Type-Driven Incremental Semantic Parsing with Polymorphism

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

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

Abstract

Semantic parsing is a burgeoning field, but most current semantic parsers are extremely slow (CKY-based) and rather primitive in representation (simply typed lambda calculus). We introduce two new techniques to tackle these problems. First, we design a linear-time, type-driven, incremental parsing algorithm that use type checking to reduce the search space, which is orders of magnitude faster than conventional cubic-time bottom-up semantic parsers, and also eliminates the need for a formal grammar such as CCG. Second, to fully exploit the power of type-driven semantic parsing beyond simple types (such as entities and truth values), we introduce a sophisticated subtype hierarchy and parametric polymorphism to the system, so that the type system is powerful enough to better guide the parsing. Together with max-violation perceptron training, 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, Jordan, and Klein 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, which means 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 which explicitly encodes 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. Unfortunately, except for those discussed in Section 3, most current semantic parsing efforts use a primitive type system with a couple of base types, such as entities, truth values, and numbers. Therefore an ambiguous term such as Mississippi is always of type entity, and disambiguation is performed by features rather than by the type system itself. We argue that it is beneficial to incorporate an explicit subtype hierarchy, such that ambiguous terms can be grounded based on context in a more explicit and declarative fashion. Compare the following two questions:

(1) What is the length of Mississippi?
(2) What is the capital of Mississippi?

If we know that length is a function from rivers to integers, then the first Mississippi can only be of type river; similarly knowing capital maps states to cities disambiguates the second Mississippi to be of type state. 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 the type system for natural language. For example, we can define the type of largest to be a template

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

where \text{''a} is a type variable that can match any type (for formal details see Section 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 \text{''a} is bound to type city (or its subtype), so that largest would eventually return a city.

We make the following contributions:

• We design a linear-time incremental semantic parsing algorithm (Section 2), which is much more efficient than the majority of existing semantic parsers that are cubic-time CKY-based.
2 Type-Driven Incremental Parsing

We start with the simplest meaning representation (MR), untyped lambda calculus, and then introduce typing and the incremental parsing algorithm for it. Later in Section 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. In other words

$$\text{argmax}(f, g) = \text{argmax}_{x} g(x).$$

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 function 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

$$(\text{capital}: e\rightarrow e \quad \text{argmax}:(e\rightarrow t)\rightarrow(e\rightarrow i)\rightarrow e \quad \text{state}: e\rightarrow t \quad \text{size}: e\rightarrow i).$$

2.2 Incremental Semantic Parsing: An Example

We use the above running example to explain our type-driven incremental semantic parsing algorithm. Figure 1(a) illustrates the full derivation.

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, where each subexpression 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, which skips 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(a)).

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. In this example we find the predicate:

$$\text{capital}: e\rightarrow e$$

and put it on the stack (step 4).

Next, words “of the” are skipped (steps 5–6). Then for word “largest”, we shift the predicate

$$\text{argmax}:(e\rightarrow t)\rightarrow(e\rightarrow i)\rightarrow e$$

onto the stack (step 7), which becomes

$$\text{capital}: e\rightarrow e \quad \text{argmax}:(e\rightarrow t)\rightarrow(e\rightarrow i)\rightarrow e \quad \text{state}: e\rightarrow t.$$

2.3 Type-Driven Reduce

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” we shift the predicate

$$\text{state}: e\rightarrow t$$

onto the stack, which becomes:

$$\text{capital}: e\rightarrow e \quad \text{argmax}:(e\rightarrow t)\rightarrow(e\rightarrow i)\rightarrow e \quad \text{state}: e\rightarrow t \quad \text{size}: e\rightarrow i.$$

1Note 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 (e→t)→i((e→i)→e) is also written as (e→t)→(e→i)→e.
followed by another, final, right reduce (step 13):

\[
\text{capital \ (argmax\ state\ size)} : e.
\]

We can combine the two expressions using predicate and since their types match, and get

\[
\lambda x : e . (\text{and : } t \rightarrow t \rightarrow (\text{major : } e \rightarrow t \ x) (\text{city : } e \rightarrow t \ x)),
\]

where type \(t \rightarrow t \rightarrow t\) takes two booleans and return one (again, using currying notation).

