EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Faculty of Philosophy
General Linguistics

Semantics & Pragmatics SoSe 2023

Lecture 8: Formal Semantics V
(Lambda Calculus)

23/05/2023, Christian Bentz

Overview

Section 1: Recap of Lecture 7

Example

All animals that live in the jungle have a color.

Propositional logic:
p

First-order predicate logic:
$\forall x((Ax \wedge Jx) \rightarrow Cx)$
Translation key: Ax: x is an animal; Jx: x lives in the jungle; Cx: x has a color.

Second-order predicate logic:
$\forall x(\exists X((\mathcal{A}X \wedge Xx) \wedge Jx) \rightarrow \exists Y(Yx \wedge \mathcal{C}Y))$
Translation key: $\mathcal{A}X$: x is a property (type of animal) which has the property of being an animal; Jx: x lives in the jungle; $\mathcal{C}X$: X is a property (a particular color) which has the property of being a color.

The Theory of Types

How can we represent this **potentially infinite number of expressions** while conserving their *internal structure* and *combinatorial relationships*? – A logical system developed to fit this requirement is the so-called **theory of types** which was developed by Bertrand Russell as a remedy for paradoxes encountered in set theory.

Gamut (1991), Volume 2, p. 78.

Definition: The Syntax of Types

For the set of types $\mathbb{T}$ we define that:

(i) $e, t \in \mathbb{T}$,

(ii) if $a, b \in \mathbb{T}$, then $\langle a, b \rangle \in \mathbb{T}$,

(iii) nothing is an element of $\mathbb{T}$ except on the basis of clauses (i) and (ii).

Gamut (1991), Volume 2, p. 79.

Note: *a* and *b* above are variables which stand in for all kinds of types. This means we can create an infinite number of types by recursively applying clause (ii). For example:

Applying (ii) to $a = e$ and $b = t$ yields $\langle e, t \rangle$
Applying (ii) to $a = \langle e, t \rangle$ and $b = t$ yields $\langle \langle e, t \rangle, t \rangle$
Applying (ii) to $a = e$ and $b = \langle \langle e, t \rangle, t \rangle$ yields $\langle e, \langle \langle e, t \rangle, t \rangle \rangle$
etc.

Examples of Valid and Invalid Types

e ✓
t ✓
$\langle e, t \rangle$ ✓
$\langle t, e \rangle$ ✓
$\langle t, \langle t, e \rangle \rangle$ ✓
$\langle \langle t, \langle t, e \rangle \rangle, t \rangle$ ✓

et x
e, t x
$\langle e, e, t \rangle$ x
$\langle e, \langle e, t \rangle$ x

Note: The usage of left and right ankled brackets as defined by clause (ii) results in a **strict binarization** of the internal structure of types, i.e. at each level of embedding we always have an **ordered pair** of more basic types.

Definition: Functional Application

How do we derive one type of expression from another?

"[...] if α is an expression of type $\langle a, b \rangle$ and β is an expression of type a, then $\alpha(\beta)$ is of type b."

Gamut (1991), Volume 2, p. 79.

Examples

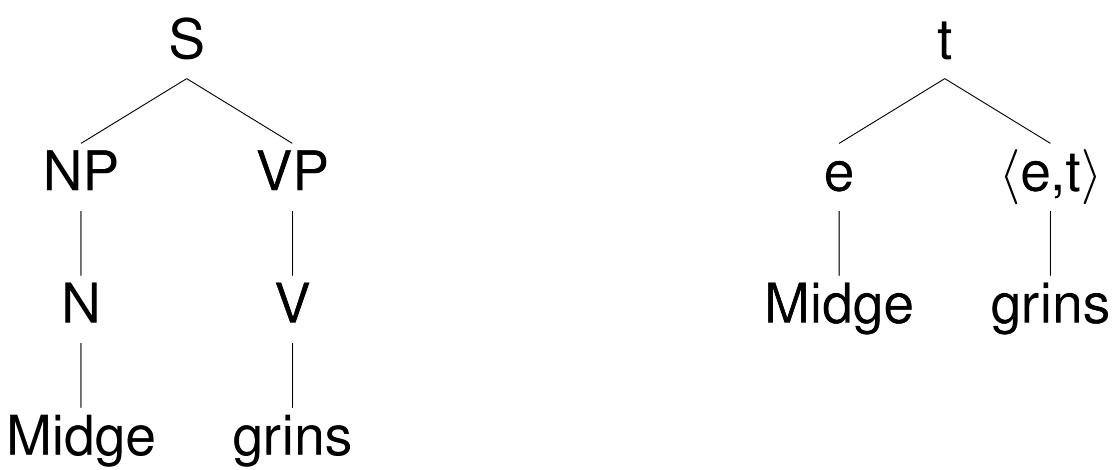
If $\alpha = \langle e, t \rangle$ and $\beta = e$ then $\alpha(\beta) = t$.
If $\alpha = \langle \langle e, t \rangle, \langle e, t \rangle \rangle$ and $\beta = \langle e, t \rangle$ then $\alpha(\beta) = \langle e, t \rangle$.
If $\alpha = \langle t, \langle t, e \rangle \rangle$ and $\beta = t$ then $\alpha(\beta) = \langle t, e \rangle$.

