Lambda Dependency-Based Compositional Semantics

名字比较拗口：
《基于Lambda依赖关系的 组合 语义》

Percy Liang

September 19, 2013

Abstract

This short note presents a new formal language, lambda dependency-based compositional semantics (lambda DCS) for representing logical forms in semantic parsing. By eliminating variables and making existential quantification implicit, lambda DCS logical forms are generally more compact than those in lambda calculus.

本文提出了一种新的形式化语言，lambda依赖的组合语义（lambdacs），用于表示语义分析中的逻辑形式。通过消除变量和隐式存在量化，lambda-DCS逻辑形式通常比lambda演算中的逻辑形式更紧凑。

1 Introduction

Semantic parsing is the task of mapping natural language utterances to logical forms in some formal language such as lambda calculus. This short note describes *lambda dependency-based compositional semantics* (lambda DCS), an alternate formal language which can be notationally simpler than lambda calculus. Here is an example:[1]

- Utterance: *"people who have lived in Seattle"*

- Logical form (lambda calculus): $\lambda x.\exists e.\texttt{PlacesLived}(x, e) \wedge \texttt{Location}(e, \texttt{Seattle})$

- Logical form (lambda DCS): `PlacesLived.Location.Seattle`

As one can see, lambda DCS attempts to remove explicit use of variables. This makes it similar in spirit to dependency-based compositional semantics (DCS) (Liang et al. 2011), but DCS is restricted to tree-structured logical forms, and is therefore not as expressive as lambda calculus. Lambda DCS, on the other hand, borrows the use of variables and lambda abstraction from lambda calculus when necessary to handle phenomena such as anaphora.

Lambda DCS was designed in the context of building a natural language interface into Freebase (Bollacker et al. 2008), a large graph database, where the logical forms are database queries (see Berant et al. (2013) for our companion paper). Therefore, we will focus on representing the semantics of noun phrases or wh-questions. Also, we will discuss only the semantics of the formal language, not the compositional semantics of mapping natural language utterances into lambda DCS. Finally, this document assumes the reader has basic familiarity with lambda calculus in the context of natural language (see Carpenter (1998) for an introduction).

---

[1]This example is based on the Freebase schema, which reifies events (called compound value types in Freebase). Here, $\texttt{PlacesLived}(x, e)$ denotes that a person x was involved in a living *event* e, and $\texttt{Location}(e, \texttt{Seattle})$ denotes that the location of that event is Seattle. Other properties of e would include $\texttt{StartDate}(e, \cdot)$ and $\texttt{EndDate}(e, \cdot)$.

2   Definition

We now present the full definition of lambda DCS. In the background, we have a *knowledge base* of assertions. Let $\mathcal{E}$ be a set of entities (e.g., Seattle $\in \mathcal{E}$), $\mathcal{P}$ be the set of properties (e.g., PlaceOfBirth $\in \mathcal{E}$). Then $\mathcal{K} \subset \mathcal{E} \times \mathcal{P} \times \mathcal{E}$ is the knowledge base, which can be visualized as a directed graph where nodes are entities and edges represent assertions in the knowledge graph. Also, let $\mathcal{V}$ be a set of variables (e.g., $x_1, x_2 \in \mathcal{V}$).

Let [condition] denote the truth value of the condition. For example, $\lambda x.[x = 3]$ denotes the function that returns true if and only if its argument is equal to 3.

We will now walk through each of the constructs in lambda DCS with concrete examples, and also provide a formal definition of their semantics by providing a formal conversion to equivalent lambda calculus forms. Notationally, let $[\![z]\!]$ be the lambda calculus form corresponding to the lambda DCS form z. In the following definitions, let x, y denote fresh variables which are not used anywhere else.

