Title
Learning Dependency-Based Compositional Semantics

Author
Liang, Percy

Publication Date
2011

Peer reviewed|Thesis/dissertation

Learning Dependency-Based Compositional Semantics

by

Percy Shuo Liang

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering and Computer Sciences

and the Designated Emphasis

in

Communication, Computation, and Statistics

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dan Klein, Chair
Professor Michael I. Jordan
Professor Tom Griffiths

Fall 2011

Learning Dependency-Based Compositional Semantics

Abstract

Learning Dependency-Based Compositional Semantics

by

Percy Shuo Liang

Doctor of Philosophy in Electrical Engineering and Computer Sciences

and the Designated Emphasis

in

Communication, Computation, and Statistics

University of California, Berkeley

Professor Dan Klein, Chair

Suppose we want to build a system that answers a natural language question by representing its semantics as a logical form and computing the answer given a structured database of facts. The core part of such a system is the semantic parser that maps questions to logical forms. Semantic parsers are typically trained from examples of questions annotated with their target logical forms, but this type of annotation is expensive.

Our goal is to learn a semantic parser from question-answer pairs instead, where the logical form is modeled as a latent variable. Motivated by this challenging learning problem, we develop a new semantic formalism, dependency-based compositional semantics (DCS), which has favorable linguistic, statistical, and computational properties. We define a log-linear distribution over DCS logical forms and estimate the parameters using a simple procedure that alternates between beam search and numerical optimization. On two standard semantic parsing benchmarks, our system outperforms all existing state-of-the-art systems, despite using no annotated logical forms.

To my parents...

and

to Ashley.

Contents

Acknowledgments

My last six years at Berkeley have been an incredible journey—one where I have grown tremendously, both academically and personally. None of this would have been possible without the support of the many people I have interacted with over the years.

First, I am extremely grateful to have been under the guidance of not one great adviser, but two. Michael Jordan and Dan Klein didn't just teach me new things—they truly shaped the way I think in a fundamental way and gave me invaluable perspective on research. From Mike, I learned how to really make sense of mathematical details and see the big important ideas; Dan taught me how to make sense of data and read the mind of a statistical model. The two influences complemented each other wonderfully, and I cannot think of a better environment for me.

Having two advisers also meant having the fortune of interacting with two vibrant research groups. I remember having many inspiring conversations in meeting rooms, hallways, and at Jupiter—thanks to all the members of the SAIL and NLP groups, as well as Peter Bartlett's group. I also had the pleasure of collaborating on projects with Alexandre Bouchard-Côté, Ben Taskar, Slav Petrov, Tom Griffiths, Aria Haghighi, Taylor Berg-Kirkpatrick, Ben Blum, Jake Abernethy, Alex Simma, and Ariel Kleiner. I especially want to thank Dave Golland and Gabor Angeli, whom I really enjoyed mentoring.

Many people outside of Berkeley also had a lasting impact on me. Nati Srebro introduced me to research when I was an undergraduate at MIT, and first got me excited about machine learning. During my masters at MIT, Michael Collins showed me the rich world of statistical NLP. I also learned a great deal from working with Paul Viola, Martin Szummer, Francis Bach, Guillaume Bouchard, and Hal Daume through various collaborations. In a fortuitous turn of events, I ended up working on program analysis with Mayur Naik, Omer Tripp, and Mooly Sagiv. Mayur Naik taught me almost everything I know about the field of programming languages.

There are several additional people I'd like to thank: John Duchi, who gave me an excuse to bike long distances for my dinner; Mike Long, with whom I first explored the beauty of the Berkeley hills; Rodolphe Jenatton, avec qui j'ai appris à parler français et courir en même temps; John Blitzer, with whom I seemed to get better research done in the hills than in the office; Kurt Miller, with whom I shared exciting adventures in St. Petersburg, to be continued in New York; Blaine Nelson, who introduced me to great biking in the Bay Area; my Ashby housemates (Dave Golland, Fabian Wauthier, Garvesh Raskutti, Lester Mackey), with whom I had many whole wheat conversations not about whole wheat; Haggai Niv and Ray Lifchez, pillars of my musical life; Simon Lacoste-Julien and Sasha Skorokhod, who expanded my views on life; and Alex Bouchard-Côté, with whom I shared so many vivid memories during our years at Berkeley.

My parents, Frank Liang and Ling Zhang, invested in me more than I can probably imagine, and they have provided their unwavering support throughout my life—words simply fail to express my appreciation for them. Finally, Ashley, thank you for your patience during my long journey, for always providing a fresh perspective, for your honesty, and for always being there for me.

Chapter 1

Introduction

We are interested in building a system that can answer natural language questions given a structured database of facts. As a running example, consider the domain of US geography (Figure 1.1).

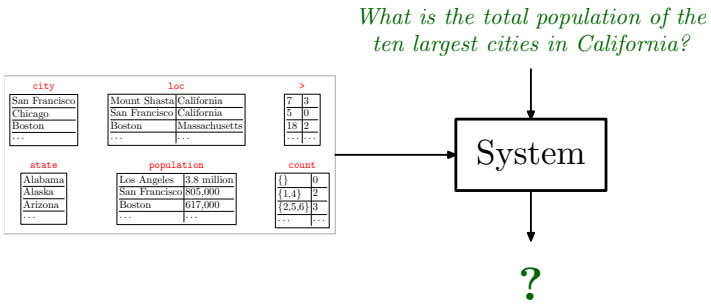


Figure 1.1: The goal: a system that answers natural language questions given a structured database of facts. An example is shown in the domain of US geography.

The problem of building these *natural language interfaces to databases* (NLIDBs) has a long history in NLP, starting from the early days of AI with systems such as Lunar (Woods et al., 1972), Chat-80 (Warren and Pereira, 1982), and many others (see Androutsopoulos et al. (1995) for an overview). While quite successful in their respective limited domains, because these systems were constructed from manually-built rules, they became difficult to scale up, both to other domains and to more complex utterances. In response, against the backdrop of a statistical revolution in NLP during the 1990s, researchers began to build systems that could learn from examples, with the hope of overcoming the limitations of rule-based methods. One of the earliest statistical efforts was the Chill system (Zelle and Mooney, 1996), which learned a shift-reduce semantic parser. Since then, there has been a healthy line of work yielding increasingly more accurate semantic parsers by using new semantic representations and machine learning techniques (Zelle and Mooney, 1996; Miller

et al., 1996; Tang and Mooney, 2001; Ge and Mooney, 2005; Kate et al., 2005; Zettlemoyer and Collins, 2005; Kate and Mooney, 2006; Wong and Mooney, 2006; Kate and Mooney, 2007; Zettlemoyer and Collins, 2007; Wong and Mooney, 2007; Kwiatkowski et al., 2010, 2011).

However, while statistical methods provided advantages such as robustness and portability, their application in semantic parsing achieved only limited success. One of the main obstacles was that these methods depended crucially on having examples of utterances paired with logical forms, and this requires substantial human effort to obtain. Furthermore, the annotators must be proficient in some formal language, which drastically reduces the size of the annotator pool, dampening any hope of acquiring enough data to fulfill the vision of learning highly accurate systems.

In response to these concerns, researchers have recently begun to explore the possibility of learning a semantic parser without any annotated logical forms (Clarke et al., 2010; Liang et al., 2011; Goldwasser et al., 2011; Artzi and Zettlemoyer, 2011). It is in this vein that we develop our present work. Specifically, given a set of $(\mathbf{x}, y)$ example pairs, where $\mathbf{x}$ is an utterance (e.g., a question) and y is the corresponding answer, we wish to learn a mapping from $\mathbf{x}$ to y. What makes this mapping particularly interesting is it passes through a latent logical form z, which is necessary to capture the semantic complexities of natural language. Also note that while the logical form z was the end goal in past work on semantic parsing, for us, it is just an intermediate variable—a means towards an end. Figure 1.2 shows the graphical model which captures the learning setting we just described: The question $\mathbf{x}$, answer y, and world/database w are all observed. We want to infer the logical forms z and the parameters θ of the semantic parser, which are unknown quantities.

While liberating ourselves from annotated logical forms reduces cost, it does increase the difficulty of the learning problem. The core challenge here is *program induction*: on each example $(\mathbf{x}, y)$, we need to efficiently search over the exponential space of possible logical forms z and find ones that produces the target answer y, a computationally daunting task. There is also a statistical challenge: how do we parametrize the mapping from utterance $\mathbf{x}$ to logical form z so that it can be learned from only the indirect signal y? To address these two challenges, we must first discuss the issue of *semantic representation*. There are two basic questions here: (i) what should the formal language for the logical forms z be, and (ii) what are the compositional mechanisms for constructing those logical forms?

The semantic parsing literature is quite multilingual with respect to the formal language used for the logical form: Researchers have used SQL (Giordani and Moschitti, 2009), Prolog (Zelle and Mooney, 1996; Tang and Mooney, 2001), a simple functional query language called FunQL (Kate et al., 2005), and lambda calculus (Zettlemoyer and Collins, 2005), just to name a few. The construction mechanisms are equally diverse, including synchronous grammars (Wong and Mooney, 2007), hybrid trees (Lu et al., 2008), Combinatorial Categorial Grammars (CCG) (Zettlemoyer and Collins, 2005), and shift-reduce derivations (Zelle and Mooney, 1996). It is worth pointing out that the choice of formal language and the construction mechanism are decisions which are really more orthogonal than it is often

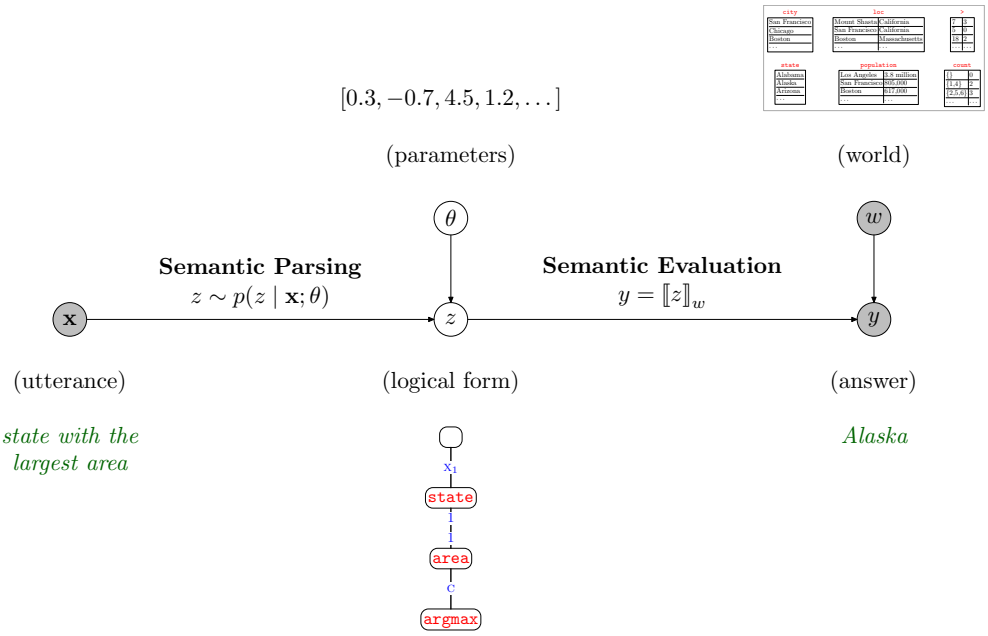


Figure 1.2: Our probabilistic model consists of two steps: (i) semantic parsing: an utterance $\mathbf{x}$ is mapped to a logical form z by drawing from a log-linear distribution parametrized by a vector θ; and (ii) evaluation: the logical form z is evaluated with respect to the world w (database of facts) to deterministically produce an answer $y = [\![z]\!]_w$. The figure also shows an example configuration of the variables around the graphical model. Logical forms z as represented as labeled trees. During learning, we are given w and $(\mathbf{x}, y)$ pairs (shaded nodes) and try to infer the latent logical forms z and parameters θ.

assumed—the former is concerned with what the logical forms look like; the latter, how to generate a set of possible logical forms in a compositional way given an utterance. (How to score these logical forms is yet another dimension.)

The formal languages and construction mechanisms in past work were chosen somewhat out of the convenience of conforming to existing standards. For example, Prolog and SQL were existing standard declarative languages built for querying a deductive or relational database, which was convenient for the end application, but they were not designed for representing the semantics of natural language. As a result, the construction mechanism that bridges the gap between natural language and formal language is quite complicated and difficult to learn. CCG (Steedman, 2000) and more generally, categorial grammar, is the de facto standard in linguistics. In CCG, logical forms are constructed compositionally using a small handful of combinators (function application, function composition, and type raising). For a wide range of canonical examples, CCG produces elegant, streamlined analyses, but its success really depends on having a good, clean lexicon. During learning, there is often

large amounts of uncertainty over the lexical entries, which makes CCG more cumbersome. Furthermore, in real-world applications, we would like to handle disfluent utterances, and this further strains CCG by demanding either extra type-raising rules and disharmonic combinators (Zettlemoyer and Collins, 2007) or a proliferation of redundant lexical entries for each word (Kwiatkowski et al., 2010).

To cope with the challenging demands of program induction, we break away from tradition in favor of a new formal language and construction mechanism, which we call *dependency-based compositional semantics* (DCS). The guiding principle behind DCS is to make a simple and intuitive framework for constructing and representing logical forms. Logical forms in DCS are tree structures called DCS trees. The motivation is two-fold: (i) DCS trees are meant to parallel syntactic dependency trees, which facilitates parsing; and (ii) a DCS tree essentially encodes a constraint satisfaction problem, which can be solved efficiently used dynamic programming. In addition, DCS provides a *mark-execute* construct, which provides a uniform way of dealing with scope variation, a major source of trouble in any semantic formalism. The construction mechanism in DCS is a generalization of labeled dependency parsing, which leads to simple and natural algorithms. To a linguist, DCS might appear unorthodox, but it is important to keep in mind that our primary goal is effective program induction, not necessarily to model new linguistic phenomena in the tradition of formal semantics.

Armed with our new semantic formalism, DCS, we then define a discriminative probabilistic model, which is depicted in Figure 1.2. The semantic parser is a log-linear distribution over DCS trees z given an utterance $\mathbf{x}$. Notably, z is unobserved, and we instead observe only the answer y, which is z evaluated on a world/database w. There are an exponential number of possible trees z, and usually dynamic programming is employed for efficiently searching over the space of these combinatorial objects. However, in our case, we must enforce the global constraint that the tree generates the correct answer y, which makes dynamic programming infeasible. Therefore, we resort to beam search and learn our model with a simple procedure which alternates between beam search and optimizing a likelihood objective restricted to those beams. This yields a natural bootstrapping procedure in which learning and search are integrated.

We evaluated our DCS-based approach on two standard benchmarks, GEO, a US geography domain (Zelle and Mooney, 1996) and JOBS, a job queries domain (Tang and Mooney, 2001). On GEO, we found that our system significantly outperforms previous work that also learns from answers instead of logical forms (Clarke et al., 2010). What is perhaps a more significant result is that our system even outperforms state-of-the-art systems that do rely on annotated logical forms. This demonstrates that the viability of training accurate systems with much less supervision than before.

The rest of this thesis is organized as follows: Chapter 2 introduces dependency-based compositional semantics (DCS), our new semantic formalism. Chapter 3 presents our probabilistic model and learning algorithm. Chapter 4 provides an empirical evaluation of our methods. Finally, Chapter 5 situates this work in a broader context.

Chapter 2

Representation

In this chapter, we present the main conceptual contribution of this work, dependency-based compositional semantics (DCS), using the US geography domain (Zelle and Mooney, 1996) as a running example. To do this, we need to define the syntax and semantics of the formal language. The syntax is defined in Section 2.2 and is quite straightforward: The logical forms in the formal language are simply trees, which we call *DCS trees*. In Section 2.3, we give a type-theoretic definition of *worlds* (also known as databases or models) with respect to which we can define the semantics of DCS trees.

The semantics, which is the heart of this thesis, contains two main ideas: (i) using trees to represent logical forms as constraint satisfaction problems or extensions thereof, and (ii) dealing with cases when syntactic and semantic scope diverge (e.g., for generalized quantification and superlative constructions) using a new construct which we call *mark-execute*. We start in Section 2.4 by introducing the semantics of a basic version of DCS which focuses only on (i) and then extend it to the full version (Section 2.5) to account for (ii).

Finally, having fully specified the formal language, we describe a construction mechanism for mapping a natural language utterance to a set of candidate DCS trees (Section 2.6).

2.1 Notation

Operations on tuples will play a prominent role in this thesis. For a sequence[1] $v = (v_1, \ldots, v_k)$, we use $|v| = k$ to denote the length of the sequence. For two sequences u and v, we use $u + v = (u_1, \ldots, u_{|u|}, v_1, \ldots, v_{|v|})$ to denote their concatenation.

For a sequence of positive indices $\mathbf{i} = (i_1, \ldots, i_m)$, let $v_\mathbf{i} = (v_{i_1}, \ldots, v_{i_m})$ consist of the components of v specified by $\mathbf{i}$; we call $v_\mathbf{i}$ the projection of v onto $\mathbf{i}$. We use negative indices

[1] We use the *sequence* to include both tuples $(v_1, \ldots, v_k)$ and arrays $[v_1, \ldots, v_k]$. For our purposes, there is no functional difference between tuples and arrays; the distinction is convenient when we start to talk about arrays of tuples.

to exclude components: $v_{-\mathbf{i}} = (v_{(1,\ldots,|v|)\setminus\mathbf{i}})$. We can also combine sequences of indices by concatenation: $v_{\mathbf{i},\mathbf{j}} = v_{\mathbf{i}} + v_{\mathbf{j}}$. Some examples: if $v = (a, b, c, d)$, then $v_2 = b$, $v_{3,1} = (c, a)$, $v_{-3} = (a, b, d)$, $v_{3,-3} = (c, a, b, d)$.

2.2 Syntax of DCS Trees

The syntax of the DCS formal language is built from two ingredients, *predicates* and *relations*:

- Let $\mathcal{P}$ be a set of *predicates*. We assume that $\mathcal{P}$ always contains a special null predicate $\emptyset$ and several domain-independent predicates (e.g., `count`, `<`, `>`, and `=`). In addition, $\mathcal{P}$ contains domain-specific predicates. For example, for the US geography domain, $\mathcal{P}$ would include `state`, `river`, `border`, etc. Right now, think of predicates as just labels, which have yet to receive formal semantics.
- Let $\mathcal{R}$ be the set of *relations*. The full set of relations are shown in Table 2.1; note that unlike the predicates $\mathcal{P}$, the relations $\mathcal{R}$ are fixed.

The logical forms in DCS are called DCS trees. A DCS tree is a directed rooted tree in which nodes are labeled with predicates and edges are labeled with relations; each node also maintains an ordering over its children. Formally:

Definition 1 (DCS trees) *Let $\mathcal{Z}$ be the set of DCS trees, where each $z \in \mathcal{Z}$ consists of (i) a predicate $z.p \in \mathcal{P}$ and (ii) a sequence of edges $z.\mathbf{e} = (z.e_1, \ldots, z.e_m)$. Each edge e consists of a relation $e.r \in \mathcal{R}$ (see Table 2.1) and a child tree $e.c \in \mathcal{Z}$.*

We will either draw a DCS tree graphically or write it compactly as $\langle p; r_1 : c_1; \ldots; r_m : c_m \rangle$ where p is the predicate at the root node and $c_1, \ldots, c_m$ are its m children connected via edges labeled with relations $r_1, \ldots, r_m$, respectively. Figure 2.2(a) shows an example of a DCS tree expressed using both graphical and compact formats.

A DCS tree is a logical form, but it is designed to look like a syntactic dependency tree, only with predicates in place of words. As we'll see over the course of this chapter, it is this transparency between syntax and semantics provided by DCS which leads to a simple and streamlined compositional semantics suitable for program induction.

