Type-Driven Incremental Semantic Parsing with Polymorphism*

Kai Zhao
Graduate Center
City University of New York
kzhao.hf@gmail.com

Liang Huang
Queens College and Graduate Center
City University of New York
liang.huang.sh@gmail.com

Abstract

Semantic parsing has made significant progress, but most current semantic parsers are extremely slow (CKY-based) and rather primitive in representation. We introduce three new techniques to tackle these problems. First, we design the first linear-time incremental shift-reduce-style semantic parsing algorithm which is more efficient than conventional cubic-time bottom-up semantic parsers. Second, our parser, being type-driven instead of syntax-driven, uses type-checking to decide the direction of reduction, which eliminates the need for a syntactic grammar such as CCG. Third, to fully exploit the power of type-driven semantic parsing beyond simple types (such as entities and truth values), we borrow from programming language theory the concepts of subtype polymorphism and parametric polymorphism to enrich the type system in order to better guide the parsing. Our system learns very accurate parses in GEOQUERY, JOBS and ATIS domains.

1 Introduction

Most existing semantic parsing efforts employ a CKY-style bottom-up parsing strategy to generate a meaning representation in simply typed lambda calculus (Zettlemoyer and Collins, 2005; Lu and Ng, 2011) or its variants (Wong and Mooney, 2007; Liang et al., 2011). Although these works led to fairly accurate semantic parsers, there are two major drawbacks: efficiency and expressiveness.

First, as many researches in syntactic parsing (Nivre, 2008; Zhang and Clark, 2011) have shown, compared to cubic-time CKY-style parsing, incremental parsing can achieve comparable accuracies while being linear-time, and orders of magnitude faster in practice. We therefore introduce the first incremental parsing algorithm for semantic parsing. More interestingly, unlike syntactic parsing, our incremental semantic parsing algorithm, being strictly **type-driven**, directly employs type checking to automatically determine the direction of function application on-the-fly, thus reducing the search space and eliminating the need for a syntactic grammar such as CCG to explicitly encode the direction of function application.

However, to fully exploit the power of type-driven incremental parsing, we need a more sophisticated type system than simply typed lambda calculus. Compare the following two phrases:

(1) the governor of New York

(2) the mayor of New York

If we know that *governor* is a function from state to person, then the first *New York* can only be of type *state*; similarly knowing *mayor* maps city to person disambiguates the second *New York* to be of type *city*. This can not be done using a simple type system with just entities and booleans.

Now let us consider a more complex question which will be our running example in this paper:

(3) What is the capital of the largest state by area?

Since we know *capital* takes a state as input, we expect *the largest state by area* to return a state. But does *largest* always return a state type? Notice that it is polymorphic, for example, *largest city by population*, or *largest lake by perimeter*. So there is no unique type for *largest*: its return type should depend on the type of its first argument (*city*, *state*, or *lake*). This observation motivates us to introduce the powerful mechanism of parametric polymorphism from programming languages into natural language semantics for the first time.

For example, we can define the type of *largest* to be a template

$$\textbf{largest} : (\texttt{'a}\to\texttt{t})\to(\texttt{'a}\to\texttt{i})\to\texttt{'a}$$

where `'a` is a *type variable* that can match any type (for formal details see §3). Just like in functional programming languages such as ML or Haskell, type variables can be bound to a real type (or a range of types) during function application, using the technique of type inference. In the above example, when *largest* is applied to *city*, we know that type variable `'a` is bound to type city (or its subtype), so *largest* would eventually return a city.

We make the following contributions:

- We design the first linear-time incremental semantic parsing algorithm (§2), which is much more efficient than the existing semantic parsers that are cubic-time and CKY-based.

*We thank the reviewers for helpful suggestions. We are also grateful to Luke Zettelmoyer, Yoav Artzi, and Tom Kwiatkowski for providing data. This research is supported by DARPA FA8750-13-2-0041 (DEFT), NSF IIS-1449278, and a Google Faculty Research Award.