Unary base case   The simplest lambda DCS form is a single entity. For example,

$$\text{Seattle} \tag{1}$$

denotes the singleton set containing Seattle. The equivalent in lambda calculus is:

$$\lambda x.[x = \text{Seattle}]. \tag{2}$$

In general, for an entity $e \in \mathcal{E}$, e is a unary logical form representing the indicator function that returns true when its argument is equal to e:

$$[\![e]\!] = \lambda x.[x = e]. \tag{3}$$

Binary base case   The next example is

$$\text{PlaceOfBirth}, \tag{4}$$

which denotes the set of pairs of people and where they were born:

$$\lambda x.\lambda y.\text{PlaceOfBirth}(x, y). \tag{5}$$

In general, for a property $p \in \mathcal{P}$, p is a binary logical form, which denotes a function mapping two arguments to whether p holds:

$$[\![p]\!] = \lambda x.\lambda y.p(x, y). \tag{6}$$

Join   The most central operation in lambda DCS is join. For example, *"people born in Seattle"* is represented as

$$\text{PlaceOfBirth.Seattle} \tag{7}$$

in lambda DCS and as

$$\lambda x.\text{PlaceOfBirth}(x, \text{Seattle}) \tag{8}$$

2

in lambda calculus. In general, for a binary logical form b and unary logical form u, we have that $b.u$ is a unary logical form which denotes:

$$\llbracket b.u \rrbracket = \lambda x.\exists y.\llbracket b \rrbracket(x,y) \wedge \llbracket u \rrbracket(y). \tag{9}$$

The key feature of a join is the *implicit existential quantification* over the argument y shared by b and u. From a database perspective, this is simply a join over relations b and u on the second argument of b and the first (and only) argument of u and then projecting out the joined variable.

The advantage of lambda DCS is more apparent when binaries are chained. For example, consider *"those who had children born in Seattle"*:

$$\texttt{Children.PlaceOfBirth.Seattle}, \tag{10}$$

and its lambda calculus equivalent:

$$\lambda x.\exists y.\texttt{Children}(x,y) \wedge \texttt{PlaceOfBirth}(y, \texttt{Seattle}). \tag{11}$$

Here, $\texttt{Children}(x,y)$ denotes whether x has a child y and $\texttt{PlaceOfBirth}(y, \texttt{Seattle})$ denotes whether y was born in Seattle.

A logical form $p_1. \cdots .p_k.e$ corresponds to a chain-structured graph pattern on the knowledge base $\mathcal{K}$ which matches any entity (node) which can reach e by following k edges labeled with the given properties $p_1, \ldots, p_k$.

Intersection   The set of scientists born in Seattle in lambda DCS:

$$\texttt{Profession.Scientist} \sqcap \texttt{PlaceOfBirth.Seattle} \tag{12}$$

and in lambda calculus:

$$\lambda x.\texttt{Profession}(x, \texttt{Scientist}) \wedge \texttt{PlaceOfBirth}(x, \texttt{Seattle}). \tag{13}$$

In general, for two unaries u_1 and u_2 $u_1 \sqcap u_2$ is a unary logical form representing:

$$\llbracket u_1 \sqcap u_2 \rrbracket = \lambda x.\llbracket u_1 \rrbracket(x) \wedge \llbracket u_2 \rrbracket(x). \tag{14}$$

In terms of the graph pattern perspective, intersection allows tree-structured graph patterns, where branch points correspond to the intersections.

Union   Whereas intersection corresponds to conjunction, union corresponds to disjunction. The set containing *"Oregon, Washington and Canadian provinces"* in lambda DCS:

$$\texttt{Oregon} \sqcup \texttt{Washington} \sqcup \texttt{Type.CanadianProvince} \tag{15}$$

and in lambda calculus:

$$\lambda x.[x = \texttt{Oregon}] \vee [x = \texttt{Washington}] \vee \texttt{Type}(x, \texttt{CanadianProvince}). \tag{16}$$

In general, for two unaries u_1 and u_2 $u_1 \sqcup u_2$ is a unary logical form representing:

$$\llbracket u_1 \sqcup u_2 \rrbracket = \lambda x.\llbracket u_1 \rrbracket(x) \vee \llbracket u_2 \rrbracket(x) \tag{17}$$

Negation Negation is straightforward. Here is *"states not bordering California"* in lambda DCS:

$$\texttt{Type.USState} \sqcap \neg\texttt{Border.California} \tag{18}$$

and in lambda calculus:

$$\lambda x.\texttt{Type}(x, \texttt{USState}) \wedge \neg\texttt{Border}(x, \texttt{California}). \tag{19}$$

In general, for a unary u,

$$[\![\neg u]\!] = \lambda x.\neg[\![u]\!](x). \tag{20}$$

Higher-order functions Higher-order functions operate on sets of entities rather than on individual entities. These include aggregation (counting, finding the min/max), superlatives (taking the argmin/argmax), and generalized quantification. In lambda DCS, these are implemented in the natural way.

