Visualizing and Computing Natural Language Expressions: Through a Typed Lambda Calculus λ

Harjit Singh

1 Introduction

Church introduced a Lambda operator λ in 1941 and it is a vital tool used in syntax and semantics. Since Montague's times, a typed formation of lambda abstraction has been popular in linguistics [1].[1] Sometimes, a lambda operator can be defined under lambda expressions to focus on a specific property in a context. It appears such as (a term = property) and fetching the predicate expressions in a second order logic and first order logic, respectively [2].[2] On the other hand, a lambda calculus is a set of expressions and rules that produce certain new expressions. In general, a single typed and monotyped lambda calculus is discussed many times in a literature [3].

However, it is a fact that the first order logic in contrast to a simple typed lambda calculus allows infinite types of expressions that fundamentally enumerate from the finite forms.[3] Secondly, typed lambda notations are standard in mathematics and

[1] It discusses a well-formed system in semantics through a systematic arrangement of typed mechanism in terms of natural language expressions that are semantically motivated. It represents the following

(i) e is a type
(ii) t is a type then
(iii) both <e, t> is a type.

[2] See Allwood et al. [2, p. 156].

[3] Carpenter [4, p. 40] has pointed out that in a simple typed lambda calculus, each type either belong to a basic type or a functional type.

$$\text{Basic type} \leq \text{Typ}$$
$$(\sigma \rightarrow \tau) \in \text{Typ if } \sigma, \tau \in \text{Typ}$$

H. Singh (✉)
Indira Gandhi National Tribal University, Amarkantak, MP 484887, India

M. K. Bajpai et al. (eds.), *Machine Vision and Augmented Intelligence—Theory and Applications*, Lecture Notes in Electrical Engineering 796,
https://doi.org/10.1007/978-981-16-5078-9_49

computer science. At the same time, a single term calculus in itself has denoted a formal representation of syntactic structures that are somehow different from the first order and high order logic. On the other hand, we find three schemes when we seeing the axiomatic nature of a simple typed lambda calculus [4]. The following schemes are as

(a) α reduction
$\vdash \lambda x.\alpha \rightarrow \lambda y.\ (\alpha\ [x \rightarrow y])$

$[y \notin \text{Free}\ (\alpha)\ \&\ y \text{ is free for } x \text{ in } \alpha]$

(b) β reduction
$\vdash (\lambda x.\alpha)\ (\beta) \rightarrow \alpha\ [x \rightarrow \beta]$
$[\beta \text{ free for } x \text{ in } \alpha]$

(c) η reduction
$\vdash \lambda x.(\alpha\ (x)) \rightarrow \alpha$

$[x \notin \text{Free}\ (\alpha)]$

In the context of a natural language (i.e. English), lambda operator λ resolves passive and other cases within the propositional functions. It transforms such a function into one place predicate situation and binds the variables (x, y) in abstraction to define the expressions in the following way.[4]

(d)	X kicked chaster	<e, t> type
	(kick' (chester')) (x)	Propositional Expression
	λ x [(kick' (chester')) (x)]	Lambda Expression

Here (d) shows that variable (x) is bound by the lambda operator and it is a well-formed expression for <e, t> type [5, p. 116].

The paper has a total of five sections. The first section begins with the basic introduction of lambda calculus. The second section discusses the syntactic and semantic background of lambda calculus. The third section deals with the aims/objectives of the study. The fourth section analyzes the natural language expressions concerning lambda abstraction and application. The fifth section concludes the results and the future research.

[4] For more details see Cann [5, p. 116].

2 Related Works

As already has been pointed out lambda abstraction finds significant during Montague, and later many semanticists incorporated this into linguistics. It became a powerful tool to establish formal semantics. The following Fig. 1 shows the grammatical nature of a lambda expression.

Figure 1 interprets how lambda expression applies in relative clause, predicates and many other cases in a natural language. However, a classic instance of the lambda abstraction usually defines under 'the proper treatment of quantifiers in English'. See the Table 1.