- In line with classical Montague theory (Heim and Kratzer, 1998), our parser is type-driven instead of syntax-driven as in CCG-based efforts (Zettlemoyer and Collins, 2005; Kwiatkowski et al., 2011; Krishnamurthy and Mitchell, 2014) (§2.3).
- We introduce parametric polymorphism into natural language semantics (§3), along with proper treatment of subtype polymorphism, and implement Hindley-Milner style type inference (Pierce, 2005, Chap. 10) during parsing (§3.3).[1]
- We adapt the latent-variable max-violation perceptron training from machine translation (Yu et al., 2013), which is a perfect fit for semantic parsing due to its huge search space (§4).

2 Type-Driven Incremental Parsing

We start with the simplest meaning representation (MR), *untyped lambda calculus*, and introduce typing and the incremental parsing algorithm for it. Later in §3, we add subtyping and type polymorphism to enrich the system.

2.1 Meaning Representation with Types

The untyped MR for the running example is:

Q: What is the capital of the largest state by area?

MR: (**capital** (**argmax** **state** **size**))

Note the binary function **argmax**$(\cdot,\cdot)$ is a higher-order function that takes two other functions as input: the first argument is a "domain" function that defines the set to search for, and second argument is an "evaluation" function that returns a integer for an element in that domain.

The simply typed lambda calculus (Heim and Kratzer, 1998; Lu and Ng, 2011) augments the system with types, including base types (entities e, truth values t, or numbers i), and function types (e.g., $\mathsf{e}\to\mathsf{t}$). So **capital** is of type $\mathsf{e}\to\mathsf{e}$, **state** is of type $\mathsf{e}\to\mathsf{t}$, and **size** is of type $\mathsf{e}\to\mathsf{i}$. The **argmax** function is of type $(\mathsf{e}\to\mathsf{t})\to(\mathsf{e}\to\mathsf{i})\to\mathsf{e}$.[2] The simply typed MR is now written as

$$(\textbf{capital}:\mathsf{e}\to\mathsf{e}\quad(\textbf{argmax}:(\mathsf{e}\to\mathsf{t})\to(\mathsf{e}\to\mathsf{i})\to\mathsf{e}\quad\textbf{state}:\mathsf{e}\to\mathsf{t}\quad\textbf{size}:\mathsf{e}\to\mathsf{i}))).$$

2.2 Incremental Semantic Parsing: An Example

Similar to a standard shift-reduce parser, we maintain a *stack* and a *queue*. The queue contains words to be parsed, while the stack contains subexpressions of the final MR, each of which is a valid typed lambda expression. At each step, the parser choose to **shift** or **reduce**, but unlike standard shift-reduce parser, there is also a third possible action, **skip**, skipping a semantically vacuous word (e.g., "the", "of", "is", etc.). For example, the first three words of the example question *"What is the ..."* are all skipped (**steps 1–3** in Figure 1 (left)).

pattern	λ-expression templates, simple types (§2.2)
JJS	$\lambda P:(\mathsf{e}\to\mathsf{t})\to(\mathsf{e}\to\mathsf{i})\to\mathsf{e}\ .\ P$
NN	$\lambda P:\mathsf{e}\to\mathsf{e}\ .\ P;\quad \lambda P:\mathsf{e}\to\mathsf{t}\ .\ P;\quad \lambda P:\mathsf{e}\to\mathsf{i}\ .\ P$
pattern	λ-expression templates, polymorphic types (§3.3)
JJS	$\lambda P:(\mathsf{'a}\to\mathsf{t})\to(\mathsf{'a}\to\mathsf{i})\to\mathsf{'a}\ .\ P$
NN	$\lambda P:\mathsf{'b}\to\mathsf{'c}\ .\ P$

Table 1: POS-based meaning representation templates used in the running example (see Figure 1). The polymorphic types greatly simplifies the representation for common nouns (NN).

The parser then **shifts** the next word, *"capital"*, from the queue to the stack. But unlike incremental syntactic parsing where the word itself is moved onto the stack, here we need to find a **grounded** predicate in the GeoQuery domain for the current word. Triggered by the POS tag NN of word "capital", the template $\lambda P:\mathsf{e}\to\mathsf{e}\ .\ P$ is fetched from a predefined MR templates set like Table 1. In its outermost lambda abstraction, variable P needs to be grounded on-the-fly before we push the expression onto the stack. We find a predicate **capital**$:\mathsf{e}\to\mathsf{e}$ in the GEOQUERY domain applicable to the MR template. After the application, we push the result onto the stack (**step 4**).