For example, the number of states and the number largest state by area are represented as follows in lambda DCS:

$$\texttt{count}(\texttt{Type.USState}), \tag{21}$$

$$\texttt{argmax}(\texttt{Type.USState}, \texttt{Area}). \tag{22}$$

In lambda calculus:

$$\texttt{count}(\lambda x.\texttt{Type}(x, \texttt{USState})), \tag{23}$$

$$\texttt{argmax}(\lambda x.\texttt{Type}(x, \texttt{USState}), \lambda x.\lambda y.\texttt{Area}(x, y)). \tag{24}$$

In general, for an aggregation operator A (e.g., $\texttt{count}$) and a unary u, we have:

$$[\![A(u)]\!] = \lambda x.[x = A([\![u]\!])]. \tag{25}$$

For a superlative operator S (e.g., $\texttt{argmax}$), unary u, and binary b:

$$[\![S(u, b)]\!] = \lambda x.[x \in S([\![u]\!], [\![b]\!])]. \tag{26}$$

Lambda abstraction Up until now, our lambda DCS logical forms have been limited in two ways:

1. Fundamentally, our logical forms have been tree-structured, since the only way different parts of the logical forms can currently interact is via one implicitly represented overlapping variable. Non-tree-structured logical forms are important for handling bound anaphora, and to capture such phenomena, we will introduce a construct called *mu abstraction*.

2. All our composition operations have created unaries. However, higher-order operations such as $\texttt{argmax}$ take in a binary which supply the comparison function, which could be non-atomic. To construct binaries compositionally, we introduce lambda abstraction, which is in general distinct from lambda abstraction in lambda calculus, although the two coincide in some cases.

Let us start with an example of bound anaphora: *"those who had a child who influenced them"*. In lambda DCS, we use mu abstraction:

$$\mu x.\texttt{Children}.\texttt{Influenced}.x \tag{27}$$

Here, the μx simply adds a constraint that the first argument of $\texttt{Children}$ be bound to the second argument of $\texttt{Influenced}$. In lambda calculus, this expression is:

$$\lambda x.\exists y.\texttt{Children}(x, y).\texttt{Influenced}(y, x). \tag{28}$$

Let us consider a superlative with a compositional comparison function: *"person who has the most children"*. In lambda DCS, we use lambda abstraction to construct a new binary logical form denoting pairs of people and the number of children they have:

$$\texttt{argmax}(\texttt{Type.Person}, \mathbf{R}[\lambda x.\texttt{count}(\mathbf{R}[\texttt{Children}].x)]), \tag{29}$$

where we use the *reverse operator* $[\![\mathbf{R}[b]]\!] = \lambda y.\lambda x.[\![b]\!](x, y)$ switches the arguments of b.

The equivalent logical form in lambda calculus:

$$\texttt{argmax}(\lambda x.\texttt{Type}(x, \texttt{Person}), \lambda x.\texttt{count}(\lambda z.\texttt{Children}(x, z))). \tag{30}$$

In general, for a variable $a \in V$ and a unary u, we define the conversion for variables, mu abstraction, and lambda abstraction as follows:

$$[\![a]\!] = \lambda x.[x = a], \tag{31}$$
$$[\![\mu a.u]\!] = \lambda x.[a = x] \wedge [\![u]\!](x), \tag{32}$$
$$[\![\lambda a.u]\!] = \lambda x.\lambda a.[\![u]\!](x). \tag{33}$$

Note that applied to unaries, lambda abstraction produces binary logical forms, while mu abstraction produces unary logical forms.