### 3 Subtyping and Type Polymorphism

As mentioned in Section 1, simply typed lambda calculus representation can not distinguish between Mississippi the river and Mississippi the state since they both have the same type \(e\). Furthermore, currently function capital can apply to any entity type, for example capital(boston), which should have been reported by the type checker. So we need a more sophisticated type system that helps ground terms to real-world entities, and this refined type system will in turn help type-driven parsing.

#### 3.1 Augmenting MR with Subtyping

We first augment the meaning representation with a type hierarchy which is domain specific. For example Figure 2 shows a (slightly simplified) version of the type hierarchy for GEOQUERY domain. Here the root type top has a sub-type of locations, \(lo\), which consists of two different kinds of locations, administrative units (\(au\)) including states (\(st\)) and cities (\(ct\)), and nature units (\(nu\)) including rivers (\(rv\)) and lakes (\(lk\)). We use \(\prec\) to denote the (transitive, reflexive, and antisymmetric) subtyping relation between types; for
They are subtypes of administrative unit, i.e., languages, a function that expects a type variable as input (i.e., \( \text{argmax} \)).

*Note* that the names like *mississippi* appears twice for two different entities. The fact that we can distinguish them by type is a crucial advantage of a typed semantic formalism.

Each constant in the \( \text{GEO} \) domain is well typed. For example, there are states (\( \text{mississippi:st} \)), rivers (\( \text{mississippi:rv} \)), and lakes (\( \text{tahoe:lk} \)).

In addition we have an integer type \( \text{int} \) for any type \( T \).

In step 4, unlike \( \text{capital: e} \rightarrow \text{e} \), we shift the predicate \( \text{capital: st} \rightarrow \text{ct} \) and in step 7, we shift the polymorphic expression for “largest” \( \text{argmax: ('a->t)->('a->i)->'a} \).

And after the shift in step 8, the stack becomes \( \text{capital: st->ct argmax: ('a->t)->('a->i)->'a state: st->t} \).

At step 9, in order to apply \( \text{argmax} \) onto \( \text{state: st->t} \), we simply bind type variable ‘a to type \( \text{st} \), i.e.,

\[ \text{argmax: (st->t)->(st->i)->st state: st->t} \]

results in \( \text{(argmax state): (st->i)->st} \).

After the shift in step 11, the stack becomes \( \text{capital: st->ct argmax state: (st->i)->st size: lo->i} \).

Can we still apply right reduce here? According to the subtyping rule (Eq. 4), we want

\[ \text{lo->i <: st->i} \]
to hold, knowing that \( st \prec \succ lo \). Luckily, there is a rule about function types in type theory that exactly fits here:

\[
A \prec \succ B \quad B \rightarrow C \Rightarrow A \rightarrow C
\]  

(5)

which states the input side is reversed (contravariant). This might look counterintuitive at first glance, but the intuition is that, is safe to allow the function \texttt{size} of type \( lo \rightarrow i \) to be used in the context where another type \( st \rightarrow i \) is expected, since in that context the argument passed to \texttt{size} will be state type \( st \), which is a subtype of location type \( lo \) that \texttt{size} expects, which will not surprise \texttt{size}. See the classical type theory textbook (Pierce 2002, Chap. 15.2) for details. See Figure 1(b) for the full derivation.

4 Training: Latent Variable Perceptron

We follow the Latent Variable Violation-Fixing Perceptron framework (Huang, Fayong, and Guo 2012; Yu et al. 2013) for the training.

4.1 Framework

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 type-driven incremental semantic parsing task, the spurious ambiguity is caused by how the expression 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:

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

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

\[
bad_{i}(x,y) \overset{\Delta}{=} \{ d \in D(x) \mid d \notin good_{i}(x,y), |d| = i \}.
\]

At step \( i \), the best reference partial derivation is

\[
d^+_i(x,y) \overset{\Delta}{=} \arg \max_{d \in good_i(x,y)} w \cdot \Phi(x,d),
\]

while the Viterbi partial derivation is

\[
d^-_i(x,y) \overset{\Delta}{=} \arg \max_{d \in bad_i(x,y)} w \cdot \Phi(x,d),
\]

where \( \Phi(x,d) \) is the defined feature set for derivation \( d \).

