: $\boldsymbol{F}, \boldsymbol{G}, \ldots$. For any higher-order predicate, $\boldsymbol{F}$, and any one-place ordinary predicate G, "$\boldsymbol{F}G$" is a formula.

Semantics:

> In any interpretation, the denotation of a higher-order predicate is a set of sets of members of the domain.
>
> The formula "$\boldsymbol{F}G$" is true in an interpretation if and only if the denotation of the ordinary predicate G is a member of the denotation of the higher-order predicate $\boldsymbol{F}$.

Also variables in predicates-of-predicates syntactic position:

> There is some type of property such that Sally and John have exactly the same properties of that type
>
> $\exists \boldsymbol{Y}\forall X(\boldsymbol{Y}X \rightarrow (Xs \leftrightarrow Xj))$

We could keep going, introducing predicates of predicates-of-predicates, etc.

2. Type theory

In type theory we introduce a very general and iterative notion of syntactic category, and have expressions (including variables) of all syntactic categories.

2.1 More about syntax

Syntactic rules specify what kinds of expressions combine with what other kinds of expressions to form still other kinds of expressions; e.g.:

Rule for one-place predicates: If you take a one-place predicate, F, and attach it to a term, t, the result Ft is a formula

Rule for $\sim$*:* if you take the expression $\sim$, and attach it to a formula, A, the result $\sim A$ is a formula

Common pattern:

> If you take an expression of category X, and attach it to an expression of category Y, the result is an expression of category Z

To illustrate how the pattern might continue, consider this rule for a new syntactic category we could introduce, that of *predicate functors* (adverbs):

> If you take a predicate functor $\boldsymbol{q}$, and attach it to a one-place predicate, F, the result $\boldsymbol{q}F$ is a one-place predicate

2.2 Types

We need to talk about syntactic categories as entities—which we call *types*—so that we can make generalizations about all syntactic categories:

There are two undefined types, e and t

> (e is the type of singular terms; t is the type of sentences)

For any types, $a_1, \ldots, a_n$ and b, there is another type $\langle a_1, \ldots, a_n, b\rangle$

> ($\langle a_1, \ldots, a_n, b\rangle$ is the type of expressions that combine with expressions of types $a_1, \ldots, a_n$ to form an expression of type b)

Example 1: $\langle e, t\rangle$. Expressions of this type combine with something of type e to form an expression of type t. That is: they combine with terms to form sentences. One-place predicates (of individuals)!

Example 2: $\langle e, e, t\rangle$. Combine with *two* expressions of type e to form an expression of type t. Two-place predicates (of individuals).

Example 3: $\langle t, t\rangle$. Combine with a formula to make a formula. One-place sentence operators (like $\sim$ and $\Box$).

Example 4: $\langle t, t, t\rangle$. Combine with two formulas to make a formula. Two-place sentence operators ($\wedge, \vee, \ldots$).

Example 5: $\langle\langle e, t\rangle, t\rangle$. Combine with one-place predicates to make formulas. Higher-order predicates.

Example 6: $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$. Combine with one-place predicates to form one-place predicates. Adverbs/predicate functors.

2.3 Meaning: Frege and functions

Idea: think of the meaning of an expression in terms of what it *does*—in terms of how it helps generate the meanings of complex expressions.

The denotation of a proper name is an individual.

The denotation of a sentence is its truth value (T or F)

The denotation of a predicate is a function from individuals to truth values

The denotation of a sentential connective is a function from truth values to truth values

Denotations of 'runs' and 'bites':

$$r(x) = \begin{cases} T \text{ if } x \text{ runs} \\ F \text{ if } x \text{ does not run} \end{cases} \qquad b(x,y) = \begin{cases} T \text{ if } x \text{ bites } y \\ F \text{ if } x \text{ does not bite } y \end{cases}$$

Denotations of 'and' ($\wedge$) and 'or' ($\vee$):

$$\begin{array}{ll} c(T,T) = T & d(T,T) = T \\ c(T,F) = F & d(T,F) = T \\ c(F,T) = F & d(F,T) = T \\ c(F,F) = F & d(F,F) = F \end{array}$$

2.4 Formal semantics for type theory

First specify what sorts of entities in an interpretation are denoted by expressions of each type:

Kinds of denotations, for a given domain D

e denotations are members of D

t denotations are truth values (i.e., T, F)

An $\langle a_1, \ldots, a_n, b\rangle$ denotation is an n-place function that maps an a_1 denotation, an a_2 denotation, …, and an a_n denotation, to a b denotation

Example: an $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$ denotation (the kind of denotation for a predicate functor) is a function that maps $\langle e,t\rangle$ denotations to $\langle e,t\rangle$ denotations—a function that maps any function from D to truth values to another function that maps D to truth values.