However,

If $\alpha = \langle t, \langle t, e \rangle \rangle$ and $\beta = \langle t, e \rangle$ then $\alpha(\beta)$ is **not defined**.

Semantic Types: One-Place Predicates

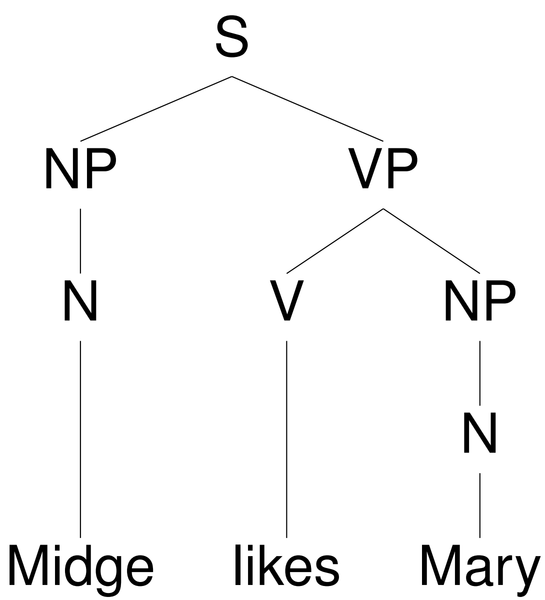
An **intransitive verb** requires **one argument** to be filled in order to form a full sentence, hence it is of the **type** $\langle \mathbf{e,t} \rangle$. Remember that the argument is on the left side of the tuple (ordered pair), hence the component of type *entity* (e) is left.

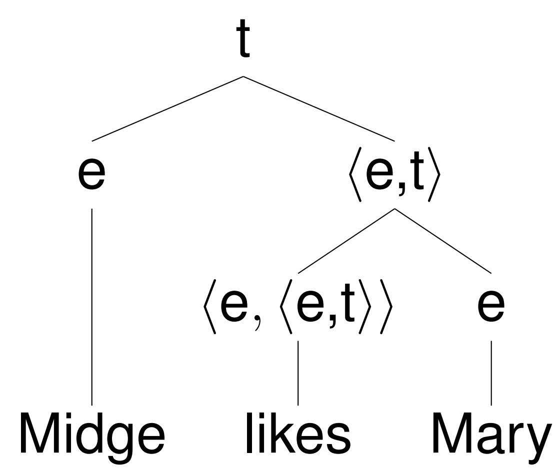


Section 1: Recap of Lecture 7
Section 2: Lambda Calculus in Logical Languages
Section 3: λ-Abstraction
Section 4: λ-Conversion
Section 5: Modelling Compositionality with λ-Calculus
Section 6: The Semantics of λ-Calculus
Summary
References

Semantic Types: Two-Place Predicates

A **transitive verb** requires **two arguments** to be filled in order to form a full sentence, hence it is of the **type** $\langle$**e,**$\langle$**e,t**$\rangle\rangle$.





Summary: Types of Expressions

There is a long (potentially infinite) list of **types of expressions** which we might want to represent in our logical language in order to capture the different combinatorial possibilities we find in natural languages.

Type	**Kind of expression**	**Examples**
e	Individual expression	*John, Jumbo*
$\langle e, t\rangle$	One-place first-order predicate	*walks, red, loves Mary*
$\langle e, \langle e, t\rangle\rangle$	Two-place first-order predicate	*loves, sees*
$\langle e, \langle e, \langle e, t\rangle\rangle\rangle$	Three-place first-order predicate	*lies between (and)*
t	Sentence	*John walks, John loves Mary*
$\langle t, t\rangle$	Sentential modifier	*Not*
$\langle e, e\rangle$	Function (entitiy to entity)	*The father of*
$\langle\langle e, t\rangle, \langle e, t\rangle\rangle$	Predicate modifier	*Quickly, beautifully, fast*
$\langle\langle e, t\rangle, t\rangle$	One-place second-order predicate	*Is a color*
$\langle\langle e, t\rangle, \langle\langle e, t\rangle, t\rangle\rangle$	Two-place second-order predicate	*Is a brighter color than*
etc.	etc.	etc.