2.3 Worlds

In the context of question answering, the DCS tree is a formal specification of the question. To obtain an answer, we still need to evaluate the DCS tree with respect to a database of facts (see Figure 2.1 for an example). We will use the term *world* to refer to this database (it is sometimes also called model, but we avoid this term to avoid confusion with the probabilistic model that we will present in Section 3.1).

Relations $\mathcal{R}$

Name	Relation	Description
join	$\overset{j}{j'}$ for $j, j' \in \{1, 2, \dots\}$	j-th component of parent = j'-th component of child
aggregate	Σ	parent = set of feasible values of child
extract	E	mark node for extraction
quantify	Q	mark node for quantification, negation
compare	C	mark node for superlatives, comparatives
execute	$\mathrm{X}_{\mathbf{i}}$ for $\mathbf{i} \in \{1, 2 \dots\}^*$	process marked nodes specified by $\mathbf{i}$

Table 2.1: Possible relations that appear on edges of DCS trees. Basic DCS uses only the join and aggregate relations; the full version uses all of them.

w:

`city`

San Francisco
Chicago
Boston
...

`loc`

Mount Shasta	California
San Francisco	California
Boston	Massachusetts
...	...

`>`

7	3
5	0
18	2
...	...

`state`

Alabama
Alaska
Arizona
...

`population`

Los Angeles	3.8 million
San Francisco	805,000
Boston	617,000
...	...

`count`

{}	0
{1,4}	2
{2,5,6}	3
...	...

Figure 2.1: An example of a world w (database) in the US geography domain. A world maps each predicate (relation) to a set of tuples. Here, the world w maps the predicate `loc` to the set of pairs of places and their containers. Note that functions (e.g., `population`) are also represented as predicates for uniformity. Some predicates (e.g., `count`) map to an infinite number of tuples and would be represented implicitly.

2.3.1 Types and Values

To define a world, we start by constructing a set of *values* $\mathcal{V}$. The exact set of values depend on the domain (we will continue to use US geography as a running example). Briefly, $\mathcal{V}$ contains numbers (e.g., $3 \in \mathcal{V}$), strings (e.g., *Washington* $\in \mathcal{V}$), tuples (e.g., (3, *Washington*) $\in \mathcal{V}$), sets (e.g., $\{3, \textit{Washington}\} \in \mathcal{V}$), and other higher-order entities.

To be more precise, we construct $\mathcal{V}$ recursively. First, define a set of primitive values $\mathcal{V}_\star$, which includes the following:

- Numeric values: each value has the form $x\!:\!t \in \mathcal{V}_\star$, where $x \in \mathbb{R}$ is a real number and $t \in \{\texttt{number}, \texttt{ordinal}, \texttt{percent}, \texttt{length}, \dots\}$ is a tag. The tag allows us to differentiate

3, 3rd, 3%, and 3 miles—this will be important in Section 2.6.3. We simply write x for the value x:`number`.

- Symbolic values: each value has the form $x : t \in \mathcal{V}_\star$, where x is a string (e.g., *Washington*) and $t \in \{\texttt{string}, \texttt{city}, \texttt{state}, \texttt{river}, \dots\}$ is a tag. Again, the tag allows us to differentiate, for example, the entities *Washington*:`city` and *Washington*:`state`.

Now we build the full set of values $\mathcal{V}$ from the primitive values $\mathcal{V}_\star$. To define $\mathcal{V}$, we need a bit more machinery: To avoid logical paradoxes, we construct $\mathcal{V}$ in increasing order of complexity using types (see Carpenter (1998) for a similar construction). The casual reader can skip this construction without losing any intuition.

Define the set of types $\mathcal{T}$ to be the smallest set that satisfies the following properties:

1. The primitive type $\star \in \mathcal{T}$;
2. The tuple type $(t_1, \dots, t_k) \in \mathcal{T}$ for each $k \geq 0$ and each non-tuple type $t_i \in \mathcal{T}$ for $i = 1, \dots, k$; and
3. The set type $\{t\} \in \mathcal{T}$ for each tuple type $t \in \mathcal{T}$.

Note that $\{\star\}$, $\{\{\star\}\}$, and $((\star))$ are not valid types.

For each type $t \in \mathcal{T}$, we construct a corresponding set of values $\mathcal{V}_t$:

1. For the primitive type $t = \star$, the primitive values $\mathcal{V}_\star$ have already been specified. Note that these types are rather coarse: Primitive values with different tags are considered to have the same type $\star$.
2. For a tuple type $t = (t_1, \dots, t_k)$, $\mathcal{V}_t$ is the cross product of the values of its component types:
$$\mathcal{V}_t = \{(v_1, \dots, v_k) : \forall i, v_i \in \mathcal{V}_{t_i}\}. \tag{2.1}$$
3. For a set type $t = \{t'\}$, $\mathcal{V}_t$ contains all subsets of its element type t':
$$\mathcal{V}_t = \{s : s \subset \mathcal{V}_{t'}\}. \tag{2.2}$$
Note that all elements of the set must have the same type.

Let $\mathcal{V} = \cup_{t \in \mathcal{T}} \mathcal{V}_t$ be the set of all possible values.

A world maps each predicate to its *semantics*, which is a set of tuples (see Figure 2.1 for an example). First, let $\mathcal{T}_{\text{TUPLE}} \subset \mathcal{T}$ be the tuple types, which are the ones of the form $(t_1, \dots, t_k)$. Let $\mathcal{V}_{\{\text{TUPLE}\}}$ denote all the sets of tuples (with the same type):
$$\mathcal{V}_{\{\text{TUPLE}\}} \stackrel{\text{def}}{=} \bigcup_{t \in \mathcal{T}_{\text{TUPLE}}} \mathcal{V}_{\{t\}}. \tag{2.3}$$

Now we define a world formally:

Definition 2 (World) *A* world $w : \mathcal{P} \mapsto \mathcal{V}_{\{\text{TUPLE}\}} \cup \{\mathcal{V}\}$ *is a function that maps each non-null predicate* $p \in \mathcal{P} \backslash \{\emptyset\}$ *to a set of tuples* $w(p) \in \mathcal{V}_{\{\text{TUPLE}\}}$ *and maps the null predicate* $\emptyset$ *to the set of all values* ($w(\emptyset) = \mathcal{V}$).

For a set of tuples A with the same arity, let $\textsc{Arity}(A) = |x|$, where $x \in A$ is arbitrary; if A is empty, then $\textsc{Arity}(A)$ is undefined. Now for a predicate $p \in \mathcal{P}$ and world w, define $\textsc{Arity}_w(p)$, the arity of predicate p with respect to w, as follows:

$$\textsc{Arity}_w(p) = \begin{cases} 1 & \text{if } p = \emptyset, \\ \textsc{Arity}(w(p)) & \text{if } p \neq \emptyset. \end{cases} \tag{2.4}$$

The null predicate has arity 1 by fiat; the arity of a non-null predicate p is inherited from the tuples in $w(p)$.

Remarks In higher-order logic and lambda calculus, we construct function types and values, whereas in DCS, we construct tuple types and values. The two are equivalent in representational power, but this discrepancy does point at the fact that lambda calculus is based on function application, whereas DCS, as we will see, is based on declarative constraints. The set type $\{(\star, \star)\}$ in DCS corresponds to the function type $\star \to (\star \to \texttt{bool})$. In DCS, there is no explicit `bool` type—it is implicitly represented by using sets.

2.3.2 Examples

The world w maps each domain-specific predicate to a finite set of tuples. For the US geography domain, w has a predicate that maps to the set of US states (`state`), another predicate that maps to set of pairs of entities and where they are located (`loc`), and so on:

$$w(\texttt{state}) = \{(\textit{California}{:}\texttt{state}), (\textit{Oregon}{:}\texttt{state}), \dots\}, \tag{2.5}$$

$$w(\texttt{loc}) = \{(\textit{San Francisco}{:}\texttt{city}, \textit{California}{:}\texttt{state}), \dots\} \tag{2.6}$$

$$\dots \tag{2.7}$$

To shorten notation, we use state abbreviations (e.g., $\texttt{CA} = \textit{California}{:}\texttt{state}$).

The world w also specifies the semantics of several domain-independent predicates (think of these as helper functions), which usually correspond to an infinite set of tuples. Functions are represented in DCS by a set of input-output pairs. For example, the semantics of the $\texttt{count}_t$ predicate (for each type $t \in \mathcal{T}$) contains pairs of sets S and their cardinalities $|S|$:

$$w(\texttt{count}_t) = \{(S, |S|) : S \in \mathcal{V}_{\{(t)\}}\} \in \mathcal{V}_{\{(\{(t)\}, \star)\}}. \tag{2.8}$$

As another example, consider the predicate $\texttt{average}_t$ (for each $t \in \mathcal{T}$), which takes a set of key-value pairs (with keys of type t) and returns the average value. For notational convenience, we treat an arbitrary set of pairs S as a set-valued function: We let $S_1 = \{x :$

$(x, y) \in S\}$ denote the domain of the function, and abusing notation slightly, we define the function $S(x) = \{y : (x, y) \in S\}$ to be the set of values y that co-occur with the given x. The semantics of $\texttt{average}_t$ contains pairs of sets and their averages:

$$w(\texttt{average}_t) = \left\{ (S, z) : S \in \mathcal{V}_{\{(t,\star)\}}, z = |S_1|^{-1} \sum_{x \in S_1} \left[|S(x)|^{-1} \sum_{y \in S(x)} y \right] \right\} \in \mathcal{V}_{\{(\{(t,\star)\},\star)\}}. \tag{2.9}$$

Similarly, we can define the semantics of $\texttt{argmin}_t$ and $\texttt{argmax}_t$, which each takes a set of key-value pairs and returns the keys that attain the smallest (largest) value:

$$w(\texttt{argmin}_t) = \left\{ (S, z) : S \in \mathcal{V}_{\{(t,\star)\}}, z \in \operatorname*{argmin}_{x \in S_1} \min S(x) \right\} \in \mathcal{V}_{\{(\{(t,\star)\},t)\}}, \tag{2.10}$$

$$w(\texttt{argmax}_t) = \left\{ (S, z) : S \in \mathcal{V}_{\{(t,\star)\}}, z \in \operatorname*{argmax}_{x \in S_1} \max S(x) \right\} \in \mathcal{V}_{\{(\{(t,\star)\},t)\}}. \tag{2.11}$$

These helper functions are monomorphic: For example, $\texttt{count}_t$ only computes cardinalities of sets of type $\{(t)\}$. In practice, we mostly operate on sets of primitives ($t = \star$). To reduce notation, we simply omit t to refer to this version: $\texttt{count} = \texttt{count}_\star$, $\texttt{average} = \texttt{average}_\star$, etc.

2.4 Semantics of DCS Trees: Basic Version

The semantics or *denotation* of a DCS tree z with respect to a world w is denoted $[\![z]\!]_w$. First, we define the semantics of DCS trees with only join relations (Section 2.4.1). In this case, a DCS tree encodes a constraint satisfaction problem (CSP); this is important because it highlights the constraint-based nature of DCS and also naturally leads to a computationally efficient way of computing denotations (Section 2.4.2). We then allow DCS trees to have aggregate relations (Section 2.4.3). The fragment of DCS which has only join and aggregate relations is called *basic DCS*.

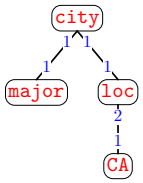
2.4.1 DCS Trees as Constraint Satisfaction Problems

Let z be a DCS tree with only join relations on its edges. In this case, z encodes a constraint satisfaction problem (CSP) as follows: For each node x in z, the CSP has a variable $a(x)$; the collection of these variables is referred to as an *assignment* a. The predicates and relations of z introduce constraints:

1. $a(x) \in w(p)$ for each node x labeled with predicate $p \in \mathcal{P}$; and

Example: *major city in California*

$z = \langle \texttt{city}; \frac{1}{1}\!:\!\langle \texttt{major} \rangle ; \frac{1}{1}\!:\!\langle \texttt{loc}; \frac{2}{1}\!:\!\langle \texttt{CA} \rangle \rangle \rangle$



(a) DCS tree

$$\lambda c\, \exists m\, \exists \ell\, \exists s\, .\ \texttt{city}(c) \wedge \texttt{major}(m) \wedge \texttt{loc}(\ell) \wedge \texttt{CA}(s) \wedge c_1 = m_1 \wedge c_1 = \ell_1 \wedge \ell_2 = s_1$$

(b) Lambda calculus formula

(c) Denotation: $[\![z]\!]_w = \{\texttt{SF}, \texttt{LA}, \ldots\}$

Figure 2.2: (a) An example of a DCS tree (written in both the mathematical and graphical notation). Each node is labeled with a predicate, and each edge is labeled with a relation. (b) A DCS tree z with only join relations encodes a constraint satisfaction problem. (c) The denotation of z is the set of feasible values for the root node.

2. $a(x)_j = a(y)_{j'}$ for each edge (x, y) labeled with $\frac{j}{j'} \in \mathcal{R}$, which says that the j-th component of $a(x)$ must equal the j'-th component of $a(y)$.

We say that an assignment a is *feasible* if it satisfies all the above constraints. Next, for a node x, define $V(x) = \{a(x) : \text{assignment } a \text{ is feasible}\}$ as the set of feasible values for x—these are the ones which are consistent with at least one feasible assignment. Finally, we define the denotation of the DCS tree z with respect to the world w to be $[\![z]\!]_w = V(x_0)$, where x_0 is the root node of z.

Figure 2.2(a) shows an example of a DCS tree. The corresponding CSP has four variables c, m, ℓ, s.[2] In Figure 2.2(b), we have written the equivalent lambda calculus formula. The non-root nodes are existentially quantified, the root node c is λ-abstracted, and all constraints introduced by predicates and relations are conjoined. The λ-abstraction of c represents the fact that the denotation is the set of feasible values for c (note the equivalence between the boolean function $\lambda c.p(c)$ and the set $\{c : p(c)\}$).

Remarks Note that CSPs only allow existential quantification and conjunction. Why did we choose this particular logical subset as a starting point, rather than allowing universal quantification, negation, or disjunction? There seems to be something fundamental about this subset, which also appears in Discourse Representation Theory (DRT) (Kamp and Reyle, 1993; Kamp et al., 2005). Briefly, logical forms in DRT are called Discourse Representation Structures (DRSes), each of which contains (i) a set of existentially-quantified discourse

[2] Technically, the node is c and the variable is $a(c)$, but we use c to denote the variable to simplify notation.

referents (variables), (ii) a set of conjoined discourse conditions (constraints), and (iii) nested DRSes. If we exclude nested DRSes, a DRS is exactly a CSP.[3] The default existential quantification and conjunction are quite natural for modeling cross-sentential anaphora: New variables can be added to a DRS and connected to other variables. Indeed, DRT was originally motivated by these phenomena (see Kamp and Reyle (1993) for more details).

Tree-structured CSPs can capture unboundedly complex recursive structures—such as *cities in states that border states that have rivers that...*. However, one limitation which will stay with us even with the full version of DCS, is the inability to capture long distance dependencies arising from anaphora (ironically, given our connection to DRT). For example, consider the phrase *a state with a river that traverses its capital*. Here, *its* binds to *state*, but this dependence cannot be captured in a tree structure. However, given the highly graphical context in which we've been developing DCS, a solution leaps out at us immediately: Simply introduce an edge between the *its* node and the *state* node. This edge introduces a CSP constraint that the two nodes must be equal. The resulting structure is a graph rather than a tree, but still a CSP. In this thesis, we will not pursue these non-tree structures, but it should be noted that such an extension is possible and quite natural.

2.4.2 Computation

So far, we have given a declarative definition of the denotation $[\![z]\!]_w$ of a DCS tree z with only join relations. Now we will show how to compute $[\![z]\!]_w$ efficiently. Recall that the denotation is the set of feasible values for the root node. In general, finding the solution to a CSP is NP-hard, but for trees, we can exploit dynamic programming (Dechter, 2003). The key is that the denotation of a tree depends on its subtrees only through their denotations:

$$[\![\left\langle p; {}^{j_1}_{j'_1}\!:c_1; \cdots; {}^{j_m}_{j'_m}\!:c_m \right\rangle]\!]_w = w(p) \cap \bigcap_{i=1}^{m} \{v : v_{j_i} = t_{j'_i}, t \in [\![c_i]\!]_w\}. \tag{2.12}$$

On the right-hand side of (2.12), the first term $w(p)$ is the set of values that satisfy the node constraint, and the second term consists of an intersection across all m edges of $\{v : v_{j_i} = t_{j'_i}, t \in [\![c_i]\!]_w\}$, which is the set of values v which satisfy the edge constraint with respect to some value t for the child c_i.

To further flesh out this computation, we express (2.12) in terms of two operations: *join* and *project*. Join takes a cross product of two sets of tuples and keeps the resulting tuples that match the join constraint:

$$A \bowtie_{j,j'} B = \{u + v : u \in A, v \in B, u_j = v'_j\}. \tag{2.13}$$

Project takes a set of tuples and keeps a fixed subset of the components:

$$A[\mathbf{i}] = \{v_{\mathbf{i}} : v \in A\}. \tag{2.14}$$

[3]DRSes are not necessarily tree-structured, though economical DRT (Bos, 2009) imposes a tree-like restriction on DRSes for computational reasons.

The denotation in (2.12) can now be expressed in terms of these join and project operations:

$$[\![\left\langle p; {}^{j_1}_{j'_1}\!:c_1; \cdots; {}^{j_m}_{j'_m}\!:c_m \right\rangle]\!]_w = ((w(p) \bowtie_{j_1,j'_1} [\![c_1]\!]_w)[\mathbf{i}] \cdots \bowtie_{j_m,j'_m} [\![c_m]\!]_w)[\mathbf{i}], \tag{2.15}$$

where $\mathbf{i} = (1, \ldots, \textsc{Arity}_w(p))$. Projecting onto $\mathbf{i}$ keeps only components corresponding to p.

The time complexity for computing the denotation of a DCS tree $[\![z]\!]_w$ scales linearly with the number of nodes, but there is also a dependence on the cost of performing the join and project operations. For details on how we optimize these operations and handle infinite sets of tuples, see Appendix A.1.

The denotation of DCS trees is defined in terms of the feasible values of a CSP, and the recurrence in (2.12) is only one way of computing this denotation. However, in light of the extensions to come, we now consider (2.12) as the actual definition rather than just a computational mechanism. It will still be useful to refer to the CSP in order to access the intuition of using declarative constraints.