Next, words *"of the"* are skipped (steps 5–6). For the next word *"largest"*, **argmax**$:(\mathsf{e}\to\mathsf{t})\to(\mathsf{e}\to\mathsf{i})\to\mathsf{e}$ is applied to the MR template triggered by its POS tag JJS in Table 1, and the stack becomes (**step 7**)

$$\textbf{capital}:\mathsf{e}\to\mathsf{e}\quad\textbf{argmax}:(\mathsf{e}\to\mathsf{t})\to(\mathsf{e}\to\mathsf{i})\to\mathsf{e}.$$

At this step we have two expressions on the stack and we could attempt to reduce. But type checking fails because for left reduce, **argmax** expects an argument (its "domain" function) of type $(\mathsf{e}\to\mathsf{t})$ which is different from **capital**'s type $(\mathsf{e}\to\mathsf{e})$, so is the case for right reduce. So we have to shift again. This time for word *"state"*: **state**$:\mathsf{e}\to\mathsf{t}$. The stack becomes:

$$\textbf{capital}:\mathsf{e}\to\mathsf{e}\quad\textbf{argmax}:(\mathsf{e}\to\mathsf{t})\to(\mathsf{e}\to\mathsf{i})\to\mathsf{e}\quad\textbf{state}:\mathsf{e}\to\mathsf{t}.$$

2.3 Type-Driven Reduce

At this step we can finally perform a **reduce** action, since the top two expressions on the stack pass the type-checking for rightward function application (a partial application): **argmax** expects an $(\mathsf{e}\to\mathsf{t})$ argument, which is exactly the type of **state**. So we conduct a right-reduce, applying **argmax** on **state**, which results in

$$(\textbf{argmax}\quad\textbf{state}):(\mathsf{e}\to\mathsf{i})\to\mathsf{e}$$

[1] There are three kinds of polymorphisms in programming languages: parametric (e.g., C++ templates), subtyping, and ad-hoc (e.g., operator overloading). See Pierce (2002, Chap. 15) for details.

[2] Note that the type notation is always *curried*, i.e., we represent a binary function as a unary function that returns another unary function. Also the type notation is always *right-associative*, so $(\mathsf{e}\to\mathsf{t})\to((\mathsf{e}\to\mathsf{i})\to\mathsf{e})$ is also written as $(\mathsf{e}\to\mathsf{t})\to(\mathsf{e}\to\mathsf{i})\to\mathsf{e}$.

step	action	stack after action (simple type)	stack after action (subtyping+polymorphism)
0	-	ϕ	ϕ
1–3	skip	ϕ	ϕ
4	$\mathsf{sh}_{\mathsf{capital}}$	**capital**:e→e	**capital**:st→ct
7	$\mathsf{sh}_{\mathsf{largest}}$	**capital**:e→e **argmax**:(e→t)→(e→i)→e	**capital**:st→ct **argmax**:('a→t)→('a→i)→'a
8	$\mathsf{sh}_{\mathsf{state}}$	**capital**:e→e **argmax**:(e→t)→(e→i)→e **state**:e→t	**capital**:st→ct **argmax**:('a→t)→('a→i)→'a **state**:st→t
9	$\mathsf{re}_{\curvearrowright}$	**capital**:e→e (**argmax state**):(e→i)→e	**capital**:st→ct (**argmax state**):(st→i)→st †
11	$\mathsf{sh}_{\mathsf{area}}$	**capital**:e→e (**argmax state**):(e→i)→e **size**:e→i	**capital**:st→ct (**argmax state**):(st→i)→st **size**:lo→i
12	$\mathsf{re}_{\curvearrowright}$	**capital**:e→e (**argmax state size**):e	**capital**:st→ct (**argmax state size**):st ‡
13	$\mathsf{re}_{\curvearrowright}$	(**capital** (**argmax state size**)):e	(**capital** (**argmax state size**)):ct

Figure 1: Type-driven Incremental Semantic Parsing (TISP) with (a) simple types and (b) subtyping+polymorphism on the example question: "what is the capital of the largest state by area?". Steps 5–6 and 10 are skip actions and thus omitted. The stack and queue in each row are the results *after* each action. †: Type variable 'a is binded to st. ‡: From Eq. 4, st <: lo ⇒ (lo→i) <: (st→i).

while the stack becomes (**step 9**)

capital:e→e (**argmax state**):(e→i)→e

Now if we want to continue reduction, it does not type check for either left or right reduction, so we have to shift again. So we move on to shift the final word *"area"* with predicate: **size**:e→i and the stack becomes (**step 11**):

capital:e→e (**argmax state**):(e→i)→e **size**:e→i.