The Syntax: Recursive Definition

The clauses for the syntax of a type-theoretic language are then:

(i) If α is a variable or a constant of type a in L [i.e. v_a or c_a], then α is an expression of type a in L.

(ii) If α is an expression of type $\langle a, b \rangle$ in L, and β is an expression of type a in L, then $(\alpha(\beta))$ is an expression of type b in L.

(iii) If ϕ and ψ are expressions of type t in L (i.e. formulas in L), then so are $\neg\phi$, $(\phi \land \psi)$, $(\phi \lor \psi)$, $(\phi \rightarrow \psi)$, and $(\phi \leftrightarrow \psi)$.

(iv) If ϕ is an expression of type t in L and v is a variable (of arbitrary type a), then $\forall v\phi$ and $\exists v\phi$ are expression of type t in L.

(v) If α and β are expressions in L which belong to the same (arbitrary) type, then $(\alpha = \beta)$ is an expression of type t in L.

(vi) Every expression L is to be constructed by means of (i)-(v) in a finite number of steps.

Gamut (1991), Volume 2, p. 81-82.

Examples of Valid and Invalid Expressions

Definition of Types

Assume j is of type e (i.e. representing an entity), x is of type e, A is of type $\langle e, t\rangle$ (i.e. a first order one-place predicate), B is of type $\langle e, \langle e, t\rangle\rangle$ (i.e. a first-order two-place predicate), and $\mathcal{C}$ is of type $\langle\langle e, t\rangle, t\rangle$ (i.e. a second-order one-place predicate).

Expressions	Clause Applied
j ✓	(i)
A ✓	(i)
A(j) ✓	(i) and (ii)
(B(j))(x) ✓ alternative notation: B(j)(x)	(i) and (ii)
$\mathcal{C}$(B(j)) ✓	(i) and (ii)
A(j) $\wedge$ $\mathcal{C}$(A)✓	(i), (ii), and (iii)
$\forall$xA(x)✓	(i), (ii), and (iv)
Aj x	–
B(A) x	–
$\forall$x$\mathcal{C}$(x) x	–

Truth Valuation

However, the complexity of defining interpretation functions for all kinds of different types of expressions (see table below from Gamut) is beyond the scope of this course.

Gamut (1991), p. 86.

Type	**Interpretation**
e	Entity
$\langle e, t\rangle$	Function from entities to truth values, i.e. characteristic function
$\langle e, \langle e, t\rangle\rangle$	Function from entities to sets of entities
$\langle e, \langle e, \langle e, t\rangle\rangle\rangle$	Function from entities to functions from entities to sets of entities
t	Truth value
$\langle t, t\rangle$	Function from truth values to truth values
$\langle e, e\rangle$	Function from entities to entities
$\langle\langle e, t\rangle, \langle e, t\rangle\rangle$	Function from sets of entities to sets of entities
$\langle\langle e, t\rangle, t\rangle$	Characteristic function of a set of sets of entities
$\langle\langle e, t\rangle, \langle\langle e, t\rangle, t\rangle\rangle$	Function from sets of entities to sets of sets of entities
etc.	etc.

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Faculty of Philosophy
General Linguistics

Section 2: Lambda Calculus in Logical Languages

Towards a Fully Compositional Account

A logical language *L* built on the theory of types is extremely powerful, since it is capable of representing a potentially infinite number of different natural language structures. However, we still **cannot (yet) represent parts of sentences** or predicates in a **fully compositional account**.

Examples	**Translations (Typed Language)**
John smokes.	$S(j)$
John smokes and drinks.	$S(j) \wedge D(j)$
Every man walks.	$\forall x(M(x) \rightarrow W(x))$
smokes and drinks	?
every man	?
every	?

Some (Possible?) Translations and Problems

Example	Translation (?)	Problems
smokes and drinks	S ∧ D	see (1)
smokes and drinks	S(x) ∧ D(x)	see (2)
every man	∀xM(x)	see (3)
every	∀, ∀x	see (4)

(1) This is not a valid expression in a type-theoretic logical language, since only formulas of type *t* can be combined with connectives, while first-order predicates lacking arguments are of type $\langle e, t \rangle$. This translation would not be valid in standard predicate logic either, since atomic sentences are here defined as predicates taking at least one constant or variable.