Remarks The fact that trees enable efficient computation is a general theme which appears in many algorithmic contexts, notably, in probabilistic inference in graphical models. Indeed, a CSP can be viewed as a deterministic version of a graphical model where we maintain only supports of distributions (sets of feasible values) rather than the full distribution. Recall that trees—dependency trees, in particular—also naturally capture the syntactic locality of natural language utterances. This is not a coincidence for the two are intimately connected, and DCS trees highlight their connection.[4]

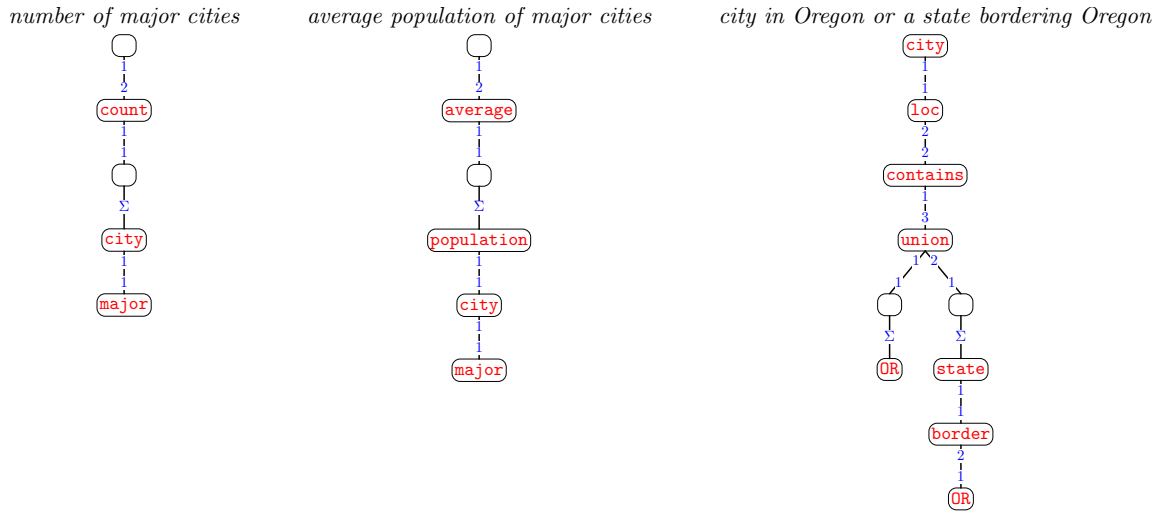
2.4.3 Aggregate Relation

Thus far, we have focused on DCS trees that only use join relations, which are insufficient for capturing higher-order phenomena in language. For example, consider the phrase *number of major cities*. Suppose that *number* corresponds to the `count` predicate, and that *major cities* maps to the DCS tree $\langle \texttt{city}; {}^{1}_{1}\!:\langle \texttt{major} \rangle \rangle$. We cannot simply join `count` with the root of this DCS tree because `count` needs to be joined with the *set* of major cities (the denotation of $\langle \texttt{city}; {}^{1}_{1}\!:\langle \texttt{major} \rangle \rangle$) not just a single city.

We therefore introduce the *aggregate relation* (Σ) that takes a DCS subtree and reifies its denotation so that it can be accessed by other nodes in its entirety. Consider a tree $\langle \Sigma\!:c \rangle$, where the root is connected to a child c via Σ. The denotation of the root is simply the singleton set containing the denotation of c:

$$[\![\langle \Sigma\!:c \rangle]\!]_w = \{([\![c]\!]_w)\}. \tag{2.16}$$

[4] As a side note, if we introduce k non-tree edges, we obtain a graph with tree-width at most k. We could then modify our recurrences to compute the denotation of those graphs, akin to the junction tree algorithm in graphical models.



(a) Counting (b) Averaging (c) Disjunction

Figure 2.3: Examples of DCS trees that use the aggregate relation (Σ) to (a) compute the cardinality of a set, (b) take the average over a set, (c) represent a disjunction over two conditions. The aggregate relation sets the parent node deterministically to the denotation of the child node.

Figure 2.3(a) shows the DCS tree for our running example. The denotation of the middle node is $\{(s)\}$, where s is all major cities. Everything above this node is an ordinary CSP: s constrains the `count` node, which in turns constrains the root node to $|s|$. Figure 2.3(b) shows another example of using the aggregate relation Σ. Here, the node right above Σ is constrained to be a set of pairs of major cities and their populations. The `average` predicate then computes the desired answer.

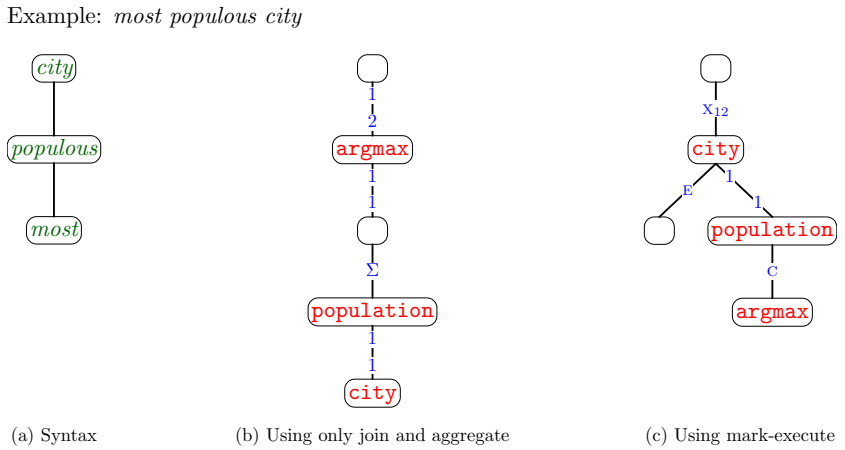
To represent logical disjunction in natural language, we use the aggregate relation and two predicates, `union` and `contains`, which are defined in the expected way:

$$w(\texttt{union}) = \{(S, B, C) : C = A \cup B\}, \tag{2.17}$$

$$w(\texttt{contains}) = \{(A, x) : x \in A\}. \tag{2.18}$$

Figure 2.3(c) shows an example of a disjunctive construction: We use the aggregate relations to construct two sets, one containing Oregon, an the other containing states bordering Oregon. We take the union of these two sets; `contains` takes the set and reads out an element, which then constrains the `city` node.

Remarks A DCS tree that contains only join and aggregate relations can be viewed as a collection of tree-structured CSPs connected via aggregate relations. The tree structure still enables us to compute denotations efficiently based on the recurrences in (2.15) and (2.16).



(a) Syntax (b) Using only join and aggregate (c) Using mark-execute

Figure 2.4: Two semantically-equivalent DCS trees are shown in (b) and (c). The DCS tree in (b), which uses the join and aggregate relations in the basic DCS, does not correspond well with the syntactic structure of *most populous city* (a), and thus is undesirable. The DCS tree in (c), by using the mark-execute construct, corresponds much better, with `city` rightfully dominating its modifiers. The full version of DCS allows us to construct (c), which is preferable to (b).

Recall that a DCS tree with only join relations is a DRS without nested DRSes. The aggregate relation corresponds to the abstraction operator in DRT and is one way of making nested DRSes. It turns the abstraction operator is sufficient to obtain the full representational power of DRT, and subsumes generalized quantification and disjunction constructs in DRT. By analogy, we use the aggregate relation to handle disjunction (Figure 2.3(c)) and generalized quantification (Section 2.5.5).

DCS restricted to join relations is less expressive than first-order logic because it does not have universal quantification, negation, and disjunction. The aggregate relation is analogous to lambda abstraction, which we can use to implement those basic constructs using higher-order predicates (e.g., `not`,`every`,`union`). We can also express logical statements such as generalized quantification, which go beyond first-order logic.

2.5 Semantics of DCS Trees: Full Version

Basic DCS allows only join and aggregate relations, but is already quite expressive. However, it is not enough to simply express a denotation using an arbitrary logical form; one logical form can be better than another even if the two are semantically-equivalent. For example, consider the superlative construction *most populous city*, which has a basic syntactic dependency structure shown in Figure 2.4(a). Figure 2.4(b) shows a DCS tree with only join and

aggregate relations that expresses the correct semantics. However, the two structures are quite divergent—the syntactic head is *city* and the semantic head is `argmax`. This divergence runs counter to the principal motivation of DCS, which is to create a transparent interface between syntax and semantics.

In this section, we resolve this dilemma by introducing mark and execute relations, which will allow us to use the DCS tree in Figure 2.4(c) to represent the same semantics. Importantly, the structure of the semantic structure in Figure 2.4(c) matches the syntactic structure in Figure 2.4(a). The focus of this section is on this *mark-execute* construct—using mark and execute relations to give proper semantically-scoped denotations to syntactically-scoped tree structures.

The basic intuition of the mark-execute construct is as follows: We mark a node low in the tree with a *mark relation*; then higher up in the tree, we invoke it with a corresponding *execute relation* (Figure 2.5). For our example in Figure 2.4(c), we mark the `population` node, which puts the child `argmax` in a temporary store; when we execute the `city` node, we fetch the superlative predicate `argmax` from the store and invoke it.

This divergence between syntactic and semantic scope arises in other linguistic contexts besides superlatives such as quantification and negation. In each of these cases, the general template is the same: a syntactic modifier low in the tree needs to have semantic force higher in the tree. A particularly compelling case of this divergence happens with quantifier scope ambiguity (e.g., *Some river traverses every city.*), where the quantifiers appear in fixed syntactic positions, but the wide or narrow reading correspond to different semantically-scoped denotations. Analogously, a single syntactic structure involving superlatives can also yield two different semantically-scoped denotations—the absolute and relative readings (e.g., *state bordering the largest state*). The mark-execute construct provides a unified framework for dealing all these forms of divergence between syntactic and semantic scope. See Figure 2.6 for more concrete examples.

2.5.1 Denotations

We now formalize our intuitions about the mark-execute construct by defining the denotations of DCS trees that use mark and execute relations. We saw that the mark-execute construct appears to act non-locally, putting things in a store and retrieving them out later. This means that if we want the denotation of a DCS tree to only depend on the denotations of its subtrees, the denotations need to contain more than the set of feasible values for the root node, as was the case for basic DCS. We need to augment denotations to include information about all marked nodes, since these are the ones that can be accessed by an execute relation higher up in the tree.

More specifically, let z be a DCS tree and $d = [\![z]\!]_w$ be its denotation. The denotation d consists of n *columns*, where each column is either the root node of z or a non-executed marked node in z. In the example in Figure 2.7, there are two columns, one for the root `state` node and the other for `size` node, which is marked by C. The columns are ordered

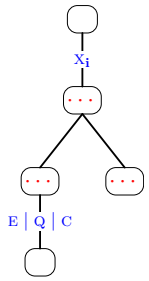


Figure 2.5: The template for the mark-execute construct. For cases where syntactic and semantic scope diverge, the objective is to build DCS trees that resemble syntactic structures but that also encode the correct semantics. Usually, a node low in the tree has a modifier (e.g., `every` or `argmax`). A mark relation (one of E, Q, C) "stores" the modifier. Then an execute relation (of the form $\text{X}_\mathbf{i}$ for indices $\mathbf{i}$) higher up "recalls" the modifier and applies it at the desired semantic point. See Figure 2.6 for examples.

according to a pre-order traversal of z, so column 1 always corresponds to the root node. The denotation d contains a set of arrays $d.A$, where each array represents a feasible assignment of values to the columns of d. For example, in Figure 2.7, the first array in $d.A$ corresponds to assigning (`OK`) to the `state` node (column 1) to and (`TX`, 2.7e5) to the `size` node (column 2). If there are no marked nodes, $d.A$ is basically a set of tuples, which corresponds to a denotation in basic DCS. For each marked node, the denotation d also maintains a *store* with information to be retrieved when that marked node is executed. A store σ for a marked node contains the following: (i) the mark relation $\sigma.r$ (C in the example), (ii) the base denotation $\sigma.b$ which essentially corresponds to denotation of the subtree rooted at the marked node excluding the mark relation and its subtree ($[\![\langle\texttt{size}\rangle]\!]_w$ in the example), and (iii) the denotation of the child of the mark relation ($[\![\langle\texttt{argmax}\rangle]\!]_w$ in the example). The store of any non-marked nodes (e.g., the root) is empty ($\sigma = \emptyset$).

Definition 3 (Denotations) *Let $\mathcal{D}$ be the set of denotations, where each denotation $d \in \mathcal{D}$ consists of*

- *a set of arrays $d.A$, where each array $\mathbf{a} = [a_1, \ldots, a_n] \in d.A$ is a sequence of n tuples; and*
- *a sequence of n* stores *$d.\sigma = (d.\sigma_1, \ldots, d.\sigma_n)$, where each store σ contains a mark relation $\sigma.r \in \{\text{E}, \text{Q}, \text{C}, \emptyset\}$, a base denotation $\sigma.b \in \mathcal{D} \cup \{\emptyset\}$, and a child denotation $\sigma.c \in \mathcal{D} \cup \{\emptyset\}$.*

Note that denotations are formally defined without reference to DCS trees (just as sets of tuples were in basic DCS), but it is sometimes useful to refer to the DCS tree that generates that denotation.

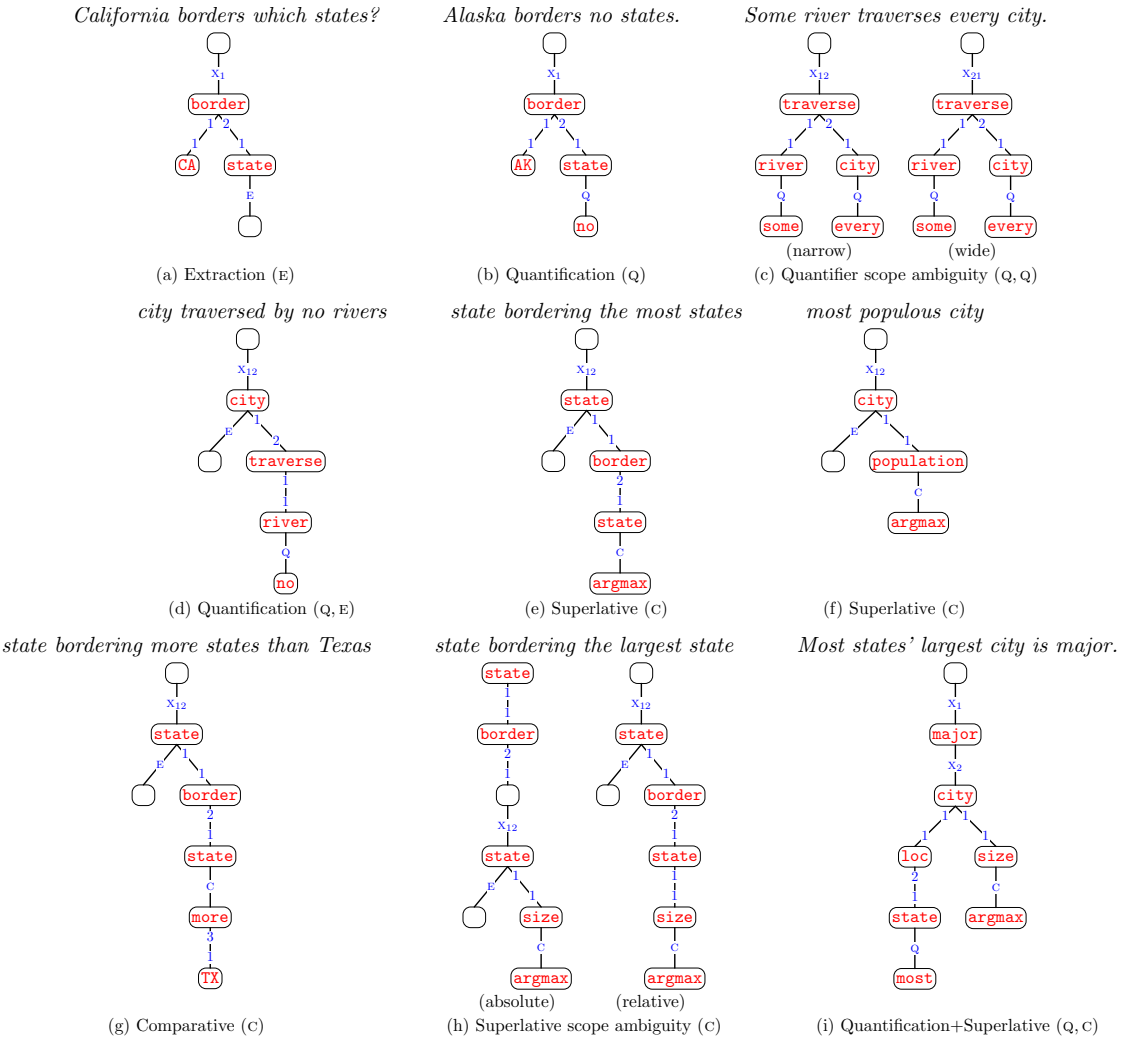


Figure 2.6: Examples of DCS trees that use the mark-execute construct. (a) The head verb *borders* has a direct object *states* modified by *which* and needs to be returned. (b) The quantifier *no* is syntactically dominated by *state* but needs to take wider scope. (c) Two quantifiers yield two possible readings; we build the same basic structure, marking both quantifiers; the choice the execute relation (X_{12} versus X_{21}) determines the reading. (d) We employ two mark relations, Q on `river` for the negation, and E on `city` to force the quantifier to be computed for each value of `city`. (e,f,g) Analogous construction but with the C relation (for comparatives and superlatives). (h) Analog of quantifier scope ambiguity for superlatives: the placement of the execute relation determines an absolute versus relative reading. (i) Interaction between a quantifier and a superlative.

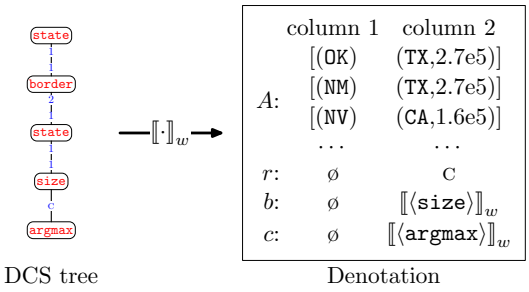


Figure 2.7: Example of the denotation for a DCS tree with a compare relation C. This denotation has two columns, one for each active node—the root node `state` and the marked node `size`.

For notational convenience, we write d as $\langle\!\langle A; (r_1, b_1, c_1); \ldots; (r_n, b_n, c_n) \rangle\!\rangle$. Also let $d.r_i = d.\sigma_i.r$, $d.b_i = d.\sigma_i.b$, and $d.c_i = d.\sigma_i.c$. Let $d\{\sigma_i = x\}$ be the denotation which is identical to d, except with $d.\sigma_i = x$; $d\{r_i = x\}$, $d\{b_i = x\}$, and $d\{c_i = x\}$ are defined analogously. We also define a project operation for denotations: $\langle\!\langle A; \sigma \rangle\!\rangle[\mathbf{i}] \stackrel{\text{def}}{=} \langle\!\langle \{\mathbf{a_i} : \mathbf{a} \in A\}; \sigma_\mathbf{i} \rangle\!\rangle$. Extending this notation further, we use $\emptyset$ to denote the indices of the *non-initial columns with empty stores* ($i > 1$ such that $d.\sigma_i = \emptyset$). We can then use $d[-\emptyset]$ to represent projecting away the non-initial columns with empty stores. For the denotation d in Figure 2.7, $d[1]$ keeps column 1, $d[-\emptyset]$ keeps both columns, and $d[2, -2]$ swaps the two columns.

In basic DCS, denotations are sets of tuples, which works quite well for representing the semantics of wh-questions such as *What states border Texas?* But what about polar questions such as *Does Louisiana border Texas?* The denotation should be a simple boolean value, which basic DCS does not represent explicitly. Using our new denotations, we can represent boolean values explicitly using zero-column structures: *true* corresponds to a singleton set containing just the empty array ($d_{\text{T}} = \langle\!\langle \{[\,]\} \rangle\!\rangle$) and *false* is the empty set ($d_{\text{F}} = \langle\!\langle \emptyset \rangle\!\rangle$).