Now we can do two consecutive right reduces supported by type checking (**step 12, 13**) and get the final result:

(**capital** (**argmax state size**)):e.

Here we can see the novelty of our shift-reduce parser: its decisions are largely driven by the type system. When we attempt a reduce, **at most** one of the two reduce actions (left, right) is possible thanks to type checking, and when neither is allowed, we have to shift (or skip). This observation suggests that our incremental parser is more deterministic than those syntactic incremental parsers where each step always faces a three-way decision (shift, left-reduce, right-reduce). We also note that this type-checking mechanism, inspired by the classical type-driven theory in linguistics (Heim and Kratzer, 1998), eliminates the need for an *explicit* encoding of direction as in CCG, which makes our formalism much simpler than the synchronous syntactic-semantic ones in most other semantic parsing efforts (Zettlemoyer and Collins, 2005; Zettlemoyer and Collins, 2007; Wong and Mooney, 2007).[3]

3 Subtype and Parametric Polymorphisms

Currently in simply typed lambda calculus representation function **capital** can apply to any entity type, for example **capital**(**boston**), which should have been disallowed by the type checker. So we need a more sophisticated system that helps ground with refined types, which will in turn help type-driven parsing.

[3]We need to distinguish between two concepts: a) "direction of reduction": $f(g)$ or $g(f)$. Obviously at any given time, between the top two (unarized) functions f and g on the stack, at most one reduction is possible. b) "order of arguments": $f(x, y)$ or $f(y, x)$. For predicates such as **loc** : lo→lo→t the order does matter. Our parser can not distinguish this purely via types, nor can CCG via its syntactic categories. In practice, it is decided by features such as the voice of the verb.

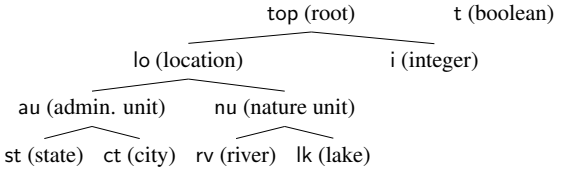

Figure 2: Type hierarchy for GEOQUERY (slightly simplified).

3.1 Semantics with Subtype Polymorphism

We first augment the meaning representation with a domain specific type hierarchy. For example Figure 2 shows a (slightly simplified) version of the type hierarchy for GEOQUERY domain. We use <: to denote the (transitive, reflexive, and antisymmetric) **subtyping relation** between types; for example in GEOQUERY, st <: lo.

Each constant in the GEOQUERY domain is well typed. For example, there are states (**michigan**:st), cities (**nyc**:ct), rivers (**mississippi**:rv), and lakes (**tahoe**:lk).

Similarly each predicate is also typed. For example, we can query the length of a river, **len**:rv→i, or the population of some administrative unit, **population**:au→i. Notice that **population**$(\cdot)$ can be applied to both states and cities, since they are subtypes of administrative unit, i.e., st <: au and ct <: au. This is because, as in Java and C++, a function that expects a certain type can always take an argument of a subtype. For example, we can query whether two locations are adjacent, using **next_to**:lo→(lo→t), as the **next_to**$(\cdot, \cdot)$ function can be applied to two states, or to a river and a city, etc.

Before we move on, there is an important consequence of polymorphism worth mentioning here. For the types of unary predicates such as **city**$(\cdot)$ and **state**$(\cdot)$ that *characterize* its argument, we define theirs argument types to be the required type, i.e., **city** : ct→t, and **state** : st→t. This might look a little weird since everything in the domain of those functions are always mapped to true; i.e., $f(x)$ is either undefined or true, and never false for such f's. This is different from classical simply-typed Montague semantics (Heim and Kratzer, 1998) which defines such predicates as type top→t so that **city**(**mississippi** : st) returns false. The reason for our design is, again, due to

subtyping and polymorphism: **capital** takes a state type as input, so **argmax** must returns a state, and therefore its first argument, the **state** function, must have type st→t so that the matched type variable 'a will be bound to st. This more refined design will also help prune unnecessary argument matching using type checking.