(2) While this is a valid expression, it would translate as *x smokes and x drinks*, which is not exactly the same as just *smokes and drinks*.

(3) Given a domain of entities *D*, ∀xM(x) backtranslates to *everybody is a man*, which is not the same as *every man*.

(4) Both of these are neither valid expressions in a type-theoretic language, nor valid formulas in standard predicate logic.

Historical Note

"[...] it was not until the late 1960s that logical techniques were applied to a **compositional analysis of natural language**, an achievement ascribable mostly to **Richard Montague**. As it turns out, predicate logic is not well suited for this task, one reason being that its expressive power resides exclusively in the **sentence-like category of formulae**: only formulae are recursively combinable, all other expressions are lexical. As long as only the meanings of full sentences are at stake, this does not matter; but when it comes to representing their parts, the **resources of predicate logic** do not suffice."

Zimmerman & Sternefeld (2013), p. 253.

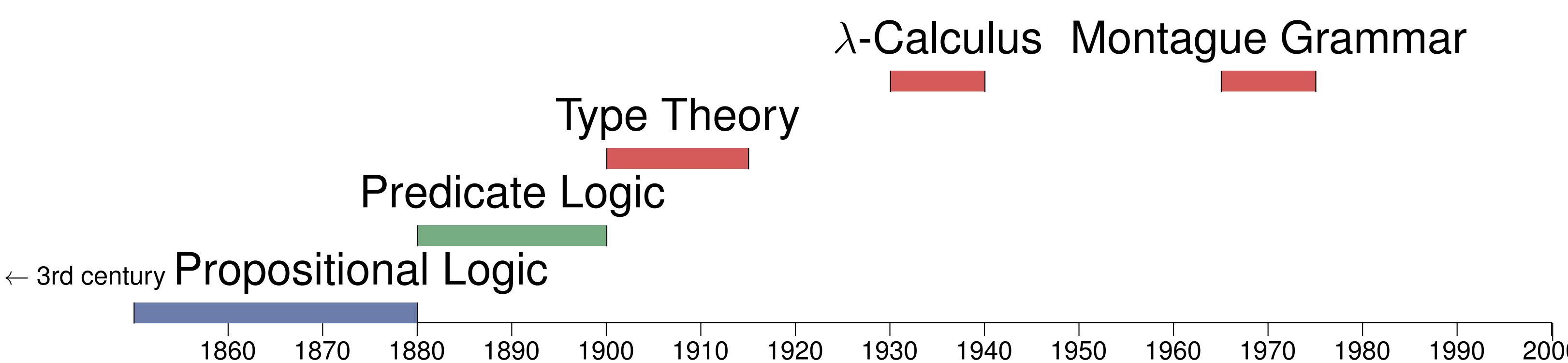
Remember Standard (First Order) Predicate Logic:

(i) If A is an n-ary predicate letter in the vocabulary of L, and each of $t_1, \ldots, t_n$ is a constant or a variable in the vocabulary of L, then $At_1, \ldots, t_n$ is a **formula** in L.

Historical Note: Lambda Calculus

"**Functional abstraction** was already introduced by Frege (1891), who expressed it by accented Greek letters. Its importance for compositional semantic analysis was first realized by Richard Montague, who used **lambdas** – a notation that goes back to Alonzo Church's (1903–1995) work on the **theory of computation** [...]."

Zimmerman & Sternefeld (2013), p. 254.



A FORMULATION OF THE SIMPLE THEORY OF TYPES

ALONZO CHURCH

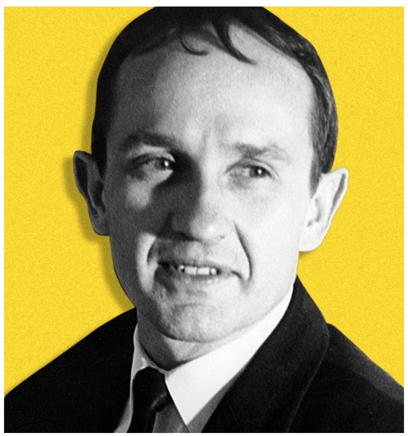
The purpose of the present paper is to give a formulation of the simple theory of types[1] which incorporates certain features of the calculus of λ-conversion.[2] A complete incorporation of the calculus of λ-conversion into the theory of types is impossible if we require that λx and juxtaposition shall retain their respective meanings as an abstraction operator and as denoting the application of function to argument. But the present partial incorporation has certain advantages from the point of view of type theory and is offered as being of interest on this basis (whatever may be thought of the finally satisfactory character of the theory of types as a foundation for logic and mathematics).