Having described denotations as n-column structures, we now give the formal mapping from DCS tree to these structures. As in basic DCS, this mapping is defined recursively over the structure of the tree. We have a recurrence for each case (the first line is the base case,

and each of the others handles a different edge relation):

$$[\![\langle p \rangle]\!]_w = \langle\!\langle \{[v] : v \in w(p)\}; \emptyset \rangle\!\rangle, \quad \text{[base case]} \tag{2.19}$$

$$[\![\langle p; \mathbf{e}; {}^{j}_{j'} : c \rangle]\!]_w = [\![\langle p; \mathbf{e} \rangle]\!]_w \bowtie^{-\emptyset}_{j,j'} [\![c]\!]_w, \quad \text{[join]} \tag{2.20}$$

$$[\![\langle p; \mathbf{e}; \Sigma : c \rangle]\!]_w = [\![\langle p; \mathbf{e} \rangle]\!]_w \bowtie^{-\emptyset}_{*,*} \Sigma\left([\![c]\!]_w\right), \quad \text{[aggregate]} \tag{2.21}$$

$$[\![\langle p; \mathbf{e}; \mathrm{X}_{\mathbf{i}} : c \rangle]\!]_w = [\![\langle p; \mathbf{e} \rangle]\!]_w \bowtie^{-\emptyset}_{*,*} \mathbf{X_i}([\![c]\!]_w), \quad \text{[execute]} \tag{2.22}$$

$$[\![\langle p; \mathbf{e}; \mathrm{E} : c \rangle]\!]_w = \mathbf{M}([\![\langle p; \mathbf{e} \rangle]\!]_w, \mathrm{E}, [\![c]\!]_w), \quad \text{[extract]} \tag{2.23}$$

$$[\![\langle p; \mathbf{e}; \mathrm{C} : c \rangle]\!]_w = \mathbf{M}([\![\langle p; \mathbf{e} \rangle]\!]_w, \mathrm{C}, [\![c]\!]_w), \quad \text{[compare]} \tag{2.24}$$

$$[\![\langle p; \mathrm{Q} : c; \mathbf{e} \rangle]\!]_w = \mathbf{M}([\![\langle p; \mathbf{e} \rangle]\!]_w, \mathrm{Q}, [\![c]\!]_w). \quad \text{[quantify]} \tag{2.25}$$

Note that these definitions depend on several operations ($\bowtie^{-\emptyset}_{\mathbf{j},\mathbf{j}'}, \Sigma, \mathbf{X_i}, \mathbf{M}$), which we have not defined yet. We will do so lazily as we walk through each of the cases.

Base Case (2.19) defines the denotation for a DCS tree z with a single node with predicate p. The denotation of z has one column whose arrays correspond to the tuples $w(p)$; the store for that column is empty.

2.5.2 Join Relations

(2.20) defines the recurrence for join relations. On the left-hand side, $\left\langle p; \mathbf{e}; {}^{j}_{j'} : c \right\rangle$ is a DCS tree with p at the root, a sequence of edges $\mathbf{e}$ followed by a final edge with relation ${}^{j}_{j'}$ connect to a child DCS tree c. On the right-hand side, we take the recursively computed denotation of $\langle p; \mathbf{e} \rangle$, the DCS tree without the final edge, and perform a *join-project-inactive* operation (notated $\bowtie^{-\emptyset}_{j,j'}$) with the denotation of the child DCS tree c.

The join-project-inactive operation joins the arrays of the two denotations (this is the core of the join operation in basic DCS—see (2.13)), and then projects away the non-initial empty columns:

$$\langle\!\langle A; \sigma \rangle\!\rangle \bowtie^{-\emptyset}_{\mathbf{j},\mathbf{j}'} \langle\!\langle A'; \sigma' \rangle\!\rangle = \langle\!\langle A''; \sigma + \sigma' \rangle\!\rangle[-\emptyset], \text{where} \tag{2.26}$$
$$A'' = \{\mathbf{a} + \mathbf{a}' : \mathbf{a} \in A, \mathbf{a}' \in A', a_{1\mathbf{j}} = a'_{1\mathbf{j}'}\}.$$

We concatenate all arrays $\mathbf{a} \in A$ with all arrays $\mathbf{a}' \in A'$ that satisfy the join condition $a_{1\mathbf{j}} = a'_{1\mathbf{j}'}$. The sequences of stores are simply concatenated ($\sigma + \sigma'$). Finally, any non-initial columns with empty stores are projected away by applying $\cdot[-\emptyset]$.

Note that the join works on column 1, the other columns are carried along for the ride. As another piece of convenient notation, we use $*$ to represent all components, so that $\bowtie^{-\emptyset}_{*,*}$ would impose the join condition that the entire tuple has to agree ($a_1 = a'_1$).

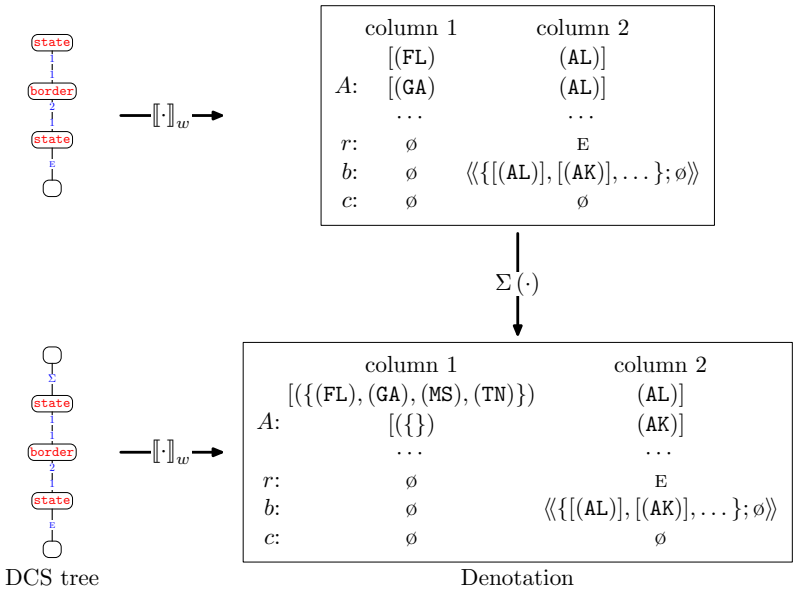


Figure 2.8: An example of applying the aggregate operation, which takes a denotation and aggregates the values in column 1 for every setting of the other columns. The base denotations (b) are used to put in $\{\}$ for values that do not appear in A. (in this example, AK, corresponding to the fact that Alaska does not border any states).

2.5.3 Aggregate Relations

(2.21) defines the recurrence for aggregate relations. Recall that in basic DCS, aggregate (2.16) simply takes the denotation (a set of tuples) and puts it into a set. Now, the denotation is not just a set, so we need to generalize this operation. Specifically, the aggregate operation applied to a denotation forms a set out of the tuples in the first column for each setting of the rest of the columns:

$$\begin{aligned}\Sigma(\langle\!\langle A; \sigma \rangle\!\rangle) &= \langle\!\langle A' \cup A''; \sigma \rangle\!\rangle, \\ A' &= \{[S(\mathbf{a}), a_2, \ldots, a_n] : \mathbf{a} \in A\}, \\ S(\mathbf{a}) &= \{a_1' : [a_1', a_2, \ldots, a_n] \in A\}, \\ A'' &= \{[\emptyset, a_2, \ldots, a_n] : \forall i \in \{2, \ldots, n\}, [a_i] \in \sigma_i.b.A[1], \neg\exists a_1, \mathbf{a} \in A\}.\end{aligned} \tag{2.27}$$

The aggregate operation takes the set of arrays A and produces two sets of arrays, A' and A'', which are unioned (note that the stores do not change). The set A' is the one that first comes to mind: For every setting of $a_2, \ldots, a_n$, we construct $S(\mathbf{a})$, the set of tuples a_1' in the first column which co-occur with $a_2, \ldots, a_n$ in A.

However, there is another case: what happens to settings of $a_2, \ldots, a_n$ that do not co-occur with any value of a_1' in A? Then, $S(\mathbf{a}) = \emptyset$, but note that A' by construction will not

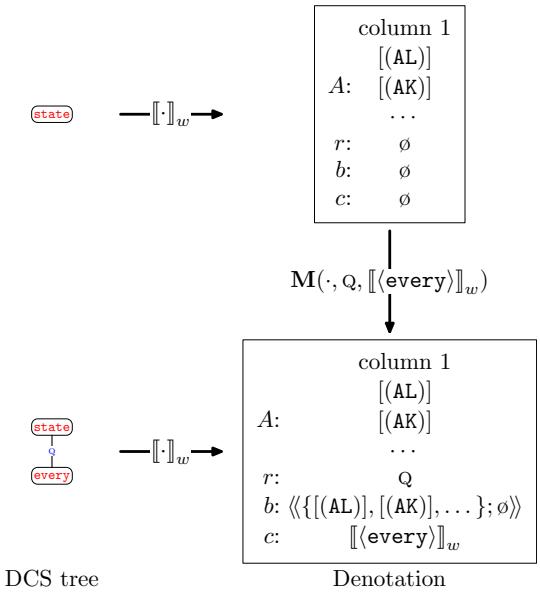


Figure 2.9: An example of applying the mark operation, which takes a denotation and modifies the store of the column 1. This information is used by other operations such as aggregate and execute.

have the desired array $[\emptyset, a_2, \dots, a_n]$. As a concrete example, suppose $A = \emptyset$ and we have one column ($n = 1$). Then $A' = \emptyset$, rather than the desired $\{[\emptyset]\}$.

Fixing this problem is slightly tricky. There are an infinite number of $a_2, \dots, a_n$ which do not co-occur with any a'_1 in A, so for which ones do we actually include $[\emptyset, a_2, \dots, a_n]$? Certainly, the answer to this question cannot come from A, so it must come from the stores. In particular, for each column $i \in \{2, \dots, n\}$, we have conveniently stored a base denotation $\sigma_i.b$. We consider any a_i that occurs in column 1 of the arrays of this base denotation ($[a_i] \in \sigma_i.b.A[1]$). For this $a_2, \dots, a_n$, we include $[\emptyset, a_2, \dots, a_n]$ in A'' as long as $a_2, \dots, a_n$ does not co-occur with any a_1. An example is given in Figure 2.8.

Now, the reason for storing base denotations is partially revealed. The arrays represent feasible values of a CSP and thus can only contain positive information. When we aggregate, we need to access possibly empty sets of feasible values—a kind of negative information, which can only be recovered from the base denotations.

2.5.4 Mark Relations

(2.23), (2.24), and (2.25) each processes a different mark relation. We define a general mark operation, $\mathbf{M}(d, r, c)$ which takes a denotation d, a mark relation $r \in \{\text{E}, \text{Q}, \text{C}\}$ and a child

denotation c, and sets the store of d in column 1 to be (r, d, c):

$$\mathbf{M}(d, r, c) = d\{r_1 = r, b_1 = d, c_1 = c\}. \tag{2.28}$$

The base denotation of the first column b_1 is set to the current denotation d. This, in some sense, creates a snapshot of the current denotation. Figure 2.9 shows an example of the mark operation.

2.5.5 Execute Relations

(2.22) defines the denotation of a DCS tree where the last edge of the root is an execute relation. Similar to the aggregate case (2.21), we recurse on the DCS tree without the last edge ($\langle p; \mathbf{e}\rangle$) and then join it to the result of applying the execute operation $\mathbf{X_i}$ to the denotation of the child ($[\![c]\!]_w$).

The execute operation $\mathbf{X_i}$ is the most intricate part of DCS and is what does the heavy lifting. The operation is parametrized by a sequence of distinct indices $\mathbf{i}$ which specifies the order in which the columns should be processed. Specifically, $\mathbf{i}$ indexes into the subsequence of columns with non-empty stores. We then process this subsequence of columns in reverse order, where processing a column means performing some operations depending on the stored relation in that column. For example, suppose that columns 2 and 3 are the only non-empty columns. Then $\mathbf{X}_{12}$ processes column 3 before column 2. On the other hand, $\mathbf{X}_{21}$ processes column 2 before column 3. For the double quantifier example, If each column stores a quantifier example (Figure 2.6(c)), these lead to the narrow and wide readings, respectively. We will define the execute operation $\mathbf{X}_i$ for a single column i. There are three distinct cases, depending on the relation stored in column i:

Extraction

For a denotation d with the extract relation E in column i, executing $\mathbf{X}_i(d)$ involves three steps: (i) moving column i to before column 1 ($\cdot[i, -i]$), (ii) projecting away non-initial empty columns ($\cdot[-\emptyset]$), and (iii) removing the store ($\cdot\{\sigma_1 = \emptyset\}$):

$$\mathbf{X}_i(d) = d[i, -i][-\emptyset]\{\sigma_1 = \emptyset\} \quad \text{if } d.r_i = \textsc{e}. \tag{2.29}$$

An example is given in Figure 2.10. There are two main uses of extraction:

1. By default, the denotation of a DCS tree is the set of feasible values of the root node (which occupies column 1). To return the set of feasible values of another node, we mark that node with E. Upon execution, the feasible values of that node move into column 1. See Figure 2.6(a) for an example.

2. Unmarked nodes are existentially quantified and have narrower scope than all marked nodes. Therefore, we can make a node x have wider scope than another node y by

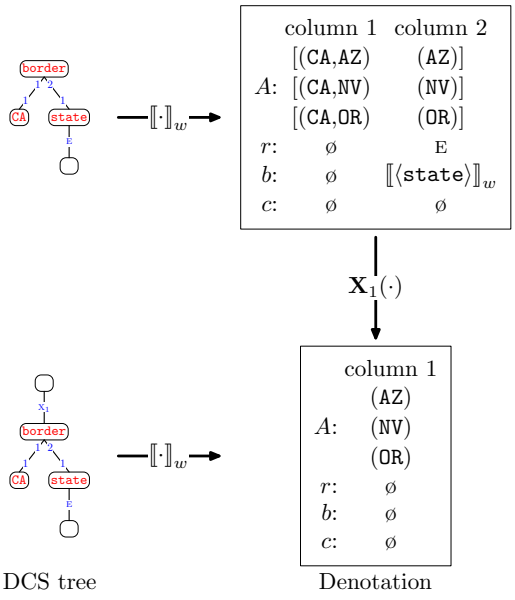


Figure 2.10: An example of applying the execute operation on column i with the extract relation E.

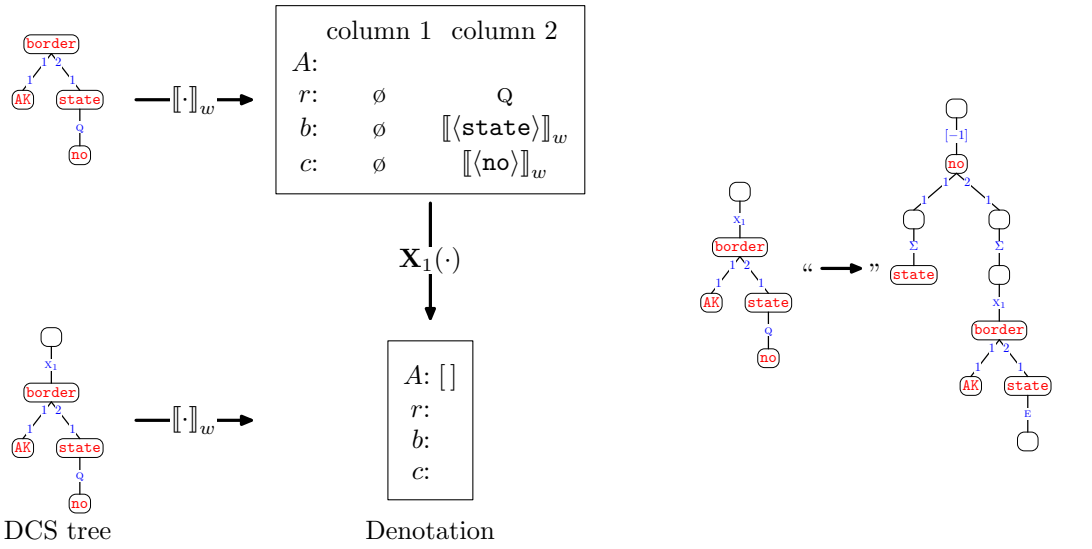
marking x (with E) and executing y before x (see Figure 2.6(d,e) for examples). The extract relation E (in fact, any mark relation) signifies that we want to control the scope of a node, and the execute relation allows us to set that scope.

Generalized Quantification

Generalized quantifiers (including negation) are predicates on two sets, a *restrictor* A and a *nuclear scope* B. For example,

$$w(\texttt{some}) = \{(A, B) : A \cap B > 0\}, \tag{2.30}$$
$$w(\texttt{every}) = \{(A, B) : A \subset B\}, \tag{2.31}$$
$$w(\texttt{no}) = \{(A, B) : A \cap B = \emptyset\}, \tag{2.32}$$
$$w(\texttt{most}) = \{(A, B) : |A \cap B| > \frac{1}{2}|A|\}. \tag{2.33}$$

We think of the quantifier as a modifier which always appears as the child of a Q relation; the restrictor is the parent. For example, in Figure 2.6(b), `no` corresponds to the quantifier and `state` corresponds to the restrictor. The nuclear scope should be the set of all states that Alaska borders. More generally, the nuclear scope is the set of feasible values of the restrictor node with respect to the CSP that includes all nodes between the mark and execute relations.



(a) Execute a quantify relation Q

(b) Execute “expands the DCS tree”

Figure 2.11: (a) An example of applying the execute operation on column i with the quantify relation Q. Before executing, note that $A = \{\}$ (because Alaska does not border any states). The restrictor (A) is the set of all states, and the nuclear scope (B) is empty. Since the pair (A, B) does exists in $w(\texttt{no})$, the final denotation, is $\langle\!\langle\{[\,]\}\rangle\!\rangle$ (which represents true). (b) Although the execute operation actually works on the denotation, think of it in terms of expanding the DCS tree. We introduce an extra projection relation $[-1]$, which projects away the first column of the child subtree's denotation.

The restrictor is also the set of feasible values of the restrictor node, but with respect to the CSP corresponding to the subtree rooted at that node.[5]

We implement generalized quantifiers as follows: Let d be a denotation and suppose we are executing column i. We first construct a denotation for the restrictor d_A and a denotation for the nuclear scope d_B. For the restrictor, we take the base denotation in column i ($d.b_i$)—remember that the base denotation represents a snapshot of the restrictor node before the nuclear scope constraints are added. For the nuclear scope, we take the complete denotation d (which includes the nuclear scope constraints) and extract column i ($d[i, -i][-\emptyset]\{\sigma_1 = \emptyset\}$—see (2.29)). We then construct d_A and d_B by applying the aggregate

[5] Defined this way, we can only handle conservative quantifiers, since the nuclear scope will always be a subset of the restrictor. This design decision is inspired by DRT, where it provides a way of modeling donkey anaphora. We are not treating anaphora in this work, but we can handle it by allowing pronouns in the nuclear scope to create anaphoric edges into nodes in the restrictor. These constraints naturally propagate through the nuclear scope's CSP without affecting the restrictor.

operation to each. Finally, we join these sets with the quantifier denotation, stored in $d.c_i$:

$$\mathbf{X}_i(d) = \left(\left(d.c_i \bowtie_{1,1}^{-\emptyset} d_A\right) \bowtie_{2,1}^{-\emptyset} d_B\right)[-1] \quad \text{if } d.r_i = \text{Q}, \text{where} \tag{2.34}$$

$$d_A = \Sigma\left(d.b_i\right), \tag{2.35}$$

$$d_B = \Sigma\left(d[i,-i][-\emptyset]\{\sigma_1 = \emptyset\}\right). \tag{2.36}$$

When there is one quantifier, think of the execute relation as performing a syntactic rewriting operation, as shown in Figure 2.11(b). For more complex cases, we must defer to (2.34).

Figure 2.6(c) shows an example with two interacting quantifiers. The denotation of the DCS tree before execution is the same in both readings, as shown in Figure 2.12. The quantifier scope ambiguity is resolved by the choice of execute relation: X_{12} gives the narrow reading, X_{21} gives the wide reading.

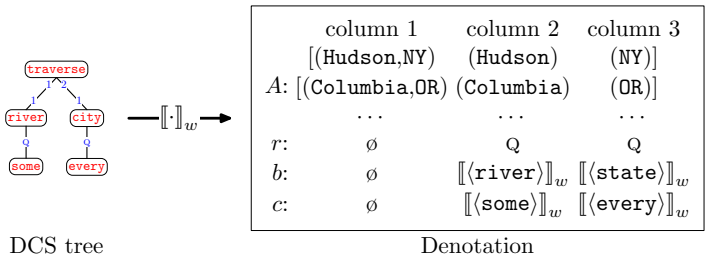


Figure 2.12: Denotation of Figure 2.6(c) before the execute relation is applied.