3.2 Semantics with Parametric Polymorphism

The above type system works smoothly for first-order functions (i.e., predicates taking atomic type arguments), but the situation with higher-order functions (i.e., predicates that take functions as input) is more involved. What is the type of **argmax** in the context "the capital of largest state ..."? One possibility is to define it to be as general as possible, as in the simply typed version (and many conventional semantic parsers):

$$\textbf{argmax}:(\mathsf{top}\to\mathsf{t})\to(\mathsf{top}\to\mathsf{i})\to\mathsf{top}.$$

But this actually no longer works for our sophisticated type system for the following reason.

Intuitively, remember that **capital**:st→ct is a function that takes a state as input, so the return type of **argmax** must be a state or its subtype, rather than top which is a supertype of st. But we can not simply replace top by st, since **argmax** can also be applied in other scenarios such as "the largest city". In other words, **argmax** is a *polymorphic* function, and to assign a correct type for it we have to introduce *type variables*:

$$\textbf{argmax}:(\,\mathtt{'a}\to\mathsf{t})\to(\,\mathtt{'a}\to\mathsf{i})\to\mathtt{'a},$$

where type variable 'a is a place-holder for "any type".

3.3 Parsing with Subtype Polymorphism and Parametric Polymorphism

We modify the previous parsing algorithm to accommodate subtyping and polymorphic types. Figure 1 (right) shows the derivation of the running example using the new parsing algorithm. Below we focus on the differences brought by the new algorithm.

Note that we also modified the MR templates as in Table 1. The new MR templates are more general due to the polymorphism from type variables. For example, now we use only one MR template $\lambda P:\mathtt{'b}\to\mathtt{'c}\,.\,P$ to replace the three NN MR templates for simple types.

In step 4, unlike **capital** : e→e, we shift the predicate **capital** : st→ct; in step 7, we shift the polymorphic expression for *"largest"*: **argmax**:('a→t)→('a→i)→ 'a. And after the shift in step 8, the stack becomes

capital:st→ct **argmax**:('a→t)→('a→i)→ 'a **state**:st→t

At **step 9**, in order to apply **argmax** onto **state**:st→t, we simply **bind** type variable 'a to type st, results in (**argmax state**) : (st→i)→st.

After the shift in **step 11**, the stack becomes:

capital:st→ct (**argmax state**):(st→i)→st **size**:lo→i.

Can we still apply right reduce here? According to subtyping requirement (§3.1), we want lo→i <: st→i to hold, knowing that st <: lo. Luckily, there is a rule about function types in type theory that exactly fits here:

$$\frac{\mathsf{A}<:\mathsf{B}}{\mathsf{B}\to\mathsf{C}<:\mathsf{A}\to\mathsf{C}} \tag{4}$$

which states the input side is reversed (contravariant). This might look counterintuitive, but the intuition is that, it is safe to allow the function **size**:lo→i to be used in the context where another type st→i is expected, since in that context the argument passed to **size** will be state type (st), which is a subtype of location type (lo) that **size** expects, which in turn will not surprise **size**. See the classical type theory textbook (Pierce, 2002, Chap. 15.2) for details.

Several works in literature (Zettlemoyer and Collins, 2005; Zettlemoyer and Collins, 2007; Wong and Mooney, 2007; Kwiatkowski et al., 2013) employ some primitive type hierarchies and parse with typed lambda calculus. However, simply introducing subtyped predicates without polymorphism will cause type checking failures in handling high-order functions, as we discussed above.

4 Training: Latent Variable Perceptron

We follow the latent variable max-violation perceptron algorithm of Yu et al. (2013) for training. This algorithm is based on the "violation-fixing" framework of Huang et al. (2012) which is tailored to structured learning problems with abundant search errors such as parsing or machine translation.

The key challenge in the training is that, for each question, there might be many different unknown derivations that lead to its annotated MR, which is known as the *spurious ambiguity*. In our task, the spurious ambiguity is caused by how the MR templates are chosen and grounded during the shift step, and the different reduce orders that lead to the same result. We treat this unknown information as latent variable.