For features of the formulation which are not immediately connected with the incorporation of λ-conversion, we are heavily indebted to Whitehead and Russell,[3] Hilbert and Ackermann,[4] Hilbert and Bernays,[5] and to forerunners of these, as the reader familiar with the works in question will recognize.

Church (1941), p. 56.

Note: Church here refers to the difference between *typed* and *untyped lambda calculus* when he states that a complete integration of the two frameworks is impossible. An untyped lambda calculus is more powerful, i.e. more expressive, but evaluation of the untyped expression might not be possible.

7. Universal Grammar



There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed, I consider it possible to comprehend the syntax and semantics of both kinds of languages within a single natural and mathematically precise theory. On this point I differ from a number of philosophers, but agree, I believe, with Chomsky and his associates. It is clear, however, that no adequate and comprehensive semantical theory has yet been constructed,[1] and

Originally published in *Theoria* 36:373–98 (1970). Reprinted with permission.

Montague (1970), reprinted in Thomason (1974), p. 222.

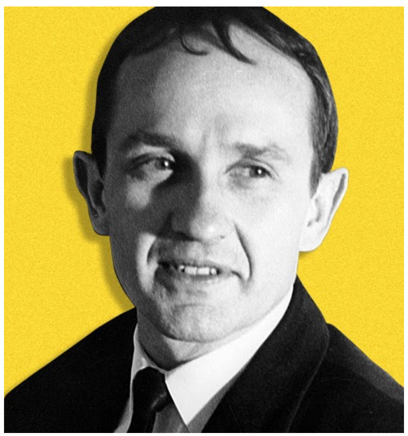
See also https://www.richardmontague.com/

4. SEMANTICS: THEORY OF REFERENCE

Let e, t, s be the respective numbers 0, 1, 2. (The precise choice of these objects is unimportant; the only requirements are that they be distinct and that none of them be an ordered pair.) By T, or the set of *types*, is understood the smallest set such that (1) e and t (which are regarded as the type of entities and the type of truth values respectively) are in T; (2) whenever $\sigma, \tau \in T$, the ordered pair $\langle \sigma, \tau \rangle$ (which is regarded as the type of functions from objects

228 RICHARD MONTAGUE

of type σ to objects of type τ) is in T; and (3) whenever $\tau \in T$, the pair $\langle s, \tau \rangle$ (which is regarded as the type of senses corresponding to objects of type τ) is in T. In connection with any sets E and I



Montague (1970), reprinted in Thomason (1974), p. 227-228.

Note: Besides *e* and *t*, Montague here introduces *s* as another type. This is used for a further elaboration of the framework which does not only deal with the extensions of expressions (denotational semantics), but also with *intensions* (e.g. English words like *seek*, *try*, *want*, etc.), and is hence called *intensional logic*.

Section 3: λ-Abstraction

The Syntax: Recursive Definition (Last Lecture)

The clauses for the syntax of a **type-theoretic language** are then:

(i) If α is a variable or a constant of type a in *L* [i.e. v_a or c_a], then α is an expression of type a in *L*.

(ii) If α is an expression of type $\langle a, b\rangle$ in *L*, and β is an expression of type a in *L*, then $(\alpha(\beta))$ is an expression of type b in *L*.

(iii) If ϕ and ψ are expressions of type *t* in *L* (i.e. formulas in L), then so are $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, and $(\phi \leftrightarrow \psi)$.

(iv) If ϕ is an expression of type *t* in *L* and v is a variable (of arbitrary type a), then $\forall v\phi$ and $\exists v\phi$ are expression of type *t* in *L*.

(v) If α and β are expressions in *L* which belong to the same (arbitrary) type, then $(\alpha = \beta)$ is an expression of type *t* in *L*.

(vi) Every expression *L* is to be constructed by means of (i)-(v) in a finite number of steps.

Gamut (1991), Volume 2, p. 81-82.

The Syntax: Adding the λ-clause

We simply add another clause to the **type-theoretic language** syntax:

(vii) If α is an expression of type a in *L*, and v is a variable of type b, then λv(α) is an expression of type $\langle b, a \rangle$ in *L*.[1]

Gamut (1991), Volume 2, p. 104.

[1] I added the brackets around α here, since at least in some cases these are necessary to disambiguate.