2.1 Discussion

A prevailing theme in lambda DCS is that the construction of logical forms centers around *sets* (of entities or entity pairs), whereas in lambda calculus, construction centers around truth values. Lifting up to sets allows us to eliminate variables, treating them as implicitly existentially quantified. Indeed, looking back to the conversion function $[\![\cdot]\!]$, they all have the form $[\![z]\!] = \lambda x.\text{something}$. Of course, there are logical forms where lambda DCS would not offer much savings in reducing variables—for example, when many predicates operate on the same variables in a non-tree-structured manner. However, we have found that for logical forms derived from natural language, especially for querying databases, lambda DCS is a good choice.

The development of lambda DCS has been inspired by many other formalisms. As evident by name, lambda DCS is adapted from Dependency-Based Compositional Semantics (DCS) (Liang et al. 2011), which prominently features (i) tree-structured logical forms and (ii) default existential quantification, the latter being imported from Discourse Representation Theory (Kamp and Reyle 1993). There are two major differences between lambda DCS and DCS. First, lambda DCS revolves around unaries and binaries, which we found to be the right level of generality; DCS allowed arbitrary arities, and thus was a little too low-level. Second, we have omitted the mark-execute construct of

DCS, which is an orthogonal direction for future exploration. We can think of mark-execute as more in the realm of constructing logical forms compositionally rather than their representation. We have focused on the latter here.

The syntax of lambda DCS and the tree-structured nature (without variables) is derived from the concept constructors in description logic (Baader 2003), but the marriage with variables and higher-order functions is specific to lambda DCS. Of course, description logic is designed for logical inference, which necessitates a simpler logic, whereas lambda DCS is designed for model checking (e.g., querying a database), and can therefore bear the additional complexity.

Given our application to querying graph databases, it is natural that there are some parallels between lambda DCS and graph query languages such as SPARQL (Harris and Seaborne 2011), to which we ultimately convert in order to execute the logical forms (Berant et al. 2013). Working with a graph database leads to thinking about logical forms as graph patterns, although this intuition is less helpful when we add higher-order operations.

In terms of expressivity, lambda DCS is clearly dominated by lambda calculus, as evident by our conversion from the former to the latter. From this perspective, lambda DCS is syntactic sugar. However, we believe that lambda DCS is more closely connected with the compositional structure of natural language and therefore can be quite convenient to work with.

We should also note that lambda calculus in formal semantics is really used for two purposes, which we'd like to dissect: The first purpose is to construct the logical form of a sentence compositionally, where lambda abstraction is merely used as a macro to piece bits of logical form together. The second purpose is to represent the final logical form, where lambda abstraction is needed to denote a function (e.g., the comparison function passed into `argmax`). Here, we have mainly focused on the latter representation issues. As for construction, we have so far used a very simple construction mechanism Berant et al. (2013), since we targeted the semantic parsing of short questions, which have limited compositional demands. In principle, lambda DCS logical forms could be constructed using lambda calculus (serving as a macro) in a CCG formalism. Analogously, lambda calculus has been used to build logical forms in Discourse Representation Theory (Muskens 1996). Down the road, we hope that working with lambda DCS can lead to new construction mechanisms suitable for complex sentences.

References

F. Baader. *The description logic handbook: theory, implementation, and applications.* Cambridge University Press, 2003.

J. Berant, A. Chou, R. Frostig, and P. Liang. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *International Conference on Management of Data (SIGMOD)*, pages 1247–1250, 2008.

B. Carpenter. *Type-Logical Semantics.* MIT Press, 1998.

S. Harris and A. Seaborne. SPARQL 1.1 query language. In *W3C Working Draft, 12 May*, 2011.

H. Kamp and U. Reyle. *From Discourse to Logic: An Introduction to the Model-theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory.* Kluwer, Dordrecht, 1993.

P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*, pages 590–599, 2011.

R. Muskens. Combining montague semantics and discourse representation. *Linguistics and Philosophy*, 19(2):143–186, 1996.