Table 1 specifies that syntactic rules on the left handside translate into parallel with lambda symbol λ that directly controls the NP and VP constitutes [1, 6, p. 350].

In fact, lambda operator/abstraction operator λ is a kind of binder which denotes the infinite set of individuals in L_1. In that case, it shows the characteristic function of the set and sometimes, it shows the value description with the notation φ. On the other hand, when both lambda operator and value notation removes from the set, the left part is called β-reduction. Based on such function, almost many predicates like love, like, kill, eat, see, meet, etc., defines effortlessly [7, pp. 94–95].

Table 2 demonstrates that a predicate 'like' can be expressed with various names (i.e. Mary, John, Bill, Keat, etc.) bind with λ operator. It has a wide range here and may cover set of individual those who like m = Mary. Secondly, the value notation φ characterizes the set of individuals. And at last, both lambda operator λ and value notation φ can be removed from the set of individuals to form a β-reduction situation.

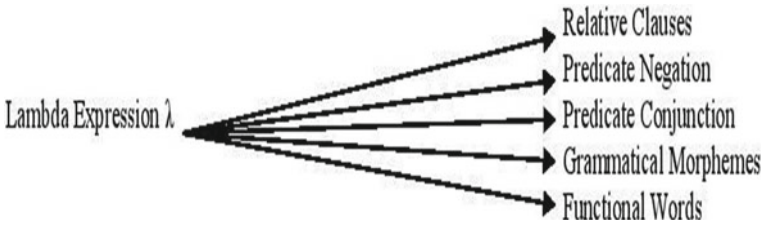

Fig. 1 Lambda expression λ with grammar

Table 1 Syntax (rules and interpretations)

Syntactic rules	Translation
S → S and S	S'_1 & S'_2
S → S or S	$S'_1 \vee S'_2$
VP → VP and VP	λx (VP'_1 (x) & VP'_2 (x))
NP → NP or NP	λP. (NP'_1 (P) & NP'_2 (P))

Table 2 Syntax (categories and λ operator)

Syntactic categories	λ operator
Predicates and variables	λx. Likes (m, x)
Predicate (Likes) with set of individuals (D_e)	λx. Likes (x, m)
Value description	[λx. ϕ]
β-reduction	Loves (m, h)

3 Aims and Objectives

- To present a general survey on lambda operator λ
- To analysis natural language expressions in syntax and semantics through typed lambda calculus λ
- To compute results when typed lambda calculus switches from one natural language to another
- To propose an algorithm based on semantics of typed lambda calculus λ.

4 Analysis with Typed Lambda Calculus λ

We know that all objects around us identify with certain names, symbols, and terms, etc. Moreover, they are significant for a natural language considered English, Hindi, Punjabi, Marathi, Malayalam, etc. It is fascinating and challenging for us to develop any logical system for such languages. We take English as a formal language to understand the logical system for its expressions through typed lambda calculus. We begin with the following proposition.

(i) Bill loves Elysha

It is a combination of Subject/NP and Predicate/VP where an individual constant Bill and Elysha respectively around the verb 'loves'. Here V denotes a verb that has binary relations and it is called a binary predicate. The 'loves' predicate attracts both arguments, but the first argument is an empty slot intituitvely. The following way defines this in L_1.

Loves (b, e)
Loves (_____, e)

We can compare the empty situation of 'loves' predicate with an abstraction that requires the formal representation to fill it up. See Table 3.

Table 3 Propositions with Lambda λ

Proposition	Explanation
P_1	Bill loves Elysha
Predicate (Binary)	Loves (b, e)
Intuition ground (Empty slot)	Loves (_____, e)
Abstraction field with λ	λ
λ with variable x	λx. Loves (x, e)

Table 4 Lambda abstraction

Lambda abstraction (syntax rule)	Lambda abstraction (semantics rule)
If α is an expression of type t and u variable of type σ then [λu] is an expression of type <σ, t>. We say σ and t as input type and output type of this expression	If α is an expression of type t and u variable of type σ then $[[\lambda u\ \alpha]]^{Mg}$ is that function f from Dσ into D_t such that for all objects o in Df (o) $= [[\alpha]]^{Mg\ [u \to o]}$

Adapted from Coppock and Champollion [6, p. 171]

Table 3 shows that a proposition 'Bill loves Elysha' has a binary predicate where the empty slot (_____) is equal to abstraction. It fills up with a lambda operator λ under the name of a variable x.[5]

4.1 Lambda Abstraction (Syntax and Semantics)

Furthermore, we must discuss both syntax and semantics-based lambda abstraction rules.