λ-Abstraction

We say that λv(α) has been formed from α by **abstraction over** the **formerly free variable v**. Hence, the free occurrences of v in α are now **bound by the λ-operator** λx.

Gamut (1991), Volume 2, p. 104.

Expression	**λ-abstraction**
S(x) of type *t*	λx(S(x)) of type $\langle e, t\rangle$

Note: The first-order predicate S of type $\langle e, t\rangle$ is applied to variable x of type *e*, and yields S(x) of type *t*. In the case of λ-abstraction this is simply reverted, i.e. the type of λx(S(x)) is $\langle e, t\rangle$ again.

Examples of λ-Abstractions

Assume a, b are constants and x, y variables of type e; A is of type $\langle e,t\rangle$; B is of type $\langle e,\langle e,t\rangle\rangle$; and X is of type $\langle e,t\rangle$.

Expressions	Types	λ-Abstraction	Types
x ✓	e	λx(x)	$\langle e,e\rangle$
A(x)✓	t	λx(A(x))	$\langle e,t\rangle$
B(y)(x) ✓	t	λx(B(y)(x)) or λy(B(y)(x))	$\langle e,t\rangle$
B(a)(x) ✓	t	λx(B(a)(x))	$\langle e,t\rangle$
$\forall$xB(x)(y)✓	t	λy($\forall$xB(x)(y))	$\langle e,t\rangle$
X(a) ✓	t	λX(X(a))	$\langle\langle e,t\rangle,t\rangle$
X(a) $\wedge$ X(b)✓	t	λX(X(a) $\wedge$ X(b))	$\langle\langle e,t\rangle,t\rangle$

Note: In our practical usage of the type-theoretic language, variables are mostly defined to have type e (i.e. x, y, z, etc.). In some cases, they might be of type $\langle e,t\rangle$, namely, if they refer to predicate variables (X, Y, Z, etc.). Hence, λ-abstraction essentially amounts to **adding an e or $\langle e,t\rangle$ as a "prefix"** to the type of the expression that is abstracted over.

More Examples: Recursive Application

Assume that B below represents the relation of "befriend" in English. The expression (B(y))(x) – here simplified to B(y)(x) – then represents "x befriends y".

Expressions	Types	λ-Abstraction	Types
B(y)(x)	t	λy(B(y)(x))	$\langle e, t\rangle$
λy(B(y)(x))	$\langle e, t\rangle$	λx(λy(B(y)(x)))	$\langle e, \langle e, t\rangle\rangle$

Note: B(y)(x) is equivalent to the standard predicate logic expression Bxy. We have already pointed out in the last lecture that for two-place predicates it is a convention in type-theoretic languages to apply the predicate first to what would be the object of a transtitive sentence (i.e. y here), and then secondly to what would be the subject (i.e. x). This is also reflected in the order of application of the λ-abstraction to first y and then x.

Eberhard Karls Universität Tübingen

Faculty of Philosophy
General Linguistics

Section 4: λ-Conversion

λ-Conversion (aka β-Reduction)

Informally speaking, **λ-conversion**[2] is the process whereby we reduce the λ-statement by removing the λ-operator (and the variable directly following it) and pluging an expression (in the simplest case a constant c, or a predicate constant C) into **every occurrence of the variable which is bound by the λ-operator**.

Typed expression	**λ-Abstraction (over x or X)**	**λ-Conversion (with c or C over x or X)**
S(x)	λx(S(x))	λx(S(x))(c) = S(c)
S(x) $\wedge$ D(x)	λx(S(x) $\wedge$ D(x))	λx(S(x) $\wedge$ D(x))(c) = S(c) $\wedge$ D(c)
X(a) $\wedge$ X(b)	λX(X(a) $\wedge$ X(b))	λX(X(a) $\wedge$ X(b))(C) = C(a) $\wedge$ C(b)

[2] The term λ-conversion is not to be confused with α-conversion. The latter refers to replacing one variable for another.

λ-Conversion (Formal Definition)

In general, **λ-conversion** is defined as the process whereby an expression of the form λv(β)(γ) is reduced to [γ/v]β.

- β is a typed expression (e.g. S(x) or S(x) $\wedge$ D(x) above).
- v is a variable (e.g. x, y, z, X, Y, Z).
- γ is another expression which the λ-expression is applied to (i.e. functional application). In the simple case, this is a constant c or C as above.
- [γ/v]β means all occurrences of v in β are replaced by γ.