Definition of interpretation for type theory

An interpretation consists of a domain, D, and a specification, for each nonlogical expression of type a, of an a denotation for D

Denotations of complex expressions (in a given interpretation)

If $E_1, \ldots, E_n$ are expressions of types $a_1, \ldots, a_n$, with denotations $d_1, \ldots, d_n$, and expression E has type $\langle a_1, \ldots, a_n, b\rangle$ and denotes a function f, then the denotation of the complex expression $E(E_1, \ldots, E_n)$ (which has type b) is: $f(d_1, \ldots, d_n)$

Example:

$$\boldsymbol{\sim q}(R)(c) \qquad \text{(“Ted doesn’t run quickly”)}$$

Denotations:

	type	kind of denotation	particular denotation
c	e	member of the domain	Ted
R	$\langle e,t\rangle$	fx from entities to truth values	the fx r that maps o to T iff o runs
$\boldsymbol{q}$	$\langle\langle e,t\rangle,\langle e,t\rangle\rangle$	fx from $\langle e,t\rangle$ fxs to $\langle e,t\rangle$ fxs	the fx q that maps g to the “g-ing quickly” fx
$\boldsymbol{\sim}$	$\langle t,t\rangle$	fx from TVs to TVs	the fx n that maps T to F and F to T

Let “$|E|$” abbreviate “the denotation of expression E”. Let’s calculate $|\boldsymbol{\sim q}(R)(c)|$:

$$|R| = r \qquad \text{(see table)}$$
$$|\boldsymbol{q}| = q \qquad \text{(see table)}$$
$$|\boldsymbol{q}(R)| = q(r) \qquad \text{(rule for denotations of complexes)}$$
$$|c| = \text{Ted} \qquad \text{(see table)}$$
$$|\boldsymbol{q}(R)(c)| = q(r)(\text{Ted}) \qquad \text{(rule for denotations of complexes)}$$
$$|\boldsymbol{\sim}| = n \qquad \text{(see table)}$$
$$|\boldsymbol{\sim q}(R)(c)| = n(q(r)(\text{Ted})) \qquad \text{(rule for denotations of complexes)}$$

Calculating $n(q(r)(\text{Ted}))$: $q(r)$ maps a member of the domain to T iff it runs quickly (since q maps a function from the domain to truth values to the corresponding “quickly function” from members of the domain to truth values.) Since I run quickly, $q(r)(\text{Ted}) = T$. And so, $n(q(r)(\text{Ted})) = F$.

Note: this is a formal semantics, and needn’t be a perfect model of intended meaning.

2.5 Variables

In type theory, we allow variables of each type. E.g., quantification into the position of $\wedge$ (‘and’):

$$\exists \bigcirc (Sc \bigcirc Ec)$$

into the positions of formulas, predicate functors, etc.

2.6 Typing symbols

Officially we write all variables as “x”, and all constants as “c”, and indicate types by superscripts. So for any type, a, x^a (and $x^a_1, x^a_2, \ldots$) is a variable of type a, and c^a (and $c^a_1, c^a_2, \ldots$) is a constant of type a. But unofficially:

official	unofficial	
x^e	$x, y, \ldots$	individual variables
c^e	$c, d, \ldots$	names
x^t	$P, Q \ldots$	sentence variables
$x^{\langle e,t\rangle}, x^{\langle e,e,t\rangle}$	$X, F, R \ldots$	predicate variables
$x^{\langle t,t,t\rangle}, x^{\langle t,t\rangle}$	$\triangle, \bigcirc$	sentence-operator variables
$c^{\langle\langle e,t\rangle,t\rangle}$	$\boldsymbol{F}, \boldsymbol{G}, \ldots$	higher-order predicate constants
$x^{\langle\langle e,t\rangle,t\rangle}$	$\boldsymbol{X}, \boldsymbol{Y}, \ldots$	higher-order predicate variables
$c^{\langle\langle e,t\rangle,\langle e,t\rangle\rangle}$	$\boldsymbol{q}, \boldsymbol{r}, \ldots$	predicate-functor constants
$x^{\langle\langle e,t\rangle,\langle e,t\rangle\rangle}$	$\boldsymbol{x}, \boldsymbol{y}, \ldots$	predicate-functor variables