Table 4, suggests that the input type goes to set of individuals and [λu. α] is the output expression. At the same time, the semantics rule gives $[[\lambda u.\ \alpha]]^{M,g}$ expression if there is an α and the domain of type must be under truth values.

4.2 Lambda Operator λ in Syntax

In syntax, individuals and their truth values and statements with conjunctions, disjunctions, quantifiers and formulas are TYPE only according to simply-typed lambda calculus. Thus, each expression carries at least one type of expression. Table 5 induces such expressions formally as.

Of these seven categories, the first is the basic level information about variables such as x, y, z in a constant form. The second application part deals with α and β; those also type expressions. The third part shows that α and β both represent similarity if

[5] Note that *e* denotes entity and *t* denotes truth values however both *e*, *t* used for functional types such as <e, t>. These functional types also called a set of infinites. They denote individuals and truth values under the domain. It is represented by D_e = the domain of individuals and D_t expresses the truth values in the form of $D_t = \{1, 0\}$ Coppock and Champollion [6, pp. 167–68].

Table 5 Lambda and syntax

Basic expressions	Application	Equality	Negation	Binary	Quantification	Lambda abstraction
C_t, n (constant) $V_{t,}$ n (variable)	For any types σ and t is an expression of type <σ, t> and β is an expression of type σ then [α] (β) is an expression of type t	If α and β are items, then α = β is an expression of type t	If ϕ is a formula, then so is ¬ϕ	If ϕ and Ψ are formulas then so are ¬ϕ, [ϕ ∧ Ψ]. [ϕ v Ψ], and [ϕ ↔ Ψ]	If ϕ is a formula and u is a variable of any type, then [∀ u ϕ] and [∃ u ϕ] are formulas	If α is an expression of type t and u is a variable of type σ then [λ u α] is an expression of type <σ, t>

Adapted from Coppock and Champollion [6, pp. 180–81]

they are terms. The negation part tells us that if φ is a formula then it can appear with ¬ also. Binary connectives mean that if any formula comes with φ ψ then it can be combined with ∨, ∧, ¬, ↔ like connectives. In quantification, a universal quantifier ∀ and existential quantifier ∃ exist with variables in a formula. On the other hand, in the lambda abstraction, any t expression binds with a lambda operator λ.

4.3 Lambda Operator λ in Semantics

Model and assignment function determine the natural language expressions by mapping the semantic values in L_λ. Further, we say that <D, I> both are necessary for M as a model that assigns the semantic values.[6] The following way describes this in L_1.

$$L_1 = M <D, I>$$
$$D = D_e\ D_t$$
$$I = \text{assignment function}$$
$$D = \text{Domain for individuals that represent through the types } e$$
$$D = \text{Domain for truth values } t$$

Table 6 shows that the first slot is for basic expressions with a non-logical constant and a variable. The second one is about the application of α and β expressions in the context of type. The third 'equality slot' discusses the truth value of α and β if they belong to the same type. The fourth 'negation slot' tell us that the φ formula translates into a ¬φ form. The fifth slot is the (negative, conjunctive, junction, etc.)

[6] Remember that in a model M = <D, I>, D is the domain for a set of individuals which interpretes with D_e and D_t describes the truth values in the form of {0, 1}.