Important caveat: This definition holds only if "all variables which occur as free variables in γ are free for v in β", i.e. if **no occurrence** of v in β is bound by a quantifier or another λ-operator.

Gamut (1991), Volume 2, p. 109-110.

Valid and Invalid λ-Conversions

Assume a, c, d are constants of type *e*; x and y are variables of type *e*; F, D, and S are predicate constants of type $\langle e, t\rangle$, A is of type $\langle e, \langle e, t\rangle\rangle$, and X is a predicate variable of type $\langle e, t\rangle$.

λ-Abstraction	λ-Conversion
$\lambda x(S(x) \wedge D(x))$	$\lambda x(S(x) \wedge D(x))(c) = S(c) \wedge D(c)$ ✓
$\lambda y(\lambda x(S(x) \wedge D(y)))$	$\lambda y(\lambda x(S(x) \wedge D(y)))(c)(d) =$ $\lambda x(S(x) \wedge D(c))(d) =$ $S(d) \wedge D(c)$ ✓
$\lambda x(\lambda y(A(y)(x)))$	$\lambda x(\lambda y(A(y)(x)))(c)(d) =$ $\lambda y(A(y)(c))(d) =$ $A(d)(c)$ ✓
$\lambda x(\forall x F(x))$	$\lambda x(\forall x F(x))(c)$ **x**
$\lambda x(\exists x F(x) \rightarrow S(x))$	$\lambda x(\exists x F(x) \rightarrow S(x))(c)$ **x**
$\lambda X(\forall X(X(a) \wedge X(b)))$	$\lambda X(\forall X(X(a) \wedge X(b)))(C)$ **x**

Note: While the rule for λ-abstraction given above in clause (vii) licenses all the abstractions in the left side, λ-conversion is only valid for a subset of these, namely the ones where the variable v is not bound by a quantifier. In practice, this means we generally avoid λ-abstractions of variables which are already bound.

Section 5: Modelling Compositionality with λ-Calculus

Disclaimer

"Now it is, of course, also possible to treat the composition of predicates without a λ-operator [...] **So why do we need a λ-operator?** The advantage of the λ-operator is that it provides a **uniform treatment** not only of these examples [combination of predicates] but also of many others too."

Gamut (1991), Volume 2, p. 107.

Why is λ-calculus needed?

If our aim is to model not only full sentences and formulas representing predicates, but also parts of sentences, and even individual words, by using in a unified account, then λ-abstraction and λ-conversion are possible solutions. Thus, λ-calculus allows us to capture the **compositionality of language**.

English sentence	**Typed expression**
John smokes and drinks.	$\lambda x(S(x) \wedge D(x))(j) = S(j) \wedge D(j)$
John smokes	$\lambda x(S(x))(j) = S(j)$
smokes	$\lambda x(S(x))$
drinks	$\lambda x(D(x))$
smokes and drinks	$\lambda x(S(x) \wedge D(x))$

Further Examples: Two-Place Predicates

English sentence	**Typed expression**
Jumbo likes Bambi.	λx(λy(L(y)(x)))(j)(b) = L(b)(j)
Jumbo likes	λy(L(y)(j))
likes Bambi	λx(L(b)(x))
likes	λx(λy(L(y)(x)))

Note: Gamut (1991), Volume 2, p. 107-108, discuss how the active relation “likes” might be represented differently from the passive relation “is liked by”. Namely, assume we have L(y)(x) representing “x likes y”. If we first abstract over x we get λx(L(y)(x)) which represents “likes y” (since x is now bound by the λ-operator). If we then abstract over y we get λy(λx(L(y)(x))), which represents “likes”. If we do it the other way around, however, we first get λy(L(y)(x)) which represents “is liked by x”, and in the second step we get λx(λy(L(y)(x))) “is liked by”. However, we will here assume that active and passive sentences are equivalent in terms of their λ-representations, otherwise we couldn’t represent a structure like “Jumbo likes”, since λy(L(y)(j)) would strictly translate as “is liked by Jumbo”.

Further Examples: The Copular "Be"

English sentence	**Typed expression**
Jumbo is grey.	λx(G(x))(j) = G(j)
is grey	λx(G(x))
Jumbo is	λX(X(j))
is	λX(λx(X(x)))

Note: In order to just represent be/is here, we have to use a predicate variable X to represent all possible one-place predicates. Since in the last step we abstract both over the variable standing in for the individual (x), and the variable standing in for a predicate applied to the individual (X), all that remains is the copular. Of course, this means that the whole sentence *Jumbo is grey* could also be represented as a λ-expression, i.e. λX(λx(X(x)))(G)(j) which is equivalent to G(j) after λ-conversion. This representation of the copular in a λ-expression is found in Kearns (2011), p. 78.