3. λ-abstraction

3.1 Complex predicates

Natural languages have complex predicates, like ‘is sitting and eating’ (as in ‘Ted is sitting and eating’). We use the symbol λ to formalize them:

$$\lambda x(Sx \wedge Ex)$$

glossed “is an x such that Sx and Ex”, or “is such that: it is sitting and it is eating”, or “is sitting and eating”. Also multi-place complex predicates, e.g., “bites or is bitten by”:

$$\lambda xy(Bxy \vee Byx)$$

So far, the syntax of λ is:

Where $x_1, \ldots, x_n$ are any variables and A is any formula, $\lambda x_1 \ldots x_n A$ is an n-place predicate

λ abstracts are used in sentences just like other predicates:

$$\lambda x(Sx \wedge Ex)c$$

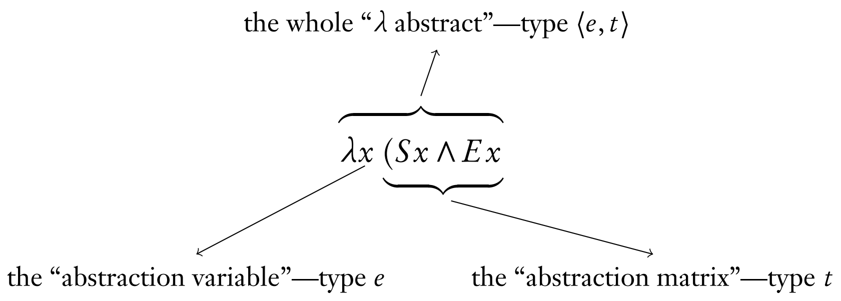
Formal semantics: $\lambda v A$ denotes the function mapping any o in the domain to T iff A is true of o.

Example: let S mean “sitting” and E mean “eating”. Then $\lambda x(Sx \wedge Ex)$ denotes the function mapping any o to T iff $Sx \wedge Ex$ is true of o—i.e., iff o is sitting and o is eating. So if c names Ted, then $\lambda x(Sx \wedge Ex)c$ is true iff Ted is sitting and Ted is eating—iff $Sc \wedge Ec$ is true.

Example: let B mean “bites”. Then $\lambda x \exists y Byx$ denotes the function mapping o to T iff $\exists y Byx$ is true of o—iff something bites o. So $\lambda x \exists y Byx(c)$ is true iff something bites Ted—iff $\exists y Byc$ is true.

So: $\lambda x(Sx \wedge Ex)c$ is equivalent to $Sc \wedge Ec$; $\lambda x \exists y Byx(c)$ is equivalent to $\exists y Byc$.

3.2 Generalizing λ: syntax


the whole “λ abstract”—type ⟨e, t⟩
λx (Sx ∧ Ex
the “abstraction variable”—type e
the “abstraction matrix”—type t

In type theory, we let the abstraction variables and matrix be of any types:

Syntax for λ generalized

For any variables, $x_1^{a_1}, \dots, x_n^{a_n}$, of types $a_1, \dots, a_n$ (the “abstraction variables), and any expression, E, of type b (the “abstraction matrix”), the expression

$$\lambda x_1^{a_1} \dots x_n^{a_n} E$$

(the λ abstract) is of type $\langle a_1, \dots, a_n, b \rangle$

3.3 Generalizing λ: semantics

We can gloss λ abstracts whose matrix expression is a *formula* using "such that":

> λxA: "is an x such that A"
>
> λXA: "is a property, X, such that A"
>
> λPA: "is a proposition (or truth value), P, such that A"
>
> etc.