Figure 2.6(d) shows how extraction and quantification work together. First, the `no` quantifier is processed for each `city`, which is an unprocessed marked node. Here, the extract relation is a technical trick to give `city` wider scope.

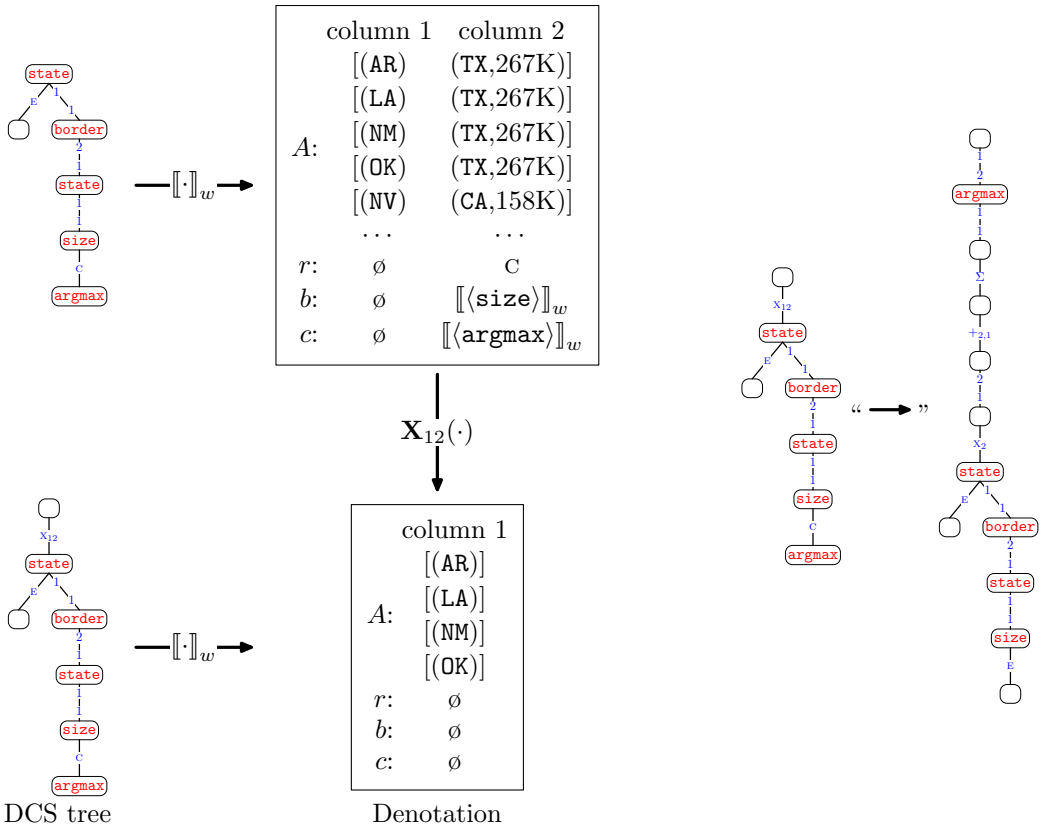
Comparatives and Superlatives

Comparative and superlative constructions involve comparing entities, and for this, we rely on a set S of entity-degree pairs (x, y), where x is an entity and y is a numeric degree. Recall that we can treat S as a function, which maps an entity x to the set of degrees $S(x)$ associated with x. Note that this set can contain multiple degrees. For example, in the relative reading of *state bordering the largest state*, we would have a degree for the size of each neighboring state.

Superlatives use the `argmax` and `argmin` predicates, which are defined in Section 2.3. Comparatives use the `more` and `less` predicates: $w(\texttt{more})$ contains triples (S, x, y), where x is "more than" y as measured by S; $w(\texttt{less})$ is defined analogously:

$$w(\texttt{more}) = \{(S, x, y) : \max S(x) > \max S(y)\}, \tag{2.37}$$

$$w(\texttt{less}) = \{(S, x, y) : \min S(x) < \min S(y)\}. \tag{2.38}$$



(a) Execute a compare relation C

(b) Execute “expands the DCS tree”

Figure 2.13: (a) Executing a compare relation C for an example superlative construction (relative reading of *state bordering the largest state* from Figure 2.6(h)). Before executing, column 1 contains the entity to compare, and column 2 contains the degree information, of which only the second component is relevant. After executing, the resulting denotation contains a single column with only the entities that obtain the highest degree (in this case, the states that border Texas) (b) For this example, think of the execute operation as expanding the original DCS tree, although the execute operation actually works on the denotation, not the DCS tree. The expanded DCS tree has the same denotation as the original DCS tree, and syntactically captures the essence of the execute-compare operation. Going through the relations of the expanded DCS tree from bottom to top: The X_2 relation swaps columns 1 and 2; the join relation keeps only the second component ($(\texttt{TX}, 267K)$ becomes $(267K)$); $+_{2,1}$ concatenates columns 2 and 1 ($[(267K), (\texttt{AR})]$ becomes $[(\texttt{AR}, 267K)]$); Σ aggregates these tuples into a set; `argmax` operates on this set and returns the elements.

We use the same mark relation C for both comparative and superlative constructions. In terms of the DCS tree, there are three key parts: (i) the root x, which corresponds to the

entity to be compared, (ii) the child c of a C relation, which corresponds to the comparative or superlative predicate, and (iii) c's parent p, which contains the "degree information" (which will be described later) used for comparison. We assume that the root is marked (usually with a relation E). This forces us to compute a comparison degree for each value of the root node. In terms of the denotation d corresponding to the DCS tree prior to execution, the entity to be compared occurs in column 1 of the arrays $d.A$, the degree information occurs in column i of the arrays $d.A$, and the denotation of the comparative or superlative predicate itself is the child denotation at column i ($d.c_i$).

First, we define a concatenating function $+_{\mathbf{i}}(d)$, which combines the columns $\mathbf{i}$ of d by concatenating the corresponding tuples of each array in $d.A$:

$$\begin{aligned} +_{\mathbf{i}}(\langle\!\langle A;\sigma\rangle\!\rangle) &= \langle\!\langle A';\sigma'\rangle\!\rangle, \text{where} \\ A' &= \{\mathbf{a}_{(1\ldots i_1)}\backslash\mathbf{i} + [a_{i_1} + \cdots + a_{i_{|\mathbf{i}|}}] + \mathbf{a}_{(i_1\ldots n)\backslash\mathbf{i}} : \mathbf{a} \in A\} \\ \sigma' &= \sigma_{(1\ldots i_1)}\backslash\mathbf{i} + [\sigma_{i_1}] + \sigma_{(i_1\ldots n)\backslash\mathbf{i}}. \end{aligned} \tag{2.39}$$

Note that the store of column i_1 is kept and the others are discarded. As an example:

$$+_{2,1}(\langle\!\langle\{[(1),(2),(3)],[(4),(5),(6)]\};\sigma_1,\sigma_2,\sigma_3\rangle\!\rangle) = \langle\!\langle\{[(2,1),(3)],[(5,4),(6)]\};\sigma_2,\sigma_3\rangle\!\rangle. \tag{2.40}$$

We first create a denotation d' where column i, which contains the degree information, is extracted to column 1 (and thus column 2 corresponds to the entity to be compared). Next, we create a denotation d_S whose column 1 contains a set of entity-degree pairs. There are two types of degree information:

1. Suppose the degree information has arity 2 ($\textsc{Arity}(d.A[i]) = 2$). This occurs, for example, in *most populous city* (see Figure 2.6(f)), where column i is the `population` node. In this case, we simply set the degree to the second component of `population` by projection ($[\![\langle\emptyset\rangle]\!]_w \bowtie_{1,2}^{-\emptyset} d'$). Now columns 1 and 2 contain the degrees and entities, respectively. We concatenate columns 2 and 1 ($+_{2,1}(\cdot)$) and aggregate to produce a denotation d_S which contains the set of entity-degree pairs in column 1.

2. Suppose the degree information has arity 1 ($\textsc{Arity}(d.A[i]) = 1$). This occurs, for example, in *state bordering the most states* (see Figure 2.6(e)), where column i is the lower marked `state` node. In this case, the degree of an entity from column 2 is the number of different values that column 1 can take. To compute this, aggregate the set of values ($\Sigma(d')$) and apply the `count` predicate. Now with the degrees and entities in columns 1 and 2, respectively, we concatenate the columns and aggregate again to obtain d_S.

Having constructed d_S, we simply apply the comparative/superlative predicate which has been patiently waiting in $d.c_i$. Finally, the store of d's column 1 was destroyed by the

concatenation operation $+_{2,1}((\,)\,\cdot)$, so we must restore it with $\cdot\{\sigma_1 = d.\sigma_1\}$. The complete operation is as follows:

$$\mathbf{X}_i(d) = \left(\llbracket\langle\emptyset\rangle\rrbracket_w \bowtie_{1,2}^{-\emptyset} \left(d.c_i \bowtie_{1,1}^{-\emptyset} d_S\right)\right)\{\sigma_1 = d.\sigma_1\} \text{ if } d.\sigma_i = \textsc{c}, d.\sigma_1 \neq \emptyset, \text{ where} \tag{2.41}$$

$$d_S = \begin{cases} \Sigma\left(+_{2,1}\left(\llbracket\langle\emptyset\rangle\rrbracket_w \bowtie_{1,2}^{-\emptyset} d'\right)\right) & \text{if } \textsc{Arity}(d.A[i]) = 2 \\ \Sigma\left(+_{2,1}\left(\llbracket\langle\emptyset\rangle\rrbracket_w \bowtie_{1,2}^{-\emptyset} \left(\llbracket\langle\texttt{count}\rangle\rrbracket_w \bowtie_{1,1}^{-\emptyset} \Sigma\left(d'\right)\right)\right)\right) & \text{if } \textsc{Arity}(d.A[i]) = 1, \end{cases} \tag{2.42}$$

$$d' = d[i, -i][-\emptyset]\{\sigma_1 = \emptyset\}. \tag{2.43}$$

An example of executing the C relation is shown in Figure 2.13(a). As with executing a Q relation, for simple cases, we can think of executing a C relation as expanding a DCS tree, as shown in Figure 2.13(b).

Figure 2.6(e) and Figure 2.6(f) show examples of superlative constructions with the arity 1 and arity 2 types of degree information, respectively. Figure 2.6(g) shows an example of an comparative construction. Comparatives and superlatives both use the exact same machinery, differing only in the predicate: `argmax` versus $\langle\texttt{more}; {}^{3}_{1}\texttt{:TX}\rangle$ (*more than Texas*). But both predicates have the same template behavior: Each takes a set of entity-degree pairs and returns any entity satisfying some property. For `argmax`, the property is obtaining the highest degree; for `more`, it is having a degree higher than a threshold. We can handle generalized superlatives (*the five largest* or *the fifth largest* or *the 5% largest*) as well by swapping in a different predicate; the execution mechanisms defined in (2.41) remain the same.

We saw that the mark-execute machinery allowed decisions regarding quantifier scope to made in a clean and modular fashion. Superlatives also have scope ambiguities in the form of absolute versus relative readings. Consider the example in Figure 2.6(g). In the absolute reading, we first compute the superlative in a narrow scope (*the largest state* is Alaska), and then connect it with the rest of the phrase, resulting in the empty set (since no states border Alaska). In the relative reading, we consider the first *state* as the entity we want to compare, and its degree is the size of a neighboring state. In this case, the lower `state` node cannot be set to Alaska because there are no states bordering it. The result is therefore any state that borders Texas (the largest state that does have neighbors). The two DCS trees in Figure 2.6(g) show that we can naturally account for this form of superlative ambiguity based on where the scope-determining execute relation is placed without drastically changing the underlying tree structure.

Remarks All these issues are not specific to DCS; every serious semantic formalism must address them as well. Not surprisingly then, the mark-execute construct bears some resemblance to other mechanisms that operate on categorial grammar and lambda calculus, such as quantifier raising, Montague's quantifying in, Cooper storage, and Carpenter's scoping constructor (Carpenter, 1998). Very broadly speaking, these mechanisms delay application of the divergent element (usually a quantifier), "marking" its spot with a dummy pronoun (as in Montague's quantifying in) or in a store (as in Cooper storage), and then "executing"

the quantifier at a later point in the derivation. One subtle but important difference between mark-execute in DCS and the others is that a DCS tree (which contains the mark and execute relations) is the final logical form, and all the action happens in the computing of the denotation of this logical form. In more traditional approaches, the action happens in the construction mechanism for building the logical form; the actually logical form produced at the end of the day is quite simple. In other words, we have pushed the inevitable complexity from the construction mechanism into the semantics of the logical from. This refactoring is important because we want our construction mechanism to focus not on linguistic issues, but on purely computational and statistical ones, which ultimately determine the success of our system.

2.6 Construction Mechanism

We have thus far defined the syntax (Section 2.2) and semantics (Section 2.5) of DCS trees, but we have only vaguely hinted at how these DCS trees might be connected to natural language utterances by appealing to idealized examples. In this section, we formally define the construction mechanism for DCS, which takes an utterance $\mathbf{x}$ and produces a set of DCS trees $\mathcal{Z}_L(\mathbf{x})$.

Since we motivated DCS trees based on dependency syntax, it might be tempting to take a dependency parse tree of the utterance, replace the words with predicates, and attach some relations on the edges to produce a DCS tree. To a first-order approximation, this is what we will do, but we need to be a bit more flexible for several reasons: (i) some nodes in the DCS tree do not have predicates (e.g., children of a E relation or parent of an $\text{X}_\mathbf{i}$ relation); (ii) nodes have predicates that do not correspond to words (e.g., in *California cities*, there is a implicit `loc` predicate that bridges `CA` and `city`); (iii) some words might not correspond to any predicates in our world (e.g., *please*); and (iv) the DCS tree might not always be aligned with the syntactic structure depending on which syntactic formalism one ascribes to. While syntax was the inspiration for the DCS formalism, we will not actually use it in construction.

It is also worth stressing the purpose of the construction mechanism. In linguistics, the purpose of the construction mechanism is to try to generate the exact set of valid logical forms for a sentence. We view the construction mechanism instead as simply a way of creating a set of candidate logical forms. A separate step defines a distribution over this set to favor certain logical forms over others. The construction mechanism should therefore overapproximate the set of logical forms. Settling for an overapproximation allows us to simplify the construction mechanism.

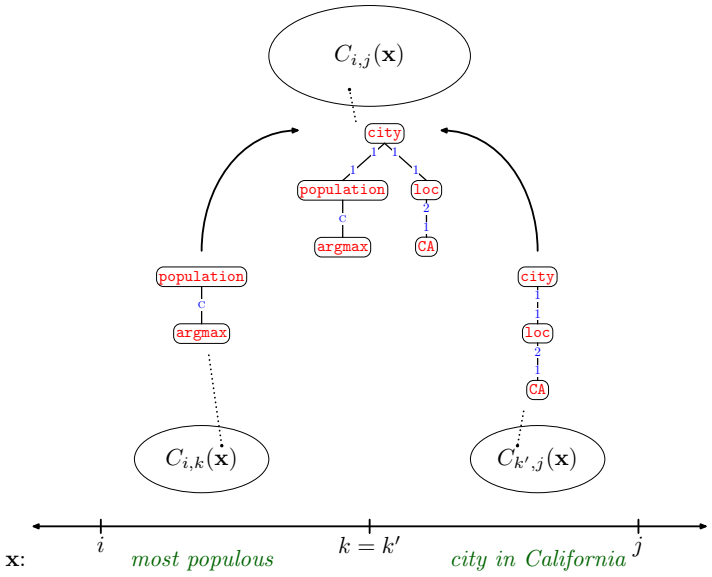


Figure 2.14: Shows an example of the recursive construction of $C_{i,j}(\mathbf{x})$, a set of DCS trees for span $i..j$.

2.6.1 Lexical Triggers

The construction mechanism assumes a fixed set of *lexical triggers* L. Each trigger is a pair $(\mathbf{s}, p)$, where $\mathbf{s}$ is a sequence of words (usually one) and p is a predicate (e.g., $\mathbf{s} =$ *California* and $p =$ `CA`). We use $L(\mathbf{s})$ to denote the set of predicates p triggered by $\mathbf{s}$ ($(\mathbf{s}, p) \in L$). We also define a set of *trace predicates*, denoted by $L(\epsilon)$, which can be introduced without an overt lexical trigger.

It is important to think of the lexical triggers L not as pinning down the precise predicate for each word. For example, L might contain $\{(\textit{city}, \texttt{city}), (\textit{city}, \texttt{state}), (\textit{city}, \texttt{river}), \dots\}$. Section 4.1.3 describes the lexical triggers that we use in our experiments.

2.6.2 Recursive Construction of DCS Trees

Given a set of lexical triggers L, we will now describe a recursive mechanism for mapping an utterance $\mathbf{x} = (x_1, \dots, x_n)$ to $\mathcal{Z}_L(\mathbf{x})$, a set of candidate DCS trees for $\mathbf{x}$. The basic approach is reminiscent of projective labeled dependency parsing: For each span $i..j$ of the utterance, we build a set of trees $C_{i,j}(\mathbf{x})$. The set of trees for the span $0..n$ is the final result:

$$\mathcal{Z}_L(\mathbf{x}) = C_{0,n}(\mathbf{x}). \tag{2.44}$$

Each set of DCS trees $C_{i,j}(\mathbf{x})$ is constructed recursively by combining the trees of its subspans $C_{i,k}(\mathbf{x})$ and $C_{k',j}(\mathbf{x})$ for each pair of split points k, k' (words between k and k'

are ignored). These combinations are then augmented via a function A and filtered via a function F; these functions will be specified later. Formally, $C_{i,j}(\mathbf{x})$ is defined recursively as follows:

$$C_{i,j}(\mathbf{x}) = F\Big(A\Big(\{\langle p\rangle_{i..j} : p \in L(\mathbf{x}_{i+1..j})\} \cup \bigcup_{\substack{i\le k\le k'<j \\ a\in C_{i,k}(\mathbf{x}) \\ b\in C_{k',j}(\mathbf{x})}} T_1(a,b)\Big)\Big). \tag{2.45}$$

This recurrence has two parts:

- The base case: we take the phrase (sequence of words) over span $i..j$ and look up the set of predicates p in the set of lexical triggers. For each predicate, we construct a one-node DCS tree. We also extend the definition of DCS trees in Section 2.2 to allow each node to store the indices of the span $i..j$ that triggered the predicate at that node; this is denoted by $\langle p\rangle_{i..j}$. This span information will be useful in Section 3.1.1, where we will need to talk about how an utterance $\mathbf{x}$ is aligned with a DCS tree z.
- The recursive case: $T_1(a,b)$, which we will define shortly, that takes two DCS trees, a and b, and returns a set of new DCS trees formed by combining a and b. Figure 2.14 shows this recurrence graphically.