Table 6 Lambda and semantics

Basic expressions	Application	Equality	Negation	Binary connectives	Quantification	Lambda abstraction
If α is a non logical constant then $[[\alpha]]^{Mg=1\ (\alpha)}$	If α is an expression of type <σ, t> and β is an expression of type σ then $[[\alpha (([[\beta]]^{Mg} = [[\alpha]]^{Mg} = (([[\beta]]^{Mg})$	If α and β related with same type then $[[\alpha = \beta]]^{Mg} = 1$	If ϕ is a formula, then $[[\neg\phi]]^{Mg} = 1$ iff $[[\phi]]^{Mg} = 0$	If ϕ and Ψ are formulas, then $[[\phi \wedge \Psi]]^{Mg} = 1$ iff $[[\phi]]^{Mg} = 1$ and $[[\Psi]]^{Mg} = 1$	If ϕ is a formula and v is a variable the t type $[[\forall$ v. $\phi]]^{Mg} = 1$ iff for all o ϵ D: $[[\phi]]^{Mg\ [v \rightarrow o]}$	If α is an expression of type t and u us a variable of type σ then $[\lambda$ u $\alpha]^{Mg}$ is that function *f* from D_σ into D_t such that for all objets o in D_σ *f* (o) = $[[\alpha]]^{Mg\ [\sigma \rightarrow o]}$

Adapted from Coppock and Champollion [6, pp. 187–88]

binary connectives formed by wffs. The sixth 'quantification slot' deals with universal and existential quantifiers, and the last seventh slot shows the lambda λ as a binder for types [7, 8, pp. 170–88].

5 Discussions and Results

Based on the above lambda abstraction λ in Tables 5 and 6, we take as input to compare two different languages to see the actual output. The first abstraction rule in syntax begins with usual expressions and ends with the lambda λ. It is sufficient to generalize the formal syntactic representations in both languages here. Secondly, abstraction rule in semantics has similar categorization and it computes such languages.

We see Tables 7 and 8 where both languages (English and Punjabi) have 1:1 correspondance in syntax and semantics of lambda calculus λ. Moreover, two natural languages are going to map in the same way. In other words, we argue that any 'y' kind of a natural language gets generated by formal language 'xs' . However, double complex and compound predicates, reduplicated forms and mainly addressing notes are typical instances that will discuss by following intention and intuition-based research and applications [8, 9].

Algorithm

Step 1: Start with assumption of type e, t for all objects

Step 2: Search the abstraction slots

Step 3: Fill up abstraction slots with a lambda operator λ

Step 4: Allow lambda operator λ as binder with variables x, y, and z

Step 5: Maintain model and assignment function in L_1

Table 7 Syntactic representations in English and Punjabi

Basic Expressions	Application	Equality	Negation	Binary Connectives	Quantification	Lambda Abstraction
English: e, t ϵ c, v	α, β ϵ t, σ α = s_t, β = P_t English: α	α = β both languages (English and Punjabi) have same equal relations between α and β	Φ is a rule and ¬ϕ is also rule. English: predicates (likes, loves, kills and ¬) Punjabi: predicate (likes, loves, kills and ¬)	Φ v/∧ Ψ is a formula. English and Punjabi have also shown predicates with binary connectives $P_1 \wedge P_2$, P_1 V P_2	Universal quantification [∀] and existential quantifier [∃]	α is t and λα Both English and Punjabi have α is bounded with λ
Punjabi: e, t ϵ c, v	[Subject] β = [predicate] Punjabi: α = [subject] β = [predicate]					
1:1	1:1	1:1	1:1	1:1	1:1	1:1