Also we can think about how they combine with other expressions. Recall, e.g., the equivalence between $\lambda x(Sx \wedge Ex)c$ and $Sc \wedge Ec$. In general, λ abstracts are equivalent to their "β reductions":[1]

β conversion

The result of applying the λ abstract $\lambda x_1^{a_1} \ldots x_n^{a_n} E$ to expressions $A_1, \ldots, A_n$ (of types $a_1, \ldots, a_n$), namely:

$$\lambda x_1^{a_1} \ldots x_n^{a_n} E(A_1, \ldots, A_n)$$

reduces by β conversion to:

$$E_{x_1^{a_1} \mapsto A_1, \ldots, x_n^{a_n} \mapsto A_n}$$

Example 0*:*

$$\lambda X(Xc \vee Xd)$$

("being a property that is had either by c or by d"). If you attach it to a one-place predicate F, you get:

$$\lambda X(Xc \vee Xd)F$$

which means the same thing as (via β conversion):

$$Fc \vee Fd$$

Thus what $\lambda X(Xc \vee Xd)$ does is this: it converts any predicate, F, into a sentence meaning that c is F or d is F.

[1]No free variables in $A_1, \ldots A_n$ may be "captured" by quantifiers in $E_{x_1^{a_1} \mapsto A_1, \ldots, x_n^{a_n} \mapsto A_n}$.

Example 1:

$$\lambda P\, P$$

P is a variable of type t; so the abstraction variable and matrix are type t, so the entire λ abstract is type $\langle t,t\rangle$. If you attach it to a formula, A, you get this formula:

$$\lambda P\, P(A)$$

which reduces by β conversion to:

$$A$$

So: $\lambda P\, P$ attaches to a sentence A to form a sentence that means A. It's a redundant sentence operator.

Example 2:

$$\lambda x\left(\boldsymbol{q}(R)(x)\wedge \boldsymbol{g}(R)(x)\right)$$

where x is a variable of type e, $\boldsymbol{q}$ and $\boldsymbol{g}$ are predicate functor constants (type $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$), and R is a one-place predicate constant (type $\langle e,t\rangle$). The entire λ abstract is type $\langle e,t\rangle$.

Think of $\boldsymbol{q}$ as meaning "quickly", $\boldsymbol{g}$ as meaning "gracefully", and R as meaning "runs". Then $\boldsymbol{q}(R)$ means "runs quickly", and $\boldsymbol{g}(R)$ means "runs gracefully"; and so, the entire λ abstract means "is an x such that x runs quickly and x runs gracefully"—i.e., "runs quickly and gracefully".

We can reach the same conclusion by thinking about what the λ abstract does. If you attach it to a name, c, symbolizing "Ted", say, you get:

$$\lambda x\left(\boldsymbol{q}(R)(x)\wedge \boldsymbol{g}(R)(x)\right)(c)$$

which reduces by β conversion to:

$$\boldsymbol{q}(R)(c)\wedge \boldsymbol{g}(R)(c)$$

which means that Ted runs quickly and runs gracefully.

Example 3:

$$\lambda Y\, \lambda x\left(\boldsymbol{q}(Y)(x)\wedge \boldsymbol{g}(Y)(x)\right)$$

The matrix variable Y is of type $\langle e,t\rangle$, and the matrix expression is itself a λ abstract:

$$\lambda x\left(\boldsymbol{q}(Y)(x)\wedge \boldsymbol{g}(Y)(x)\right)$$

This "inner" λ abstract is of type $\langle e, t\rangle$. Thus the "outer" λ abstract is type $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$. It's a predicate functor.

If you attach the outer λ abstract to a one-place predicate, R, you get:

$$\lambda Y\, \lambda x \left(\boldsymbol{q}(Y)(x) \wedge \boldsymbol{g}(Y)(x)\right)(R)$$

which β-reduces to:

$$\lambda x \left(\boldsymbol{q}(R)(x) \wedge \boldsymbol{g}(R)(x)\right)$$

—a predicate meaning "runs quickly and gracefully". So the outer λ abstract converts "runs" into "runs quickly and gracefully". It's a complex adverb: "quickly and gracefully"!