In practice, to compute Eq. 7 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:

\[
i^* \overset{\Delta}{=} \arg \max_i w \cdot \Delta \Phi(x,d^+_i(x,y),d^-_i(x,y)),
\]

and do update:

\[
w \leftarrow w + \Delta \Phi(x,d^+_i(x,y),d^-_i(x,y))
\]

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

4.2 Forced Decoding

We use forced decoding to retrieve the reference derivations \( good_i(x,y) \) for each question/MR pair \( (x,y) \) in Eq. 6.

Unlike syntactic incremental parsing, where the forced decoding can be done in polynomial time (Goldberg, Sartorio, and Satta 2014), we do not have an algorithm designed for efficient forced decoding. We apply exponential-time brute-force search to calculate \( good(x,y) \), during which we do pruning based on the predicate application orders.

However, this requires heavy computation we can not afford. In practice we choose multi-pass forced decoding. First we use brute-force search to decode, but with a time limit. Then we train a Perceptron using successfully decoded reference derivations, and use the trained Perceptron to decode the unfinished questions with a large beam. We then add the reference derivations newly discovered into the next step training.

5 Experiments

We implement our type-driven incremental semantic parser (TISP) using \texttt{Python}, and evaluate its performance of both speed and accuracy on \texttt{GeoQUERY} and \texttt{JOBS} datasets.

Our feature design is inspired by the very effective Word-Edge features in syntactic parsing (Charniak and Johnson 2005) and MT (He, Liu, and Lin 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 expression template used in the shift action.

To ease the overfitting problem caused by the feature sparsity, we assign different budgets to different kinds of features and only generate feature combinations within a budget limit. We get 84 combined feature templates in total.

For evaluation, we follow Zettlemoyer and Collins (2005) to use precision and recall. In addition, we calculate the percentage of successfully parsed questions for each method.

5.1 Evaluation on \texttt{GeoQUERY} Dataset

We first evaluate TISP on \texttt{GeoQUERY} dataset.

Following the scheme of Zettlemoyer and Collins (2007), we use the first 600 sentences of Geo880 as the training set and the rest 280 sentences as the testing set.

Note that we do not have a separate development set, due to the relatively small size of Geo880. So to find the best number of iterations to stop the training, we do a 10-fold cross-validation training over the training set, and choose to train 20 iterations and then evaluate.

We use two-pass forced decoding. In the initial brute-force pass we set the time limit to 1,200 seconds, and find the reference derivations for 530 of the total 600 training sentences, a coverage of \( \sim 88\% \). In the second pass we set beam size to 16,384 and get 581 sentences covered (\( \sim 97\% \)).

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 \( \sim 144 \) seconds to decode the 280 sentences (2,147 words) in the testing set, which means \( \sim 0.5 \) second per sentence, or \( \sim 0.07 \) second per word.
Table 1: Performances of various parsing algorithms on GEOQUERY dataset. *: λ-WASP is trained on 792 examples.

| System            | Prec. | Recall | Parsed % |
|-------------------|-------|--------|----------|
| Zettlemoyer’(05)  | 96.3  | 79.3   | 82.3     |
| Zettlemoyer’(07)  | 91.6  | 86.1   | 90.3     |
| UBL               | 94.1  | 85.0   | 89.3     |
| FUBL              | 88.6  | 86.6   | 100.0    |
| TISP (simple types) | 86.3 | 82.9   | 96.1     |
| TISP              | 91.5  | 87.9   | 96.1     |
| λ-WASP*           | 92.0  | 86.6   | 94.1     |

Table 2: Performances on JOBS dataset.

| System            | Prec. | Recall | Parsed % |
|-------------------|-------|--------|----------|
| Zettlemoyer’(05)  | 85.8  | 84.6   | 98.6     |
| UBL               | 72.1  | 71.4   | 99.0     |
| FUBL              | 82.8  | 82.8   | 100.0    |
| TISP              | 83.9  | 82.6   | 98.5     |

Table 3: Performances on ATIS dataset.

We compare the accuracy performance with existing methods in Table 1. 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 1) 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

The JOBS domain contains descriptions about required and desired qualifications of a job. The qualifications include programming language (la), years of experience (ye), diploma degree (de), area of fields (ar), platform (pa), title of the job (ti), etc. We show a simplified version of the type hierarchy for JOBS in Figure 3.