Table 8 Semantic representations in English and Punjabi

Basic expressions	Application	Equality	Negation	Binary connectives	Quantification	Lambda abstraction
English: α either a non-logical constant or α is a variable	α, β are type expressions in Model. Both English and Punjabi follow the type expression in the form of α and β	$\alpha = \beta$ then $[[\alpha = \beta]]^{Mg} = 1$ Both languages (English and Punjabi) have equal relations between α and β and if α is true then β must be true	ϕ is a rule and $\neg\phi$ is also a rule English and Punjabi have similar negative representations in the case of $[[\neg\phi]]^{Mg} = 1$ iff $[[\phi]]^{Mg} = 0$	Φ v/∧ Ψ is a formula. English and Punjabi have also shown predicates with binary connectives $P_1 \wedge P_2$, P_1 V P_2 in M with assignment function	Universal quantification [∀] and existential quantifier [∃] Both quantifiers are part of set of individuals in D	α is t and λα Both English and Punjabi have similar structure of D_σ and D_t
English: α either a non-logical constant or α is a variable						
1:1	1:1	1:1	1:1	1:1	1:1	1:1

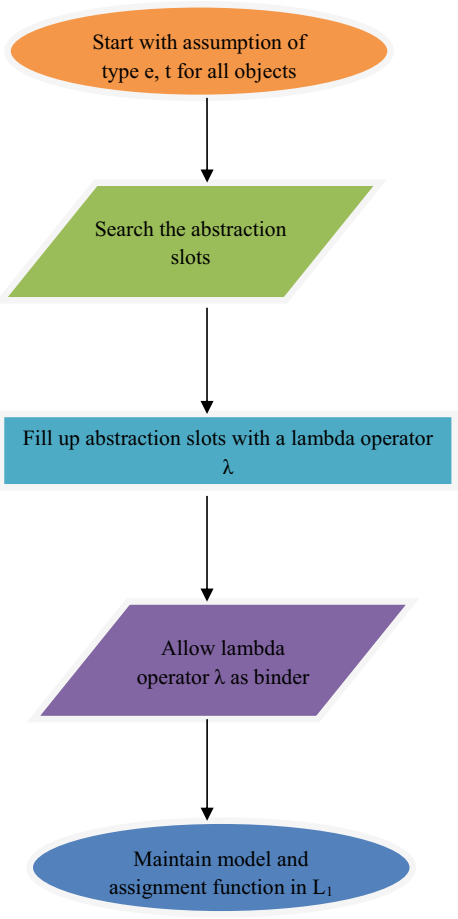
Start with assumption of type e, t for all objects
Search the abstraction slots
Fill up abstraction slots with a lambda operator λ
Allow lambda operator λ as binder
Maintain model and assignment function in L_1

Chart 1

Step 1: Objects with <*e*, *t*/*et*>

Step 1: Objects with <*e*, *t*/*et*>

$CN(AGR_1)$	Predicate	$CN(AGR_2)$
Man	Loves	Woman
Woman	"	Man
Boy	"	Girl
Girl	"	Boy
Husband	"	Wife
Wife	"	Husband
Father	"	Mother
Mother	"	Father
Grandfather	"	Grandmother
Uncle	"	Aunt

$$e, t/et \begin{Bmatrix} CN(AGR_1): M_x \ldots\ldots U_x \\ CN(AGR_2): W_x \ldots\ldots A_x \end{Bmatrix}$$

$PN(AGR_1)$	Predicate	$PN(AGR_2)$
Sandeep	Loves	Kavita
Kala	"	Mala
Mehta	"	Kiran
Dalip	"	Deepa
Shamlal	"	Simran
Aman	"	Geeta
Sushil	"	Amita
Kuljeet	"	Sunita
Mahesh	"	Meena
Suresh	"	Sangeeta
Harmeet	"	Mamta
Amar	"	Sarita

$$e, t/et \begin{Bmatrix} PN(AGR_1): S_x \ldots\ldots S_x \\ PN(AGR_2): K_x \ldots\ldots S_x \end{Bmatrix}$$

Under step 1 we take two sets of common nouns (CNs) and proper nouns (PNs) with a single predicate 'loves'. AGR refers to another name such as arguments of both CNs and PNs that represent either *e* type or *t* type in a discourse [10, 11].