Further Examples: Quantified Expressions

English sentence	**Typed expression**
Every man walks.	$\forall x(M(x) \rightarrow W(x))$
every man	$\lambda X(\forall x(M(x) \rightarrow X(x)))$
every	$\lambda Y(\lambda X(\forall x(Y(x) \rightarrow X(x))))$
Some man walks.	$\exists x(M(x) \wedge W(x))$
some man	$\lambda X(\exists x(M(x) \wedge X(x)))$
some	$\lambda Y(\lambda X(\exists x(Y(x) \wedge X(x))))$
No man walks.	$\neg\exists x(M(x) \wedge W(x))$
no man	$\lambda X(\neg\exists x(M(x) \wedge X(x)))$
no	$\lambda Y(\lambda X(\neg\exists x(Y(x) \wedge X(x))))$

Note: In order to represent just the quantifiers *every*, *some*, and *no* we need two predicate variables here (X and Y). Same as before, whole sentences can also be represented by λ-expressions, e.g. *every man walks* is $\lambda Y(\lambda X(\forall x(Y(x) \rightarrow X(x))))(M)(W)$ before λ-conversion, which becomes (and is hence equivalent to) $\forall x(M(x) \rightarrow W(x))$ after λ-conversion.

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Faculty of Philosophy
General Linguistics

Section 6: The Semantics of λ-Calculus

Truth Valuation

As seen before for other logical languages, the semantic side of type-theoretic languages consists of the **valuation of truth** given a syntactically valid expression.

Example

Remember from last lecture that in a type-theoretic language we might represent a verb like *walks* by W of type $\langle e, t \rangle$. In order to valuate the truth of a particular formula (e.g. W(j)) we define the set of all relevant entities D (the domain) with members d, and a subset $W \subseteq D$ whose members can be said to *walk*. We then define an interpretation function I for which it holds that:

$$I(W)(d) = 1 \text{ iff } d \in W; \text{ and } I(W)(d) = 0 \text{ iff } d \notin W. \quad (1)$$

I(W) is a so-called *characteristic function of W* (over D).

Truth Valuation: λ-Calculus

It is important to realize that **λ-calculus is not just a syntactic extension to type-theoretic languages**, i.e. some descriptive convention of how to encode certain parts of natural language sentences, but it is also fully compatible with the **semantic side of truth valuation**.

Example

For an expression W(x) we have the problem that while it is of type *t* we cannot actually assign a truth value $\{0, 1\}$ to it. It can be shown, however, that λx(W(x)) is a function h such that

$$h = I(W). \quad (2)$$

In other words, for all entities d in the domain D it holds that h(d)=1 iff I(W)(d)=1. This illustrates that the denotation of λx(W(x)) is indeed the same as one would expect for just the word *walks* represented by W.

Gamut (1991), Volume 2, p. 105.

Summary

Summary

- λ-calculus enables us to **represent the semantic compositionality** of natural language sentences even below the level of predicates and formulas.
- There are two main processes of λ-calculus, namely, λ-abstraction and λ-conversion. The former leads to the binding of formerly unbound variables, the latter can be used to reduce complex λ-expressions to simpler ones by pluging in the constants.
- The **semantic interpretation** of λ expressions is fully compatible with truth valuation as defined for logical languages more generally.

References

References

Gamut, L.T.F (1991). *Logic, Language, and Meaning. Volume 1: Introduction to Logic*. Chicago: University of Chicago Press.

Gamut, L.T.F (1991). *Logic, Language, and Meaning. Volume 2: Intensional Logic and Logical Grammar*. Chicago: University of Chicago Press.

Kroeger, Paul R. (2019). *Analyzing meaning. An introduction to semantics and pragmatics.* Second corrected and slightly revised version. Berlin: Language Science Press.

Thomason, Richmond H. (1974). *Formal Philosophy. Selected papers pf Richard Montague.* New Haven/London: Yale University Press.

Zimmermann, Thomas E. & Sternefeld, Wolfgang (2013). *Introduction to semantics. An essential guide to the composition of meaning.* Mouton de Gruyter.

Thank You.

Contact:

Faculty of Philosophy
General Linguistics
Dr. Christian Bentz
SFS Wilhelmstraße 19-23, Room 1.15
chris@christianbentz.de
Office hours:
During term: Wednesdays 10-11am
Out of term: arrange via e-mail