We now focus on how to combine two DCS trees. Define $T_d(a,b)$ as the set of resulting DCS trees by making either a or b the root and connecting the other via a chain of relations and at most d trace predicates:

$$T_d(a,b) = T_d^{\searrow}(a,b) \cup T_d^{\swarrow}(b,a), \tag{2.46}$$

Here, $T_d^{\searrow}(a,b)$ is the set of DCS trees where a is the root; for $T_d^{\swarrow}(a,b)$, b is the root. The former is defined recursively as follows:

$$\begin{aligned} T_0^{\searrow}(a,b) &= \emptyset, \\ T_d^{\searrow}(a,b) &= \bigcup_{\substack{r\in\mathcal{R} \\ p\in L(\epsilon)}} \{\langle a; r:b\rangle, \langle a; r:\langle\Sigma:b\rangle\rangle\} \cup T_{d-1}^{\searrow}(a, \langle p; r:b\rangle). \end{aligned} \tag{2.47}$$

First, we consider all possible relations $r \in \mathcal{R}$ and try putting r between a and b ($\langle a; r:b\rangle$), possibly with an additional aggregate relation ($\langle a; r:\langle\Sigma:b\rangle\rangle$). Of course, $\mathcal{R}$ contains an infinite number of join and execute relations, but only a small finite number of them make sense: we consider join relations $\overset{j}{j'}$ only for $j \in \{1,\ldots,\textsc{Arity}(a.p)\}$ and $j' \in \{1,\ldots,\textsc{Arity}(b.p)\}$, and execute relations $\mathrm{x}_{\mathbf{i}}$ for which $\mathbf{i}$ does not contain indices larger than the number of columns of $[\![b]\!]_w$. Next, we further consider all possible trace predicates $p \in L(\epsilon)$, and recursively try to connect a with the intermediate $\langle p; r:b\rangle$, now allowing $d-1$ additional predicates. See Figure 2.15 for an example. In the other direction, $T_d^{\swarrow}$ is defined similarly:

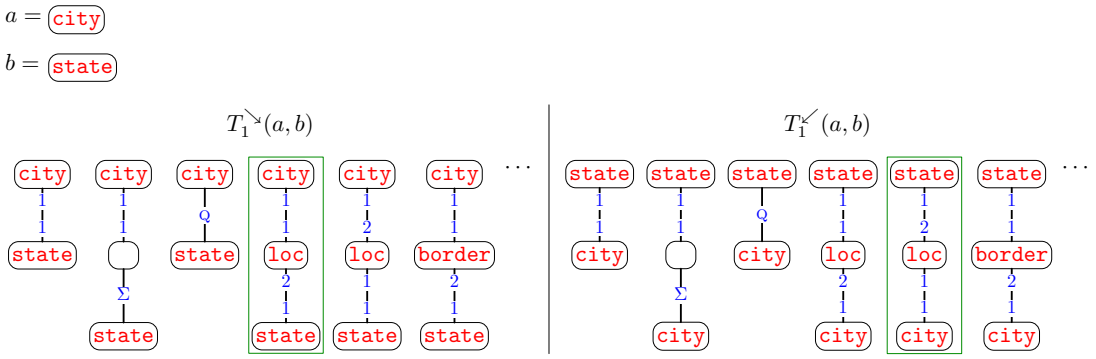


Figure 2.15: Given two DCS trees, a and b, $T_1^{\searrow}(a,b)$ and $T_1^{\swarrow}(a,b)$ are the two sets of DCS trees formed by combining a and b with a at the root and b at the root, respectively; one trace predicate can be inserted in between. In this example, the DCS trees which survive filtering (Section 2.6.3) are shown.

$$T_0^{\swarrow}(a,b) = \emptyset, \tag{2.48}$$
$$T_d^{\swarrow}(a,b) = \bigcup_{\substack{r\in\mathcal{R}\\ p\in L(\epsilon)}} \{\langle b.p; r:a; b.\mathbf{e}\rangle, \langle b.p; r:\langle\Sigma:a\rangle; b.\mathbf{e}\rangle\} \cup T_{d-1}^{\searrow}(a, \langle p; r:b\rangle).$$

Inserting trace predicates allows us to build logical forms with more predicates than what is explicitly triggered by the words. This ability is useful for several reasons. Sometimes, there actually is a predicate not overtly expressed, especially in noun compounds (e.g., *California cities*). For semantically light words such as prepositions (e.g., *for*) it is difficult to enumerate all the possible predicates that it might trigger; it is computationally simpler to try to insert trace predicates. We can even omit lexical triggers for transitive verbs such as *border* because the corresponding predicate `border` can be just inserted as trace predicate.

The function $T_1(a,b)$ connects two DCS trees via a path of relations and trace predicates. The augmentation function A adds additional relations (specifically, $\textsc{e}$ and/or $\textsc{x}_\mathbf{i}$) on a single DCS tree:

$$A(Z) = \bigcup_{\substack{z\in Z\\ \textsc{x}_\mathbf{i}\in\mathcal{R}}} \{z, \langle z; \textsc{e}:\langle\emptyset\rangle\rangle, \langle \textsc{x}_\mathbf{i}:z\rangle, \langle \textsc{x}_\mathbf{i}:\langle z; \textsc{e}:\langle\emptyset\rangle\rangle\rangle\}, \tag{2.49}$$

2.6.3 Filtering using Abstract Interpretation

The current construction procedure is extremely permissive and generates many DCS trees which are obviously wrong—for example, $\langle \texttt{state}; \frac{1}{1}:\langle \texttt{>}; \frac{2}{1}\langle 3\rangle\rangle\rangle$, which tries to compare a state with the number 3. There is nothing syntactically wrong this expression: its denotation will simply be empty (with respect to the world). But semantically, this DCS tree is anomalous.

We cannot simply just discard DCS trees with empty denotations, because we would incorrectly rule out $\langle \texttt{state}; {}^{1}_{1} : \langle \texttt{border}; {}^{2}_{1} \langle \texttt{AK} \rangle \rangle \rangle$. The difference here is that even though the denotation is empty in this world, it is possible that it might not be empty in a different world where history and geology took another turn, whereas it is simply impossible to compare cities and numbers.

Now let us quickly flesh out this intuition before falling into a philosophical discussion about possible worlds. Given a world w, we define an abstract world $\alpha(w)$, to be described shortly. We compute the denotation of a DCS tree z with respect to this abstract world. If at any point in the computation we create an empty denotation, we judge z to be impossible and throw it away. The filtering function F is defined as follows:[6]

$$F(Z) = \{z \in Z : \forall z' \text{ subtree of z }, [\![z']\!]_{\alpha(w)}.A \neq \emptyset\}. \tag{2.50}$$

Now we need to define the abstract world $\alpha(w)$. The intuition is to map concrete values to abstract values: $3 : \texttt{length}$ becomes $\texttt{length}$, $\mathit{Oregon} : \texttt{state}$ becomes $* : \texttt{state}$, and in general, primitive value $x : t$ becomes $* : t$. We perform abstraction on tuples componentwise, so that $(\mathit{Oregon} : \texttt{state}, 3 : \texttt{length})$ becomes $(* : \texttt{state}, * : \texttt{length})$. Our abstraction of sets is slightly more complex: the empty set maps to the empty set, a set containing values all with the same abstract value a maps to $\{a\}$, and a set containing values with more than one abstract value maps to a $\{\text{MIXED}\}$. Finally, a world maps each predicate onto a set of (concrete) tuples; the corresponding abstract world maps each predicate onto the set of abstract tuples. Formally, the abstraction function is defined as follows:

$$\alpha(x : t) = * : t, \qquad \text{[primitive values]} \tag{2.51}$$

$$\alpha((v_1, \ldots, v_n)) = (\alpha(v_1), \ldots, \alpha(v_n)), \qquad \text{[tuples]} \tag{2.52}$$

$$\alpha(A) = \begin{cases} \emptyset & \text{if } A = \emptyset, \\ \{\alpha(x) : x \in A\} & \text{if } |\{\alpha(x) : x \in A\}| = 1, \\ \{\text{MIXED}\} & \text{otherwise.} \end{cases} \qquad \text{[sets]} \tag{2.53}$$

$$\alpha(w) = \lambda p.\{\alpha(x) : x \in w(p)\}. \qquad \text{[worlds]} \tag{2.54}$$

As an example, the abstract world might look like this:

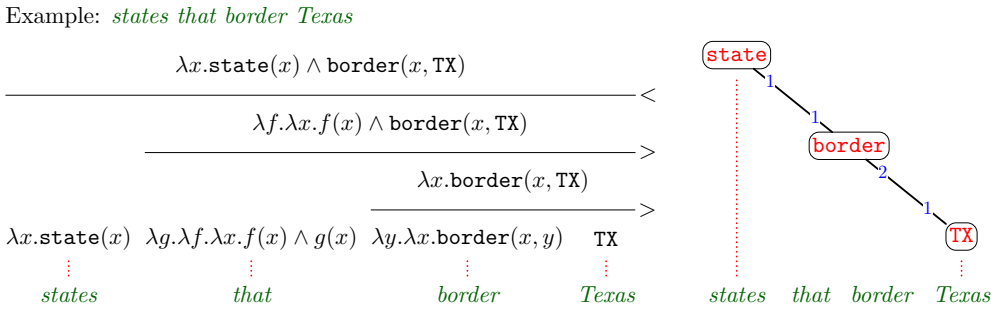
$$\alpha(w)(\texttt{>}) = \{(* : \texttt{number}, * : \texttt{number}, * : \texttt{number}), (* : \texttt{length}, * : \texttt{length}, * : \texttt{length}), \ldots\}, \tag{2.55}$$

$$\alpha(w)(\texttt{state}) = \{(* : \texttt{state})\}, \tag{2.56}$$

$$\alpha(w)(\texttt{AK}) = \{(* : \texttt{state})\}, \tag{2.57}$$

$$\alpha(w)(\texttt{border}) = \{(* : \texttt{state}, * : \texttt{state})\}. \tag{2.58}$$

[6] To further reduce the search space, F imposes a few additional constraints, e.g., limiting the number of columns to 2, and only allowing trace predicates between arity 1 predicates.



(a) CCG construction (b) DCS construction

Figure 2.16: Comparison between the construction mechanisms of CCG and DCS. There are three principal differences: First, in CCG, words are mapped onto a lambda calculus expression; in DCS, words are just mapped onto a predicate. Second, in CCG, lambda calculus expressions are built by combining (e.g., via function application) two smaller expressions; in DCS, trees are combined by inserting relations (and possibly other predicates between them). Third, in CCG, all words map to a logical expression; in DCS, only a small subset of words (e.g., *state* and `Texas`) map to predicates; the rest participate in features for scoring DCS trees.

Now returning our motivating example at the beginning of this section, we see that the bad DCS tree has an empty abstract denotation $[\![\langle \texttt{state}; \frac{1}{1}\!:\langle \texttt{>}; \frac{2}{1}\langle 3\rangle\rangle\rangle]\!]_{\alpha(w)} = \langle\langle \emptyset; \emptyset\rangle\rangle$. The good DCS tree has an non-empty abstract denotation: $[\![\langle \texttt{state}; \frac{1}{1}\!:\langle \texttt{border}; \frac{2}{1}\langle \texttt{AK}\rangle\rangle\rangle]\!]_{\alpha(w)} = \langle\langle \{(* : \texttt{state})\}; \emptyset\rangle\rangle$, as desired.

Remarks Computing denotations on an abstract world is called *abstract interpretation* (Cousot and Cousot, 1977) and is very powerful framework commonly used in the programming languages community. The idea is to obtain information about a program (in our case, a DCS tree) without running it concretely, but rather just by running it abstractly. It is closely related to type systems, but the type of abstractions one uses is often much richer than standard type systems.

2.6.4 Comparison with CCG

We now compare our construction mechanism with CCG (see Figure 2.16 for an example). The main difference is that our lexical triggers contain less information than a lexicon in a CCG. In CCG, the lexicon would have an entry such as

$$major \vdash \text{N}/\text{N} : \lambda f.\lambda x.\texttt{major}(x) \wedge f(x), \tag{2.59}$$

which gives detailed information about how this word should interact with its context. However, in DCS construction, each lexical trigger only has the minimal amount of information:

$$major \vdash \texttt{major}. \tag{2.60}$$

A lexical trigger specifies a pre-theoretic "meaning" of a word which does not commit to any formalisms. One advantage of this minimality is that lexical triggers could be easily obtained from non-expert supervision: One would only have to associate words with database table names (predicates).

In some sense, the DCS construction mechanism pushes the complexity out of the lexicon. In linguistics, this complexity usually would end up in the grammar, which would be undesirable. However, we do not have to fight this tradeoff, because the construction mechanism only produces an overapproximation, which means it is possible to have both a simple "lexicon" and a simple "grammar."

There is an important but subtle rationale for this design decision. During learning, we never just have one clean lexical entry per word (as is typically assumed in formal linguistics). Rather, there are often many possible lexical entries (and to handle disfluent utterances or utterances in free word-order languages, we might actually need many of them (Kwiatkowski et al., 2010, 2011)):

$$major \vdash \textsc{n} : \lambda x.\texttt{major}(x) \tag{2.61}$$
$$major \vdash \textsc{n}/\textsc{n} : \lambda f.\lambda x.\texttt{major}(x) \wedge f(x) \tag{2.62}$$
$$major \vdash \textsc{n}\backslash\textsc{n} : \lambda f.\lambda x.\texttt{major}(x) \wedge f(x) \tag{2.63}$$
$$\ldots \tag{2.64}$$

Now think of a DCS lexical trigger $major \vdash \texttt{major}$ as simply a *compact representation* for a set of CCG lexical entries. Furthermore, the choice of the lexical entry is made not at the initial lexical base case, but rather during the recursive construction by inserting relations between DCS subtrees. It is exactly at this point that the choice *can* be made, because after all, the choice is one that depends on context. The general principle is to compactly represent the indeterminacy until one can resolve it.

Type raising is a combinator in CCG that turns one logical form into another. It can be used to turn one entity into related entity (a kind of generalized metonymy). For example, Zettlemoyer and Collins (2007) used it to allow conversion from `Boston` to $\lambda x.\texttt{from}(x, \texttt{Boston})$. Type raising in CCG is analogous to inserting trace predicates in DCS, but there is an important distinction: Type raising is a unary operation and thus is somewhat unconstrained because it changes logical forms into new ones without regard for how they will interact with the context. Inserting trace predicates is a binary operation which is constrained by the two predicates that it is mediating. In the example, `from` would only be inserted to combine `Boston` with `flight`. This is another instance of the general principle of delaying uncertain decisions until there is more information.

Chapter 3

Learning

In Chapter 2, we defined DCS trees and a construction mechanism for producing a set of candidate DCS trees given an utterance. We now define a probability distribution over that set (Section 3.1) and an algorithm for estimating the parameters (Section 3.2). The number of candidate DCS trees grows exponentially, so we use beam search to control this growth. The final learning algorithm alternates between beam search and optimizing the parameters, leading to a natural bootstrapping procedure which integrates learning and search.

3.1 Semantic Parsing Model

The semantic parsing model specifies a conditional distribution over a set of candidate DCS trees $C(\mathbf{x})$ given an utterance $\mathbf{x}$. This distribution depends on a function $\phi(\mathbf{x}, z) \in \mathbb{R}^d$, which takes a $(\mathbf{x}, z)$ pair and extracts a set of local features (see Section 3.1.1 for a full specification). Associated with this feature vector is a parameter vector $\theta \in \mathbb{R}^d$. The inner product between the two vectors $\phi(\mathbf{x}, z)^\top \theta$ yields a numerical score, which intuitively measures the compatibility of the utterance $\mathbf{x}$ with the DCS tree z. We exponentiate the score and normalize over $C(\mathbf{x})$ to obtain a proper probability distribution:

$$p(z \mid \mathbf{x}; C, \theta) = \exp\{\phi(\mathbf{x}, z)^\top \theta - A(\theta; \mathbf{x}, C)\}, \tag{3.1}$$

$$A(\theta; \mathbf{x}, C) = \log \sum_{z \in C(\mathbf{x})} \exp\{\phi(\mathbf{x}, z)^\top \theta\}. \tag{3.2}$$

Here, $A(\theta; \mathbf{x}, C)$ is the log-partition function with respect to the candidate set function $C(\mathbf{x})$.

3.1.1 Features

We now define the feature vector $\phi(\mathbf{x}, z) \in \mathbb{R}^d$, the core part of the semantic parsing model. Each component $j = 1, \ldots, d$ of this vector is a feature, and $\phi(\mathbf{x}, z)_j$ is the number of times that feature occurs in $(\mathbf{x}, z)$. Rather than working with indices, we treat features as

symbols (e.g., TRIGGERPRED[*states*, `state`]). Each feature captures some property about $(\mathbf{x}, z)$ which abstracts away from the details of the specific instance and allow us to generalize to new instances that share common features.

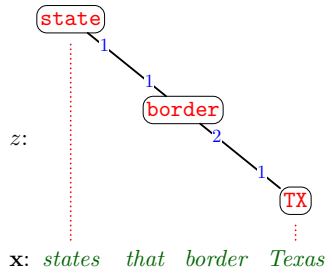
The features are organized into feature templates, where each feature template instantiates a set of features. Figure 3.1 shows all the feature templates for a concrete example. The feature templates are as follows:

- PREDHIT contains the single feature PREDHIT, which fires for each predicate in z.
- PRED contains features $\{\text{PRED}[\alpha(p)] : p \in \mathcal{P}\}$, each of which fires on $\alpha(p)$, the abstraction of predicate p, where

$$\alpha(p) = \begin{cases} *{:}t & \text{if } p = x{:}t \\ p & \text{otherwise.} \end{cases} \tag{3.3}$$

  The purpose of the abstraction is to abstract away the details of concrete values such as `TX` = *Texas*:`state`.

- PREDREL contains features $\{\text{PREDREL}[\alpha(p), \mathbf{q}] : p \in \mathcal{P}, \mathbf{q} \in (\{\swarrow, \searrow\} \times \mathcal{R})^*\}$. A feature fires when a node x has predicate p and is connected via some path $\mathbf{q} = (d_1, r_1), \ldots, (d_m, r_m)$ to the lowest descendant node y with the property that each node between x and y has a null predicate. Each (d, r) on the path represents an edge labeled with relation r connecting to a left ($d = \swarrow$) or right ($d = \searrow$) child. If x has no children, then $m = 0$. The most common case is when $m = 1$, but $m = 2$ also occurs with the aggregate and execute relations (e.g., $\text{PREDREL}[\texttt{count}, \searrow {}^{1}_{1} \searrow \Sigma]$ fires for Figure 2.3(a)).
- PREDRELPRED contains features $\{\text{PREDRELPRED}[\alpha(p), \mathbf{q}, \alpha(p')] : p, p' \in \mathcal{P}, \mathbf{q} \in (\{\swarrow, \searrow\} \times \mathcal{R})^*\}$, which are the same as PREDREL, except that we include both the predicate p of x and the predicate p' of the descendant node y. These features do not fire if $m = 0$.
- TRIGGERPRED contains features $\{\text{TRIGGERPRED}[\mathbf{s}, p] : \mathbf{s} \in W^*, p \in \mathcal{P}\}$, where $W = \{it, \textit{Texas}, \ldots\}$ is the set of words. Each of these features fires when a span of the utterance with words $\mathbf{s}$ triggers the predicate p—more precisely, when a subtree $\langle p; \mathbf{e} \rangle_{i..j}$ exists with $\mathbf{s} = \mathbf{x}_{i+1..j}$. Note that these lexicalized features use the predicate p rather than the abstracted version $\alpha(p)$.
- TRACEPRED contains features $\{\text{TRACEPRED}[s, p, d] : s \in W^*, p \in \mathcal{P}, d \in \{\swarrow, \searrow\}\}$, each of which fires when a trace predicate p has been inserted over a word s. The situation is the following: Suppose we have a subtree a that ends at position k (there is a predicate in a that is triggered by a phrase with right endpoint k) and another subtree b that begins at k'. Recall that in the construction mechanism (2.46), we



Feature template	Feature j	Count $\phi(\mathbf{x}, z)_j$	Parameter θ_j
[Number of predicates]	PredHit	3	2.721
[Predicate]	Pred[state]	1	0.570
	Pred[border]	1	−2.596
	Pred[∗:state]	1	1.511
[Predicate + relation]	PredRel[state $\searrow_1^1$]	1	−0.262
	PredRel[border $\searrow_1^2$]	1	−2.248
	PredRel[∗:state]	1	1.059
[Predicate + relation + predicate]	PredRelPred[state $\searrow_1^1$ border]	1	2.119
	PredRelPred[border $\searrow_1^2$ ∗:state]	1	1.090
[Word + trigger predicate]	TriggerPred[*states*, state]	1	3.262
	TriggerPred[*Texas*, *Texas*:state]	1	−2.272
[Word + trace predicate]	TracePred[*that*, $\searrow$ border]	1	3.041
	TracePred[*border*, $\searrow$ border]	1	−0.253
[Word + trace relation]	TraceRel[*that*, $\searrow_1^1$]	1	0.000
	TraceRel[*border*, $\searrow_1^1$]	1	0.000
[Word + trace predicate + relation]	TracePredRel[*that*, state $\searrow_1^1$]	1	0.000
	TracePredRel[*border*, state $\searrow_1^1$]	1	0.000
		Score: $\phi(\mathbf{x}, z)^\top \theta = 13.184$	

Figure 3.1: For each utterance-DCS tree pair $(\mathbf{x}, z)$, we define a feature vector $\phi(\mathbf{x}, z)$, whose j-th component is the number of times a feature j occurs in $(\mathbf{x}, z)$. Each feature has an associated parameter θ_j, which is estimated from data in Section 3.2. The inner product of the feature vector and parameter vector yields a compatibility score.

can insert a trace predicate $p \in L(\epsilon)$ between the roots of a and b. Then, for every word $\mathbf{x}_j$ in between the spans of the two subtrees ($j = \{k+1, \ldots, k'\}$), the feature

TracePred[$\mathbf{x}_j, p, d$] fires ($d = \swarrow$ if b dominates a and $d = \searrow$ if a dominates b).

