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

\[ \text{largest} : ('a \rightarrow t) \rightarrow ('a \rightarrow i) \rightarrow 'a \]

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).

• 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(·, ·) 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., e→t). So capital is of type e→e, state is of type e→t, and size is of type e→i. The argmax function is of type (e→t)→(e→i)→e. The simply typed MR is now written as

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

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 λP: e→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: e→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: (e→t)→(e→i)→e is applied to the MR template triggered by its POS tag JJS in Table 1, and the stack becomes (step 7)

capital: e→e argmax: (e→t)→(e→i)→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 (e→t) which is different from capital’s type (e→e), so is the case for right reduce. So we have to shift again. This time for word “state”: state: e→t. The stack becomes:

capital: e→e argmax: (e→t)→(e→i)→e state: e→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 (e→t) argument, which is exactly the type of state. So we conduct a right-reduce, applying argmax on state, which results in

(argmax state): (e→i)→e

<table>
<thead>
<tr>
<th>pattern</th>
<th>λ-expression templates, simple types (§2.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JJS</td>
<td>λP: (e→t)→(e→i)→e. P</td>
</tr>
<tr>
<td>NN</td>
<td>λP: e→e. P; λP: e→i. P</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pattern</th>
<th>λ-expression templates, polymorphic types (§3.3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JJS</td>
<td>λP: ((\lambda e \leftarrow t \rightarrow \lambda i \rightarrow e) . P)</td>
</tr>
<tr>
<td>NN</td>
<td>λP: ((\lambda e \leftarrow i \rightarrow c \rightarrow e) . P)</td>
</tr>
</tbody>
</table>
In practice, it is decided by features such as the voice of the verb. After action (simple type)

<table>
<thead>
<tr>
<th>step</th>
<th>action</th>
<th>stack after action (simple type)</th>
<th>stack after action (subtyping+polymorphism)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>skip</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>sh_cap</td>
<td>capital:e→e</td>
<td>capital:st→ct</td>
</tr>
<tr>
<td>7</td>
<td>sh_arg</td>
<td>capital:e→e argmax:e→i→e</td>
<td>capital:ct argmax:st→ct</td>
</tr>
<tr>
<td>8</td>
<td>sh_sta</td>
<td>capital:e→e argmax:e→i→e state:e→t</td>
<td>capital:st→ct argmax:st→ct state:st→t</td>
</tr>
<tr>
<td>9</td>
<td>re_sh</td>
<td>capital:e→e (argmax state):e→i</td>
<td>capital:st→ct (argmax state):st→t</td>
</tr>
<tr>
<td>11</td>
<td>sh_area</td>
<td>capital:e→e (argmax state):e</td>
<td>capital:st→ct (argmax state):st size:lo→i</td>
</tr>
<tr>
<td>12</td>
<td>re_sh</td>
<td>capital:e→e (argmax state size):e</td>
<td>capital:st→ct (argmax state size):st</td>
</tr>
<tr>
<td>13</td>
<td>re_sh</td>
<td>(capital (argmax state size)):e</td>
<td>(capital (argmax state size)):ct</td>
</tr>
</tbody>
</table>

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 's is binded to st. ‡: From Eq. 4, st <: lo ⇒ (lo→i) <: (st→i).

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.

3.1 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(·) 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(·, ·) 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(·) and state(·) 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):

\[ \text{argmax}: (\text{top} \rightarrow t) \rightarrow (\text{top} \rightarrow i) \rightarrow \text{top}. \]

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:

\[ \text{argmax}: (a\rightarrow t)\rightarrow (a\rightarrow i)\rightarrow 'a, \]

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 \( \lambda P : t_0 \rightarrow 'c . P \) 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:

\[ \text{capital}: st \rightarrow ct \hspace{2mm} \text{(argmax state)}: (st \rightarrow i) \rightarrow st \hspace{2mm} \text{size}: lo \rightarrow i. \]

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:

\[
A :<; B \quad \Brightarrow C :<; A \rightarrow C
\]

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 lambda calculi 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:

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

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

\[
d^+_i(x,y) \triangleq \argmax_{d \in good_i(x,y)} w \cdot \Phi(x,d), \tag{5}
\]
While the Viterbi partial derivation is

\[ d_i^-(x, y) = \arg\max_{d \in \text{bad}_i(x, y)} w \cdot \Phi(x, d), \tag{6} \]

where \( \Phi(x, d) \) 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:

\[ i^* = \arg\max_{i} w \cdot \Delta \Phi(x, d_i^+(x, y), d_i^-(x, y)), \]

and do update \( w \leftarrow w + \Delta \Phi(x, d_i^+(x, y), d_i^-(x, y)) \)

where \( \Delta \Phi(x, d, d') = \Phi(x, d) - \Phi(x, d') \).

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:

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

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

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

<table>
<thead>
<tr>
<th>System</th>
<th>GEOQUERY</th>
<th>JOBS</th>
<th>ATIS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>F1</td>
</tr>
<tr>
<td>Z&amp;C’05</td>
<td>96.3</td>
<td>79.3</td>
<td>87.0</td>
</tr>
<tr>
<td>Z&amp;C’07</td>
<td>91.6</td>
<td>86.1</td>
<td>88.8</td>
</tr>
<tr>
<td>UBL</td>
<td>94.1</td>
<td>85.0</td>
<td>89.3</td>
</tr>
<tr>
<td>FUBL</td>
<td>88.6</td>
<td>88.6</td>
<td>88.6</td>
</tr>
<tr>
<td>TISP (st)</td>
<td>89.7</td>
<td>86.8</td>
<td>82.2</td>
</tr>
<tr>
<td>TISP</td>
<td>92.9</td>
<td>89.9</td>
<td>90.9</td>
</tr>
</tbody>
</table>

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 \((\text{population:au} \rightarrow i \text{ mississippi:rv})\).

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


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.