We can also think of the meanings of λ abstracts by extending our formal semantics to them:

Semantics for λ abstracts

In any interpretation, the λ abstract

$$\lambda x_1^{a_1} \ldots x_n^{a_n}\, E$$

denotes the n-place function that maps any n denotations, of types $a_1, \ldots, a_n$, respectively, to the object that E denotes when the variables $x_1^{a_1}, \ldots, x_n^{a_n}$ are assigned those denotations

4. Alternate systems

4.1 Binary-only types and schönfinkelization

In some systems, complex types must always be "binary": $\langle a, b\rangle$. But more complex types can still be simulated.

Example: instead of L ("loves") being a two-place predicate (type $\langle e, e, t\rangle$), it has type $\langle e, \langle e, t\rangle\rangle$. Combining L with a name c yields a predicate $L(c)$ meaning *is loved by* c. So $L(c)(d)$ is a sentence meaning that d is such that it is loved by c.

Example: instead of $\wedge$ being a two-place sentence operator (type $\langle t, t, t\rangle$), it has type $\langle t, \langle t, t\rangle\rangle$: it attaches to a formula to make a one-place sentence operator. So instead of writing $A \wedge B$ we write $\wedge(A)(B)$.

4.2 Relational type theory

The theory we have been developing is called "functional type theory". In "relational type theory", all types (other than names and formulas) are predicate types (of arbitrarily high order and complexity):

Definition of relational types

Undefined type: e

(The type of singular terms)

For any types $a_1, \ldots a_n$, $(a_1, \ldots, a_n)$ is also a type

(The type of expressions that combine with n expressions, of types $a_1, \ldots, a_n$, respectively, to make a *formula*. The case where $n = 0$ is allowed: () is a type—the type of formulas)

Note: the functional type $\langle a, b\rangle$ is the type of one-place expressions which convert an a into a b, whereas the relational type (a, b) is the type of two-place expressions which convert an a and a b into a *formula*.

For instance, the functional type $\langle e, e\rangle$ is the type of *one-place function symbols* (like the successor sign $'$ from the language of arithmetic), whereas the relational type (e, e) is the type of two place (first-order) predicates, such as B for "bites".

Any relational type has an "equivalent" functional type. (For example, to (e) there corresponds $\langle e, t\rangle$; to $((), e)$ there corresponds $\langle t, e, t\rangle$.) But no relational types correspond (in this sense) to, e.g., $\langle e, e\rangle$ or $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$. (Though one can simulate expressions of these types using Russell's theory of descriptions.)

4.3 Quantifiers as higher-order predicates

Why not think of a quantified natural language sentence like:

Something is sitting and eating

as meaning:

Sitting-and-eating has at least one instance

has-at-least-one-instance$\big(\lambda x(Sx \wedge Ex)\big)$

$$\exists \lambda x(Sx \wedge Ex)$$

Thus we could have λ do all the variable binding, and have quantifiers attach to λ abstracts. $\exists$ means "has at least one instance"; $\forall$ means "everything is an instance". New syntax for the quantifiers:

If Π is a one-place predicate then $\forall\Pi$ and $\exists\Pi$ are formulas

Example: instead of writing $\exists x \forall y Rxy$ for "there is someone who respects everyone", we instead write:

$$\exists \lambda x \forall \lambda y Rxy$$

We can play this trick in type theory too. Instead of $\exists X X c$ we write $\exists \lambda X X c$; instead of $\forall P(P \vee {\sim}P)$ we write $\forall \lambda P(P \vee {\sim}P)$. In general, for any type a, instead of writing:

$$\exists x^a A \qquad\qquad \forall x^a A$$

(where A is a formula) we now write instead:

$$\exists^a \lambda x^a A \qquad\qquad \forall^a \lambda x^a A$$

$\exists^a$ and $\forall^a$ are of type $\langle\langle a, t\rangle, t\rangle$. They mean "applies to at least one a-entity" and "applies to every a-entity", respectively. In the formal semantics, they denote:

$\exists^a$ denotes the function that maps any $\langle a, t\rangle$ denotation, d, to T if and only if for some a denotation, d', $d(d') = T$

$\forall^a$ denotes the function that maps any $\langle a, t\rangle$ denotation, d, to T if and only if for every a denotation, d', $d(d') = T$