More formally, we denote $D(x)$ to be the set of *all* partial and full parsing derivations for an input sentence x, and $mr(d)$ to be the MR yielded by a full derivation d. Then we define the sets of (partial and full) reference derivations as:

$$good_i(x,y) \triangleq \{d \in D(x) \mid |d| = i, \exists \text{full derivation } d' \text{ s.t. } d \text{ is a prefix of } d', mr(d') = y\},$$

Those "bad" partial and full derivations that do not lead to the annotated MR can be defined as:

$$bad_i(x,y) \triangleq \{d \in D(x) \mid d \notin good_i(x,y), |d| = i\}.$$

At step i, the best reference partial derivation is

$$d_i^+(x,y) \triangleq \underset{d \in good_i(x,y)}{\textbf{argmax}}\ \mathbf{w}\cdot\mathbf{\Phi}(x,d), \tag{5}$$

System	GEOQUERY			JOBS			ATIS		
	P	R	F1	P	R	F1	P	R	F1
Z&C'05	**96.3**	79.3	87.0	**97.3**	79.3	87.4	-	-	-
Z&C'07	91.6	86.1	88.8	-	-	-	**85.8**	**84.6**	**85.2**
UBL	94.1	85.0	89.3	-	-	-	72.1	71.4	71.7
FUBL	88.6	88.6	88.6	-	-	-	82.8	82.8	82.8
TISP (st)	89.7	86.8	88.2	76.4	76.4	76.4	-	-	-
TISP	92.9	**88.9**	**90.9**	85.0	**85.0**	**85.0**	84.7	84.2	84.4

Table 2: Performances (precision, recall, and F1) of various parsing algorithms on GEOQUERY, JOBS, and ATIS datasets. TISP with simple types are marked "st".

while the Viterbi partial derivation is

$$d_i^-(x,y) \triangleq \operatorname*{\mathbf{argmax}}_{d \in bad_i(x,y)} \mathbf{w} \cdot \mathbf{\Phi}(x,d), \quad (6)$$

where $\mathbf{\Phi}(x,d)$ is the defined feature set for derivation d. In practice, to compute Eq. 6 exactly is intractable, and we resort to beam search. Following Yu et al. (2013), we then find the step i^* with the maximal score difference between the best reference partial derivation and the Viterbi partial derivation:

$$i^* \triangleq \operatorname*{\mathbf{argmax}}_i \mathbf{w} \cdot \Delta\mathbf{\Phi}(x, d_i^+(x,y), d_i^-(x,y)),$$

and do update $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{\Phi}(x, d_{i^*}^+(x,y), d_{i^*}^-(x,y))$ where $\Delta\mathbf{\Phi}(x,d,d') \triangleq \mathbf{\Phi}(x,d) - \mathbf{\Phi}(x,d')$.

We also use minibatch parallelization of Zhao and Huang (2013); in practice we use 24 cores.

5 Experiments

We implement our type-driven incremental semantic parser (TISP) using `Python`, and evaluate its performance on GEOQUERY, JOBS, and ATIS datasets.

Our feature design is inspired by the very effective Word-Edge features in syntactic parsing (Charniak and Johnson, 2005) and MT (He et al., 2008). From each parsing state, we collect atomic features including the types and the leftmost and rightmost words of the span of the top 3 MR expressions on the stack, the top 3 words on the queue, the grounded predicate names and the ID of the MR template used in the shift action. We use budget scheme similar to (Yu et al., 2013) to alleviate the overfitting problem caused by feature sparsity. We get 84 combined feature templates in total. Our final system contains 62 MR expression templates, of which 33 are triggered by POS tags, and 29 are triggered by specific phrases.

In the experiments, we use the same training, development, and testing data splits as Zettlemoyer and Collins (2005) and Zettlemoyer and Collins (2007).