- TraceRel contains features $\{\text{TraceRel}[s, d, r] : s \in W^*, d \in \{\swarrow, \searrow\}, r \in \mathcal{R}\}$, each of which fires when some trace predicate with parent relation r has been inserted over a word s.
- TracePredRel contains features $\{\text{TracePredRel}[s, p, d, r] : s \in W^*, p \in \mathcal{P}, d \in \{\swarrow, \searrow\}, r \in \mathcal{R}\}$, each of which fires when a predicate p is connected via child relation r to some trace predicate over a word s.

These features are simple generic patterns which can be applied for modeling essentially any distribution over sequences and labeled trees—there is nothing specific to DCS at all. The first half of the feature templates (PredHit, Pred, PredRel, PredRelPred) capture properties of the tree independent of the utterance, and are similar to ones used for syntactic dependency parsing. The other feature templates (TriggerPred, TracePred, TraceRel, TracePredRel) connect predicates in the DCS tree with words in the utterance, similar to those in a model of machine translation.

3.2 Parameter Estimation

We have now fully specified the details of the graphical model in Figure 1.2: Section 3.1 described semantic parsing and Chapter 2 described semantic evaluation. Next, we focus on estimating the parameters θ of the model from data.

3.2.1 Objective Function

We assume that our learning algorithm is given a training dataset $\mathcal{D}$ containing question-answer pairs $(\mathbf{x}, y)$. Because the logical forms are unobserved, we work with $\log p(y \mid \mathbf{x}; C, \theta)$, the marginal log-likelihood of obtaining the correct answer y given an utterance $\mathbf{x}$. This marginal log-likelihood sums over all $z \in C(\mathbf{x})$ that evaluate to y:

$$\log p(y \mid \mathbf{x}; C, \theta) = \log p(z \in C^y(\mathbf{x}) \mid \mathbf{x}; C, \theta) \tag{3.4}$$

$$= A(\theta; \mathbf{x}, C^y) - A(\theta, \mathbf{x}, C), \text{ where} \tag{3.5}$$

$$C^y(\mathbf{x}) \stackrel{\text{def}}{=} \{z \in C(\mathbf{x}) : [\![z]\!]_w = y\}. \tag{3.6}$$

Here, $C^y(\mathbf{x})$ is the set of DCS trees z with denotation y.

We call an example $(\mathbf{x}, y) \in \mathcal{D}$ *feasible* if the candidate set of $\mathbf{x}$ contains a DCS tree that evaluates to y ($C^y(\mathbf{x}) \neq \emptyset$). Define an objective function $\mathcal{O}(\theta, C)$ containing two terms: The first term is the sum of the marginal log-likelihood over all feasible training examples. The

second term is a quadratic penalty on the parameters θ with regularization parameter λ. Formally:

$$\begin{aligned}\mathcal{O}(\theta, C) &\stackrel{\text{def}}{=} \sum_{\substack{(\mathbf{x},y)\in\mathcal{D}\\ C^y(\mathbf{x})\neq\emptyset}} \log p(y \mid \mathbf{x}; C, \theta) - \frac{\lambda}{2}\|\theta\|_2^2 \\ &= \sum_{\substack{(\mathbf{x},y)\in\mathcal{D}\\ C^y(\mathbf{x})\neq\emptyset}} \left(A(\theta; \mathbf{x}, C^y) - A(\theta; \mathbf{x}, C)\right) - \frac{\lambda}{2}\|\theta\|_2^2.\end{aligned} \tag{3.7}$$

We would like to maximize $\mathcal{O}(\theta, C)$. The log-partition function $A(\theta; \cdot, \cdot)$ is convex, but $\mathcal{O}(\theta, C)$ is the difference of two log-partition functions and hence is not concave (nor convex). Our plan is to use gradient-based optimization to hill climb the objective anyway. A standard result is that the derivative of the log-partition function is the expected feature vector (Wainwright and Jordan, 2008). Using this, we obtain the gradient of our objective function:

$$\frac{\partial\mathcal{O}(\theta, C)}{\partial\theta} = \sum_{\substack{(\mathbf{x},y)\in\mathcal{D}\\ C^y(\mathbf{x})\neq\emptyset}} \left(\mathbb{E}_{p(z|\mathbf{x};C^y,\theta)}[\phi(\mathbf{x}, z)] - \mathbb{E}_{p(z|\mathbf{x};C,\theta)}[\phi(\mathbf{x}, z)]\right) - \lambda\theta. \tag{3.8}$$

Updating the parameters in the direction of the gradient would move the parameters towards the DCS trees that yield the correct answer (C^y) and away from over all candidate DCS trees (C). We can use any standard numerical optimization algorithm that requires only black-box access to a gradient. Section 4.3.4 will discuss the empirical ramifications the choice of optimization algorithm.

3.2.2 Algorithm

Given a candidate set function $C(\mathbf{x})$, we can optimize (3.7) to obtain parameters θ. Ideally, we would use $C(\mathbf{x}) = \mathcal{Z}_L(\mathbf{x})$, the candidate sets from our construction mechanism in Section 2.6, but we quickly run into the problem of computing (3.8) efficiently. Note that $\mathcal{Z}_L(\mathbf{x})$ (defined in (2.44)) grows exponentially with the length of $\mathbf{x}$. This by itself is not a show stopper. Our features (Section 3.1.1) actually decompose along the edges of the DCS tree, so it is possible to use dynamic programming[1] to compute the second expectation $\mathbb{E}_{p(z|\mathbf{x};\mathcal{Z}_L,\theta)}[\phi(\mathbf{x}, z)]$ of (3.8). The problem is computing the first expectation $\mathbb{E}_{p(z|\mathbf{x};\mathcal{Z}_L^y,\theta)}[\phi(\mathbf{x}, z)]$, which sums over the subset of candidate DCS trees z satisfying the constraint $[\![z]\!]_w = y$. Though this is a smaller set, there is no efficient dynamic program for this set since the constraint does not decompose along the structure of the DCS tree. Therefore, we need to approximate $\mathcal{Z}_L^y$, and in fact, we will approximate $\mathcal{Z}_L$ as well so that the two expectations in (3.8) are coherent.

[1] The state of the dynamic program would be the span $i..j$ and the head predicate over that span.

Learning Algorithm

Initialize: $\theta^{(0)} \leftarrow (0, \ldots, 0)$
For each iteration $t = 1, \ldots, T$:
  Update candidate sets: $C^{(t)}(\mathbf{x}) \leftarrow \tilde{\mathcal{Z}}_{L,\theta^{(t-1)}}(\mathbf{x})$
  Update parameters: $\theta^{(t)} \leftarrow \text{argmax}_\theta \mathcal{O}(\theta, C^{(t)})$
Return $\theta^{(T)}$

Figure 3.2: The learning algorithm alternates between updating the candidate sets based on beam search and updating the parameters using standard numerical optimization.

Recall that $\mathcal{Z}_L(\mathbf{x})$ was built by recursively constructing a set of DCS trees $C_{i,j}(\mathbf{x})$ for each span $i..j$. In our approximation, we simply use beam search, which truncates each $C_{i,j}(\mathbf{x})$ to include the (at most) K DCS trees with the highest score $\phi(\mathbf{x}, z)^\top \theta$. We let $\tilde{C}_{i,j,\theta}(\mathbf{x})$ denote this approximation and define the set of candidate DCS trees with respect to the beam search:

$$\tilde{\mathcal{Z}}_{L,\theta}(\mathbf{x}) = \tilde{C}_{0,n,\theta}(\mathbf{x}). \tag{3.9}$$

We now have a chicken-and-egg problem: If we had good parameters θ, we could generate good candidate sets $C(\mathbf{x})$ using beam search $\tilde{\mathcal{Z}}_{L,\theta}(\mathbf{x})$. If we had good candidate sets $C(\mathbf{x})$, we could generate good parameters by optimizing our objective $\mathcal{O}(\theta, C)$ in (3.7). This problem leads to a natural solution: simply alternate between the two steps (Figure 3.2). This procedure is not theoretically guaranteed to converge due to the heuristic nature of the beam search, but we found it to be effective in practice.

Lastly, we use the trained model with parameters θ to answer new questions $\mathbf{x}$ by choosing the most likely answer y, summing out the latent logical form z:

$$F_\theta(\mathbf{x}) \stackrel{\text{def}}{=} \operatorname*{argmax}_y p(y \mid \mathbf{x}; \theta, \tilde{\mathcal{Z}}_{L,\theta}) \tag{3.10}$$

$$= \operatorname*{argmax}_y \sum_{\substack{z \in \tilde{\mathcal{Z}}_{L,\theta}(\mathbf{x}) \\ [\![z]\!]_w = y}} p(z \mid \mathbf{x}; \theta, \tilde{\mathcal{Z}}_{L,\theta}). \tag{3.11}$$

Chapter 4

Experiments

We have now completed the conceptual part of this thesis—using DCS trees to represent logical forms (Chapter 2), and learning a probabilistic model over them (Chapter 3). In this section, we evaluate and study our approach empirically. Our main result is that our system obtains higher accuracies than existing systems, despite requiring no annotated logical forms.

4.1 Experimental Setup

We first describe the datasets (Section 4.1.1) that we use to train and evaluate our system. We then mention various choices in the model and learning algorithm (Section 4.1.2). One of these choices is the lexical triggers, which is further discussed in Section 4.1.3.

4.1.1 Datasets

We tested our methods on two standard datasets, referred to in this thesis as GEO and JOBS. These datasets were created by Ray Mooney's group during the 1990s and have been used to evaluate semantic parsers for over a decade.

US Geography The GEO dataset, originally created by Zelle and Mooney (1996), contains 880 questions about US Geography and a database of facts encoded in Prolog. The questions in GEO ask about general properties (e.g., area, elevation, population) of geographical entities (e.g., cities, states, rivers, mountains). Across all the questions, there are 280 word types, and the length of an utterance ranges from 4 to 19 words, with an average of 8.5 words. The questions involve conjunctions, superlatives, negation, but no generalized quantification. Each question is annotated with a logical form in Prolog, for example:

Utterance: *What is the highest point in Florida?*
Logical form: `answer(A,highest(A,(place(A),loc(A,B),const(B,stateid(florida)))))`

Since our approach learns from answers, not logical forms, we evaluated the annotated logical forms on the provided database to obtain the correct answers.

Recall that a world/database w maps each predicate $p \in \mathcal{P}$ to a set of tuples $w(p)$. Some predicates contain the set of tuples explicitly (e.g., `mountain`); others can be derived (e.g., `higher` takes two entities x and y and returns true if $\texttt{elevation}(x) > \texttt{elevation}(y)$). Other predicates are higher-order (e.g., `sum`, `highest`) in that they take other predicates as arguments. We do not use the provided domain-specific higher-order predicates (e.g., `highest`), but rather provide domain-independent higher-order predicates (e.g., `argmax`) and the ordinary domain-specific predicates (e.g., `elevation`). This provides more compositionality and therefore better generalization. Similarly, we use `more` and `elevation` instead of `higher`. Altogether, $\mathcal{P}$ contains 43 predicates plus one predicate for each value (e.g., `CA`).

Job Queries The JOBS dataset (Tang and Mooney, 2001) contains 640 natural language queries about job postings. Most of the questions ask for jobs matching various criteria: job title, company, recruiter, location, salary, languages and platforms used, areas of expertise, required/desired degrees, and required/desired years of experience. Across all utterances, there are 388 word types, and the length of an utterance ranges from 2 to 23 words, with an average of 9.8 words. The utterances are mostly based on conjunctions of criteria, with a sprinkling of negation and disjunction. Here is an example:

Utterance: *Are there any jobs using Java that are not with IBM?*
Logical form: `answer(A,(job(A),language(A,'java'),¬company(A,'IBM')))`

The JOBS dataset comes with a database, which we can use as the world w. However, when the logical forms are evaluated on this database, close to half of the answers are empty (no jobs match the requested criteria). Therefore, there is a large discrepancy between obtaining the correct logical form (which has been the focus of most work on semantic parsing) and obtaining the correct answer (our focus).

To bring these two into better alignment, we generated a random database as follows: We created $m = 100$ jobs. For each job j, we go through each predicate p (e.g., `company`) that takes two arguments, a job and a target value. For each of the possible target values v, we add (j, v) to $w(p)$ independently with probability $\alpha = 0.8$. For example, for $p = \texttt{company}$, $j = \texttt{job37}$, we might add $(\texttt{job37}, \texttt{IBM})$ to $w(\texttt{company})$. The result is a database with a total of 23 predicates (which includes the domain-independent ones) in addition to the value predicates (e.g., `IBM`).

The goal of using randomness is to ensure that two different logical forms will most likely yield different answers. For example, consider two logical forms:

$$z_1 = \lambda j.\texttt{job}(j) \wedge \texttt{company}(j, \texttt{IBM}), \tag{4.1}$$
$$z_2 = \lambda j.\texttt{job}(j) \wedge \texttt{language}(j, \texttt{Java}). \tag{4.2}$$

Under the random construction, The denotation of z_1 is S_1, an random subset of the jobs, where each job is included in S_1 independently with probability α, and the denotation

of z_2 is S_2, which has the same distribution as S_1 but importantly is independent of S_1. Therefore, the probability that $S_1 = S_2$ is $[\alpha^2 + (1-\alpha)^2]^m$, which is exponentially small in m. This construction yields a world that is not entirely "realistic" (a job might have multiple employers), but it ensures that if we get the correct answer, we probably also obtain the correct logical form.

4.1.2 Settings

There are a number of settings which control the tradeoffs between computation, expressiveness, and generalization power of our model, shown below. For now, we will use generic settings chosen rather crudely; Section 4.3.4 will explore the effect of changing these settings.

Lexical Triggers The lexical triggers L (Section 2.6.1) define the set of candidate DCS trees for each utterance. There is a tradeoff between expressiveness and computational complexity: The more triggers we have, the more DCS trees we can consider for a given utterance, but then either the candidate sets become too large or beam search starts dropping the good DCS trees. Choosing lexical triggers is important and requires additional supervision (Section 4.1.3).

Features Our probabilistic semantic parsing model is defined in terms of feature templates (Section 3.1.1). Richer features increase expressiveness but also might lead to overfitting. By default, we include all the feature templates.

Number of training examples (n) An important property of any learning algorithm is its sample complexity—how many training examples are required to obtain a certain level of accuracy? By default, all training examples are used.

Number of training iterations (T) Our learning algorithm (Figure 3.2) alternates between updating candidate sets and update parameters for T iterations. We use $T = 5$ as the default value.

Beam size (K) The computation of the candidate sets in Figure 3.2 is based on beam search where each intermediate state keeps at most K DCS trees. The default value is $K = 100$.

Optimization algorithm Updating the parameters in Figure 3.2 involves optimizing an the objective function $\mathcal{O}(\theta, C)$. The optimization algorithm determines both the speed of convergence and the quality of the converged solution. By default, we use the standard L-BFGS algorithm (Nocedal, 1980) with a backtracking line search for choosing the step size.

Regularization (λ) The regularization parameter $\lambda > 0$ in the objective function $\mathcal{O}(\theta, C)$ is another knob for controlling the tradeoff between fitting and overfitting. The default is $\lambda = 0.01$.

4.1.3 Lexical Triggers

The lexical triggers L (Section 2.6.1) is a set of entries $(\mathbf{s}, p)$, where $\mathbf{s}$ is a sequence of words and p is a predicate. We will run experiments on two sets of lexical triggers: *base triggers* L_{B} and *augmented triggers* $L_{\text{B+P}}$.

Base Triggers The base triggers L_{B} include three types of entries:

- Domain-independent triggers: For each domain-independent predicate (e.g., `argmax`), we manually specify a few words associated with that predicate (e.g., *most*). The full list is shown at the top of Figure 4.1.
- Values: For each value x that appears in the world (specifically, $x \in v_j \in w(p)$ for some tuple v, index j, and predicate p), L_{B} contains an entry (x, x) (e.g., $(\mathit{Boston}, \mathit{Boston}$: `city`)). Note that this rule implicitly specifies an infinite number of triggers.

  Regarding predicate names, we do *not* add entries such as (*city*, `city`), because we want our system to be language-independent. In Turkish, for instance, we would not have the luxury of lexicographical cues that associate `city` with *şehir*. So we should think of the predicates as just symbols `predicate1`, `predicate2`, etc. On the other hand, values in the database are generally proper nouns (e.g., city names) for which there are generally strong cross-linguistic lexicographic similarities.
- Part-of-speech (POS) triggers:[1] For each domain-specific predicate p, we specify a set of part-of-speech tags T. Implicitly, L_{B} contains all pairs (x, p) where the word x has a POS tag $t \in T$. For example, for `city`, we would specify NN and NNS, which means that any word which is a singular or plural common noun triggers the predicate `city`. Note that *city* triggers `city` as desired, but *state* also triggers `city`.

  The POS triggers for GEO and JOBS domains are shown in the left side of Figure 4.1. Note that that some predicates such as `traverse` and `loc` are not associated with any POS tags. Predicates corresponding to verbs and prepositions are not included as overt lexical triggers, but rather included as trace predicates $L(\epsilon)$. In constructing the logical forms, nouns and adjectives serve as anchor points. Trace predicates can be inserted in between these anchors. This strategy is more flexible than requiring each predicate to spring from some word.

Augmented Triggers We now define the augmented triggers $L_{\text{B+P}}$, which contains more domain-specific information than L_{B}. Specifically, for each domain-specific predicate (e.g., `city`), we manually specify a single prototype word (e.g., *city*) associated with that predicate.

[1] To perform POS tagging, we used the Berkeley Parser (Petrov et al., 2006), trained on the WSJ Treebank (Marcus et al., 1993) and the Question Treebank (Judge et al., 2006)—thanks to Slav Petrov for providing the trained grammar.