Following the splitting scheme of Zettlemoyer and Collins (2005), we use 500 sentences as training set and 140 sentences as testing set.

Table 2 shows that our algorithm achieves significantly higher recall than existing method of Zettlemoyer and Collins (2005), although our precision is not as high as theirs. This is actually because our method parses a lot more questions in the dataset, as the column of the percentage of successfully parsed sentences suggests.

We also evaluate the performance of TISP on ATIS dataset as in Table 3 ATIS dataset contains more than 5,000 examples and is a lot larger than GEOQUERY and JOBS. Our method achieves comparable performance on this dataset. Due to space constraints, we do not show its type hierarchy here.

6 Related Work

Zettlemoyer and Collins (2005) introduce a type hierarchy to semantic parsing and parse with typed lambda calculus combined with CCG. However, simply introducing subtyped predicates without polymorphism will cause type checking failures in handling high-order functions, as shown in Section 6. Furthermore, our system, being type-driven, almost completely rely on the types of MR expressions to guide parsing (except for some simple POS tag triggers) while their system is heavily CCG-based and syntax-driven.

Kwiatkowski et al. (2013) use “on-the-fly” matching to fetch the most possible predicate in the dataset for some MR subexpression. The matching happens at the end of parsing, and is constrained by the type of the subexpression. We do matching and parsing jointly, both of which are constrained by the typing, and affect the typing, which is more similar to how human do semantic parsing, i.e., we parse part of the sentence and bind that part to some specific meaning, and continue parsing using grounded meaning.

Wong and Mooney (2007) also use type information to help reduce unnecessary tree joining in decoding. However, their types are static, while our type system is stronger so that we can infer type from polymorphism, which gives use better search quality in decoding.

7 Conclusions and Future Work

We have presented an incremental semantic parser that is guided by a powerful type system of subtyping and parametric 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 orders of magnitude faster than CKY-based parsers in theory and in practice.

For future work, we would like to work on weakly supervised learning that learn from question-answer pairs instead of question-MR pairs, where the datasets are larger, and TISP should benefit more on such problems.
References

[2005] Charniak, E., and Johnson, M. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In Proceedings of ACL, 173–180.

[2014] Goldberg, Y.; Sartorio, F.; and Satta, G. 2014. A tabular method for dynamic oracles in transition-based parsing.

[2008] He, Z.; Liu, Q.; and Lin, S. 2008. Improving statistical machine translation using lexicalized rule selection. In Proceedings of COLING, 321–328.

[1998] Heim, I., and Kratzer, A. 1998. Semantics in Generative Grammar. Blackwell Publishing.

[2012] Huang, L.; Fayong, S.; and Guo, Y. 2012. Structured perceptron with inexact search. In Proceedings of NAACL.

[2011] Kwiatkowski, T.; Zettlemoyer, L.; Goldwater, S.; and Steedman, M. 2011. Lexical generalization in ccg grammar induction for semantic parsing. In Proceedings of EMNLP, EMNLP '11.

[2013] Kwiatkowski, T.; Choi, E.; Artzi, Y.; and Zettlemoyer, L. 2013. Scaling semantic parsers with on-the-fly ontology matching.

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

[2011] Lu, W., and Ng, H. T. 2011. A probabilistic forest-to-string model for language generation from typed lambda calculus expressions. In Proceedings of EMNLP.

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

[2002] Pierce, B. C. 2002. Types and Programming Languages. MIT Press.

[2007] Wong, Y. W., and Mooney, R. J. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In Annual Meeting-Association for computational Linguistics, volume 45, 960.

[2013] Yu, H.; Huang, L.; Mi, H.; and Zhao, K. 2013. Max-violation perceptron and forced decoding for scalable mt training. In Proceedings of EMNLP 2013.

[2005] Zettlemoyer, L., and Collins, M. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In Proceedings of UAI.

[2007] Zettlemoyer, L. S., and Collins, M. 2007. Online learning of relaxed ccg grammars for parsing to logical form. In In Proceedings of EMNLP-CoNLL-2007. Citeseer.

[2011] Zhang, Y., and Clark, S. 2011. Shift-reduce ccg parsing. In Proceedings of ACL.