For evaluation, we follow Zettlemoyer and Collins (2005) to use *precision* and *recall*:

$$\text{Precision} = \frac{\#\text{ of correctly parsed questions}}{\#\text{ of successfully parsed questions}},$$

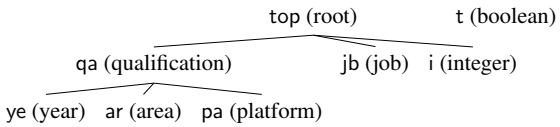

Figure 3: Type hierarchy for JOBS domain (slightly simplified).

$$\text{Recall} = \frac{\#\text{ of correctly parsed questions}}{\#\text{ of questions}}.$$

5.1 Evaluation on GEOQUERY Dataset

We first evaluate TISP on GEOQUERY dataset.

In the training and evaluating time, we use a very small beam size of 16, which gives us very fast decoding. In serial mode, our parser takes $\sim$ 83s to decode the 280 sentences (2,147 words) in the testing set, which means $\sim$0.3s per sentence, or $\sim$0.04s per word.

We compare the our accuracy performance with existing methods (Zettlemoyer and Collins, 2005; Zettlemoyer and Collins, 2007; Kwiatkowski et al., 2010; Kwiatkowski et al., 2011) in Table 2. Given that all other methods use CKY-style parsing, our method is well balanced between accuracy and speed.

In addition, to unveil the helpfulness of our type system, we train a parser with only simple types. (Table 2) In this setting, the predicates only have primitive types of location lo, integer i, and boolean t, while the constants still keep their types. It still has the type system, but it is weaker than the polymorphic one. Its accuracy is lower than the standard one, mostly caused by that the type system can not help pruning the wrong applications like (**population**:au→i **mississippi**:rv).

5.2 Evaluations on JOBS and ATIS Datasets

We also evaluate the performance of our parser on JOBS and ATIS datasets. Figure 3 shows the type hierarchy for JOBS. We omit the type hierarchy for ATIS due to space constraint. Note that ATIS contains more than 5,000 examples and is a lot larger than GEOQUERY and JOBS.

We show the results in Table 2. In JOBS, we achieves very good recall, but the precision is not as good as Zettlemoyer and Collins (2005), which is actually because we parsed a lot more sentences. Also, TISP with simple types is still weaker than the one with subtyping and parametric polymorphisms. For ATIS, our performance is very close to the state-of-the-art.

6 Conclusion

We have presented an incremental semantic parser that is guided by a powerful type system of subtyping and polymorphism. This polymorphism greatly reduced the number of templates and effectively pruned search space during the parsing. Our parser is competitive with state-of-the-art accuracies, but, being linear-time, is faster than CKY-based parsers in theory and in practice.

References

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of ACL*, pages 173–180, Ann Arbor, Michigan, June.

Zhongjun He, Qun Liu, and Shouxun Lin. 2008. Improving statistical machine translation using lexicalized rule selection. In *Proceedings of COLING*, pages 321–328, Manchester, UK, August.

Irene Heim and Angelika Kratzer. 1998. *Semantics in Generative Grammar*. Blackwell Publishing.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proceedings of NAACL*.

Jayant Krishnamurthy and Tom M Mitchell. 2014. Joint syntactic and semantic parsing with combinatory categorial grammar.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic ccg grammars from logical form with higher-order unification. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 1223–1233. Association for Computational Linguistics.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in ccg grammar induction for semantic parsing. In *Proceedings of EMNLP*, EMNLP '11.

Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching.

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

Wei Lu and Hwee Tou Ng. 2011. A probabilistic forest-to-string model for language generation from typed lambda calculus expressions. In *Proceedings of EMNLP*.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Benjamin Pierce, editor. 2005. *Advanced Topics in Types and Programming Languages*. MIT Press.

Yuk Wah Wong and Raymond J Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Annual Meeting-Association for computational Linguistics*, volume 45, page 960.

Heng Yu, Liang Huang, Haitao Mi, and Kai Zhao. 2013. Max-violation perceptron and forced decoding for scalable mt training. In *Proceedings of EMNLP 2013*.

Luke Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of UAI*.

Luke S Zettlemoyer and Michael Collins. 2007. Online learning of relaxed ccg grammars for parsing to logical form. In *In Proceedings of EMNLP-CoNLL-2007*. Citeseer.

Yue Zhang and Stephen Clark. 2011. Shift-reduce ccg parsing. In *Proceedings of ACL*.

Kai Zhao and Liang Huang. 2013. Minibatch and parallelization for online large margin structured learning. In *Proceedings of NAACL 2013*.