Step 2: Slots (_____)

CN (AGR_s)	PN (AGR_s)
Loves (_____, W)	Loves (_____, K)
Loves (_____, M)	Loves (_____, M)
Loves (_____, G)	Loves (_____, K)
Loves (_____, B)	Loves (_____, D)
Loves (_____, W)	Loves (_____, S)
Loves (_____, H)	Loves (_____, G)
Loves (_____, M)	Loves (_____, A)
Loves (_____, F)	Loves (_____, S)
Loves (_____, G)	Loves (_____, M)
Loves (_____, A)	Loves (_____, S)

According to step 2, it shows that an external or a front AGR place must be an empty slot in a predicate 'loves'. Such slot finds with CNs and PNs sets.

Step 3: Slots with lambda operator λ

Step 3: Slots with lambda operator λ

CN (AGR_s)	PN (AGR_s)
Loves (_____, λ_W)	Loves (_____, λ_K)
Loves (_____, λ_M)	Loves (_____, λ_M)
Loves (_____, λ_G)	Loves (_____, λ_K)
Loves (_____, λ_B)	Loves (_____, λ_D)
Loves (_____, λ_W)	Loves (_____, λ_S)
Loves (_____, λ_H)	Loves (_____, λ_G)
Loves (_____, λ_M)	Loves (_____, λ_A)
Loves (_____, λ_F)	Loves (_____, λ_S)
Loves (_____, λ_G)	Loves (_____, λ_M)
Loves (_____, λ_A)	Loves (_____, λ_S)

Step 3 determines that the empty slots in CNs and PNs are abstraction places that fills up with a lambda operator λ. We add this operator in front of each empty slot.

Step 4: Lambda operator λ as binder for variables x

$$\lambda x.\ \text{Loves}\ (x, w\ldots\ldots A)$$
$$\lambda x.\ \text{Loves}\ (x, k\ldots\ldots S)$$

In step 4, we argue that a lambda operator λ binds x variable in the same slot. Because it is difficult to select each name separately for the same function here.

Step 5: Model and assignment functions

$$M = \langle D, I \rangle \text{ and } g$$
$$M = D_e \text{ (Any type } e, \textit{t/et} \text{ in CN or PN in Step 1)}$$
$$M = D_t (1, 0)$$

The step 5 defines the formal representation with a model (M) that contains (D) and (I) function assignment as g. Any set of individuals, as appears in step 1, is a part of e, t/et domain which directly links with truth values (1, 0).

6 Conclusion

We understand that a language like English has a formal representation. Small and simple statements in any natural language are expressions that are nothing but a set of finite or infinite types. We find that a lambda operator λ describes such infinite sets better because it takes all types of expressions (equality-based, negation-based, and so on) in the form of a type and binds them to avoid repeating any name. It also frames syntax and semantics of a natural language. Following such observations, a proposed algorithm helps us to analyze small CNs and PNs individual sets.

References

1. Partee BB, Ter Meulen AG, Wall R (2012) Mathematical methods in linguistics, vol 30. Springer, Dordrecht
2. Allwood J, Andersson GG, Andersson LG, Dahl O (1977) Logic in linguistics. Cambridge University Press, Cambridge
3. Gillon BS (2019) Natural language semantics: formation and valuation. MIT Press, Massachusetts
4. Carpenter B (1997) Type-logical semantics. MIT Press, Massachusetts

5. Cann R (1993) Formal semantics: an introduction. Cambridge University Press, Cambridge
6. Coppock E, Champollion L (2019) Invitation to formal semantics. Manuscript, Boston University and New York University. http://eecoppock.info/semantics-boot-camp.pdf
7. Barendregt HP (1984) The lambda calculus: its syntax and semantics. In: Studies in Logic, vol 103. NorthHolland Publishing Company, Netherlands
8. Moggi E (1988) Computational lambda-calculus and monads. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science
9. Lamping J (1989, December) An algorithm for optimal lambda calculus reduction. In: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on principles of programming languages, pp 16–30
10. Liang P (2013) Lambda dependency-based compositional semantics. Technical Report, arXiv: 1309.4408
11. Boudol G, Curien PL, Lavatelli C (1999) A semantics for lambda calculi with resources. Math Struct Comput Sci 9(4):437–482