Domain-independent triggers

no | *not* | *dont* | *doesnt* | *outside* | *exclude*	⊢	`not`
each | *every*	⊢	`every`
most	⊢	`argmax`
least | *fewest*	⊢	`argmin`
count | *number* | *many*	⊢	`count`
large | *high* | *great*	⊢	`affirm`
small | *low*	⊢	`negate`
sum | *combined* | *total*	⊢	`sum`
less | *at most*	⊢	`less`
more | *at least*	⊢	`more`
called | *named*	⊢	`nameObj`

GEO **POS triggers**

NN | NNS	⊢	`city` | `state` | `country` | `lake` | `mountain` | `river` | `place` | `person` | `capital` | `population`
NN | NNS | JJ	⊢	`len` | `negLen` | `size` | `negSize` | `elevation` | `negElevation` | `density` | `negDensity` | `area` | `negArea`
NN | NNS	⊢	`usa:country`
JJ	⊢	`major`
WRB	⊢	`loc`
ϵ	⊢	`loc` | `next_to` | `traverse` | `hasInhabitant`

GEO **prototypes**

city	⊢ `city`	*person*	⊢ `person`
state	⊢ `state`	*population*	⊢ `population`
country	⊢ `country`	*long*	⊢ `len`
lake	⊢ `lake`	*short*	⊢ `negLen`
mountain	⊢ `mountain`	*large*	⊢ `size`
river	⊢ `river`	*small*	⊢ `negSize`
point	⊢ `place`	*high*	⊢ `elevation`
where	⊢ `loc`	*low*	⊢ `negElevation`
major	⊢ `major`	*dense*	⊢ `density`
capital	⊢ `capital`	*sparse*	⊢ `negDensity`
high point	⊢ `high_point`	*area*	⊢ `area`

JOBS **POS triggers**

NN | NNS	⊢	`job` | `deg` | `exp` | `language` | `loc`
ϵ	⊢	`salary_greater_than` | `require` | `desire` | `title` | `company` | `recruiter` | `area` | `platform` | `application` | `language` | `loc`

JOBS **prototypes**

(beginning of utterance)	⊢	`job`
degree	⊢	`deg`
experience	⊢	`exp`
language	⊢	`language`
location	⊢	`loc`

Figure 4.1: Lexical triggers used in our experiments.

Under $L_{\text{B+P}}$, *city* would trigger only `city` because *city* is a prototype word, but *town* would trigger all the NN predicates (`city`, `state`, `country`, etc.) because it is not a prototype word.

Prototype triggers require only a modest amount of domain-specific supervision (see the right side of Figure 4.1 for the entire list for GEO and JOBS). In fact, as we'll see in Section 4.2, prototype triggers are not absolutely required to obtain good accuracies, but they give an extra boost and also improve computational efficiency by reducing the set of candidate DCS trees.

Finally, we use a small set of rules that expand morphology (e.g., *largest* is mapped to

most large). To determine triggering, we stem all words using the Porter stemmer Porter (1980), so that *mountains* triggers the same predicates as *mountain*.

4.2 Comparison with Other Systems

We now compare our approach with existing methods (Section 4.2). We used the same training-test splits as Zettlemoyer and Collins (2005) (600 training and 280 test examples for GEO, 500 training and 140 test examples for JOBS). For development, we created five random splits of the training data. For each split, we put 70% of the examples into a *development training set* and the remaining 30% into a *development test set*. The actual test set was only used for obtaining final numbers.

4.2.1 Systems that Learn from Question-Answer Pairs

We first compare our system (henceforth, LJK11) with Clarke et al. (2010) (henceforth, CGCR10), which is most similar to in our work in that it also learns from question-answer pairs without using annotated logical forms. CGCR10 works with the FunQL language and casts semantic parsing as integer linear programming (ILP). In each iteration, the learning algorithm solves the ILP to predict the logical form for each training example. The examples with correct predictions are fed to a structural SVM and the model parameters are updated.

Though similar in spirit, there are some important differences between CGCR10 and our approach. They use ILP instead of beam search and structural SVM instead of log-linear models, but the main difference is which examples are used for learning. Our approach learns on any feasible example (Section 3.2.1), one where the candidate set contains a logical form that evaluates to the correct answer. CGCR10 uses a much more stringent criteria: the highest scoring logical form must evaluate to the correct answer. Therefore, for their algorithm to progress, the model already must be non-trivially good before learning even starts. This is reflected in the amount of prior knowledge and initialization that CGCR10 employs before learning starts: WordNet features, and syntactic parse trees, and a set of lexical triggers with 1.42 words per non-value predicate. Our system with base triggers requires only simple indicator features, POS tags, and 0.5 words per non-value predicate.

CGCR10 created a version of GEO which contains 250 training and 250 test examples. Table 4.1 compares the empirical results on this split. We see that our system (LJK11) with base triggers significantly outperforms CGCR10 (84% over 73.2%), and it even outperforms the version of CGCR10 that is trained using logical forms (84.0% over 80.4%). If we use augmented triggers, we widen the gap by another 3.6%.[2]

[2]Note that the numbers for LJK11 differ from those presented in Liang et al. (2011), which reports results based on 10 different splits rather than the one used by CGCR10.

System		Accuracy
CGCR10 w/answers	(Clarke et al., 2010)	73.2
CGCR10 w/logical forms	(Clarke et al., 2010)	80.4
LJK11 w/base triggers	(Liang et al., 2011)	84.0
LJK11 w/augmented triggers	(Liang et al., 2011)	**87.6**

Table 4.1: Results on GEO with 250 training and 250 test examples. Our system (LJK11 with base triggers and no logical forms) obtains higher test accuracy than CGCR10, even when CGCR10 is trained using logical forms.

4.2.2 State-of-the-Art Systems

We now compare our system (LJK11) with state-of-the-art systems, which all require annotated logical forms (except PRECISE). Here is a brief overview of the systems:

- COCKTAIL (Tang and Mooney, 2001) uses inductive logic programming to learn rules for driving the decisions of a shift-reduce semantic parser. It assumes that a lexicon (mapping from words to predicates) is provided.
- PRECISE (Popescu et al., 2003) does not use learning, but instead relies on matching words to strings in the database using various heuristics based on WordNet and the Charniak parser. Like our work, it also uses database type constraints to rule out spurious logical forms. One of the unique features of PRECISE is that it has 100% precision—it refuses to parse an utterance which it deems *semantically intractable*.
- SCISSOR (Ge and Mooney, 2005) learns a generative probabilistic model that extends the Collins models (Collins, 1999) with semantic labels, so that syntactic and semantic parsing can be done jointly.
- SILT (Kate et al., 2005) learns a set of transformation rules for mapping utterances to logical forms.
- KRISP (Kate and Mooney, 2006) uses SVMs with string kernels to drive the local decisions of a chart-based semantic parser.
- WASP (Wong and Mooney, 2006) uses log-linear synchronous grammars to transform utterances into logical forms, starting with word alignments obtained from the IBM models.
- λ-WASP (Wong and Mooney, 2007) extends WASP to work with logical forms that contain bound variables (lambda abstraction).

- LNLZ08 (Lu et al., 2008) learns a generative model over *hybrid trees*, which are logical forms augmented with natural language words. IBM model 1 is used to initialize the parameters, and a discriminative reranking step works on top of the generative model.
- ZC05 (Zettlemoyer and Collins, 2005) learns a discriminative log-linear model over CCG derivations. Starting with a manually-constructed domain-independent lexicon, the training procedure grows the lexicon by adding lexical entries derived from associating parts of an utterance with parts of the annotated logical form.
- ZC07 (Zettlemoyer and Collins, 2007) extends ZC05 with extra (disharmonic) combinators to increase the expressive power of the model.
- KZGS10 (Kwiatkowski et al., 2010) uses a restricted higher-order unification procedure, which iteratively breaks up a logical form into smaller pieces. This approach gradually adds lexical entries of increasing generality, thus obviating the need for the manually-specified templates used by ZC05 and ZC07 for growing the lexicon. IBM model 1 is used to initialize the parameters.
- KZGS11 (Kwiatkowski et al., 2011) extends KZGS10 by factoring lexical entries into a template plus a sequence of predicates which fill the slots of the template. This factorization improves generalization.

With the exception of PRECISE, all other systems require annotated logical forms, whereas our system learns from annotated answers. On the other hand, many of the later systems require essentially no manually-crafted lexicon and instead rely on unsupervised word alignment (e.g., Wong and Mooney (2006, 2007); Kwiatkowski et al. (2010, 2011)) and/or lexicon learning (e.g., Zettlemoyer and Collins (2005, 2007); Kwiatkowski et al. (2010, 2011)). We cannot use these automatic techniques because they require annotated logical forms. Our system instead relies on lexical triggers, which does require some manual effort. These lexical triggers play a crucial role in the initial stages of learning, because they constrain the set of candidate DCS trees, barring a hopelessly intractable search problem.

Table 4.2 shows the results for GEO. Semantic parsers are typically evaluated on the accuracy of the logical forms: precision (the accuracy on utterances which are successfully parsed) and recall (the accuracy on all utterances). We only focus on recall (a lower bound on precision) and simply use the word *accuracy* to refer to recall.[3] Our system is evaluated only on answer accuracy because our model marginalizes out the latent logical form. All other systems are evaluated on the accuracy of logical forms. To calibrate, we also evaluated KZGS10 on answer accuracy and found that it was quite similar to its logical form accuracy (88.9% versus 88.2%).[4] This does not imply that our system would necessarily have a high

[3] Our system produces a logical form for every utterance, and thus our precision is the same as our recall.

[4] The 88.2% corresponds to 87.9% in Kwiatkowski et al. (2010). The difference is due to using a slightly newer version of the code.

System		LF	Answer
COCKTAIL	(Tang and Mooney, 2001)	79.4	–
PRECISE	(Popescu et al., 2003)	77.5	77.5
SCISSOR	(Ge and Mooney, 2005)	72.3	–
SILT	(Kate et al., 2005)	54.1	–
KRISP	(Kate and Mooney, 2006)	71.7	–
WASP	(Wong and Mooney, 2006)	74.8	–
λ-WASP	(Wong and Mooney, 2007)	86.6	–
LNLZ08	(Lu et al., 2008)	81.8	–
ZC05	(Zettlemoyer and Collins, 2005)	79.3	–
ZC07	(Zettlemoyer and Collins, 2007)	86.1	–
KZGS10	(Kwiatkowski et al., 2010)	88.2	88.9
KZGS11	(Kwiatkowski et al., 2010)	**88.6**	–
LJK11 w/base triggers	(Liang et al., 2011)	–	87.9
LJK11 w/augmented triggers	(Liang et al., 2011)	–	**91.4**

Table 4.2: Results on GEO. Logical form accuracy (LF) and answer accuracy (Answer) of the various systems. The first group of systems are evaluated using 10-fold cross-validation on all 880 examples; the second are evaluated on the 680 + 200 split of Zettlemoyer and Collins (2005). Our system (LJK11) with base triggers obtains comparable accuracy to past work, while with augmented triggers, our system obtains the highest overall accuracy.

logical form accuracy because multiple logical forms can produce the same answer, and our system does not receive a training signal to tease them apart. Even with only base triggers, our system (LJK11) outperforms all but two of the systems, falling short of KZGS10 by only one point (87.9% versus 88.9%).[5] With augmented triggers, our system takes the lead (91.4% over 88.9%).

Table 4.3 shows the results for JOBS. The two learning-based systems (COCKTAIL and ZC05) are actually outperformed by PRECISE, which is able to use strong database type constraints. By exploiting this information and doing learning, we obtain the best results.

4.3 Empirical Properties

In this section, we try to gain more intuition about the properties of our approach. All experiments in this section are performed on random development splits. Throughout this section, "accuracy" means development test accuracy.

[5]The 87.9% and 91.4% correspond to 88.6% and 91.1% in Liang et al. (2011). These differences are due to minor differences in the code.

System		LF	Answer
COCKTAIL	(Tang and Mooney, 2001)	79.4	–
PRECISE	(Popescu et al., 2003)	**88.0**	88.0
ZC05	(Zettlemoyer and Collins, 2005)	79.3	–
LJK11 w/base triggers	(Liang et al., 2011)	–	90.7
LJK11 w/augmented triggers	(Liang et al., 2011)	–	**95.0**

Table 4.3: Results on JOBS. Both PRECISE and our system use database type constraints, which results in a decisive advantage over the other systems. In addition, LJK11 incorporates learning and therefore obtains the highest accuracies.

4.3.1 Error Analysis

To understand the type of errors our system makes, we examined one of the development runs, which had 34 errors on the test set. We classified these errors into the following categories (the number of errors in each category is shown in parentheses):

- Incorrect POS tags (8): GEO is out-of-domain for our POS tagger, so the tagger makes some basic errors which adversely affect the predicates that can be lexically triggered. For example, the question *What states border states ...* is tagged as WP VBZ NN NNS ..., which means that the first *states* cannot trigger `state`. In another example, *major river* is tagged as NNP NNP, so these cannot trigger the appropriate predicates either, and thus the desired DCS tree cannot even be constructed.

- Non-projectivity (3): The candidate DCS trees are defined by a projective construction mechanism (Section 2.6) that prohibits edges in the DCS tree from crossing. This means we cannot handle utterances such as *largest city by area*, since the desired DCS tree would have `city` dominating `area` dominating `argmax`. To construct this DCS tree, we could allow local reordering of the words.

- Unseen words (2): We never saw *at least* or *sea level* at training time. The former has the correct lexical trigger, but not a sufficiently large feature weight (0) to encourage its use. For the latter, the problem is more structural: We have no lexical triggers for 0:`length`, and only adding more lexical triggers can solve this problem.

- Wrong lexical triggers (7): Sometimes the error is localized to a single lexical trigger. For example, the model incorrectly thinks *Mississippi* is the state rather than the river, and that *Rochester* is the city in New York rather than the name, even though there are contextual cues to disambiguate in these cases.

- Extra words (5): Sometimes, words trigger predicates that should be ignored. For example, for *population density*, the first word triggers `population`, which is used

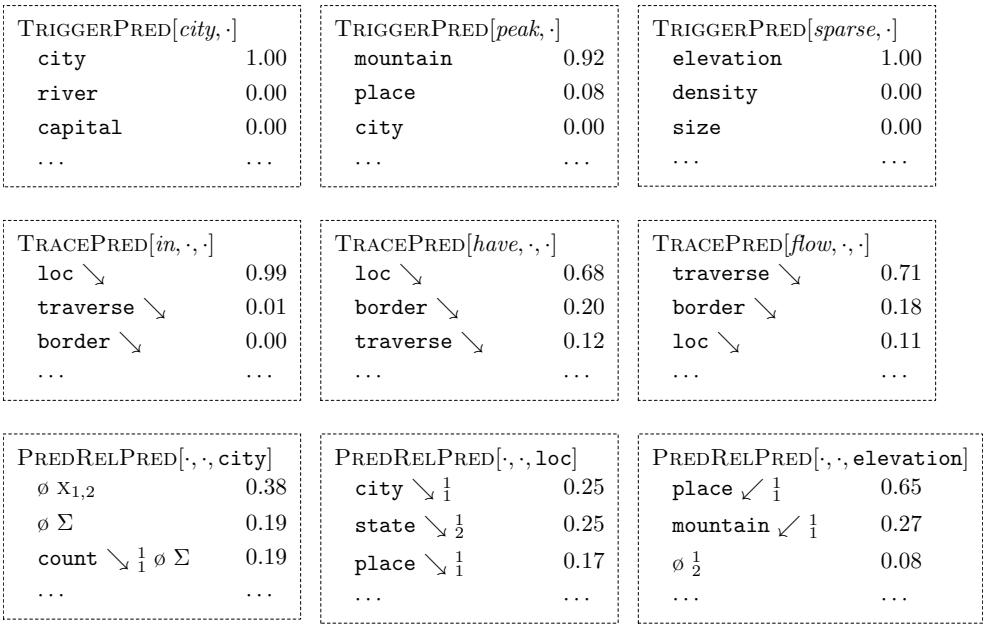


Figure 4.2: Learned feature distributions. In a feature group (e.g., TriggerPred[*city*, ·]), each feature is associated with the marginal probability that the feature fires according to (4.3). Note that we have successfully learned that *city* means `city`, but incorrectly learned that *sparse* means `elevation` (due to the confounding fact that Alaska is the most sparse state and has the highest elevation).

rather than `density`.

- Over-smoothing of DCS tree (9): The first half of our features (Figure 3.1) are defined on the DCS tree alone; these produce a form of smoothing that encourages DCS trees to look alike regardless of the words. We found several instances where this essential tool for generalization went too far. For example, in *state of Nevada*, the trace predicate `border` is inserted between the two nouns, because it creates a structure more similar to that of the common question *what states border Nevada?*

4.3.2 Visualization of Features

Having analyzed the behavior of our system for individual utterances, let us move from the token-level to the type-level and analyze the learned parameters of our model. We do not look at raw feature weights, because there are complex interactions between them not represented by examining individual weights. Instead, we look at expected feature counts, which we think are more interpretable.

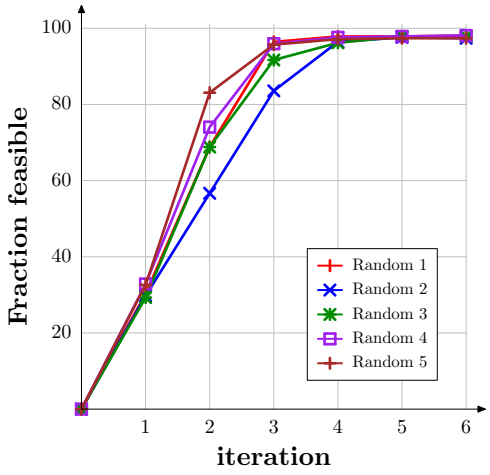


Figure 4.3: The fraction of feasible training examples increases steadily as the parameters, and thus, the beam search, improves. Each curve corresponds to a run on a different development split.

Consider a group of "competing" features J, for example $J = \{\textsc{TriggerPred}[\mathit{city}, p] : p \in \mathcal{P}\}$. We define a distribution $q(\cdot)$ over J as follows:

$$q(j) = \frac{N_j}{\sum_{j' \in J} N_{j'}}, \text{ where} \tag{4.3}$$
$$N_j = \sum_{(\mathbf{x},y) \in \mathcal{D}} \mathbb{E}_{p(z|\mathbf{x},\tilde{\mathcal{Z}}_{L,\theta},\theta)}[\phi(\mathbf{x}, z)].$$

Think of $q(j)$ as a marginal distribution (since all our features are positive) which represents the relative frequencies with which the features $j \in J$ fire with respect to our training dataset $\mathcal{D}$ and trained model $p(z \mid \mathbf{x}, \tilde{\mathcal{Z}}_{L,\theta}, \theta)$. To appreciate the difference between what this distribution and raw feature weights each capture, suppose we had two features, j_1 and j_2, which are identical ($\phi(\mathbf{x}, z)_{j_1} \equiv \phi(\mathbf{x}, z)_{j_2}$). The weights would be split across the two features, but the features would have the same marginal distribution ($q(j_1) = q(j_2)$). Figure 4.2 shows some of the feature distributions learned.

4.3.3 Learning, Search, Bootstrapping

Recall from Section 3.2.1 that a training example is feasible (with respect to our beam search) if the resulting candidate set contains a DCS tree with the correct answer. Infeasible examples are skipped, but an example may become feasible in a later iteration. A natural question is how many training examples are feasible in each iteration. Figure 4.3 shows the answer: Initially, only around 30% of the training examples are feasible; this is not surprising

given that all the parameters are zero, so our beam search is essentially unguided. However, training on just these examples improves the parameters, and over the next few iterations, the number of feasible examples steadily increases to around 97%.

In our algorithm, learning and search are deeply intertwined. Search is of course needed to learn, but learning also improves search. The general approach is similar in spirit to Searn (Daume et al., 2009), although we do not have any formal guarantees at this point.

Our algorithm also has a bootstrapping flavor. The "easy" examples are processed first, where easy is defined by the ability of beam search to generate the correct answer. This bootstrapping occurs quite naturally: Unlike most bootstrapping algorithms, we do not have to set a confidence threshold for accepting new training examples, something that can be quite tricky to do. Instead, our threshold falls out of the discrete nature of the beam search.

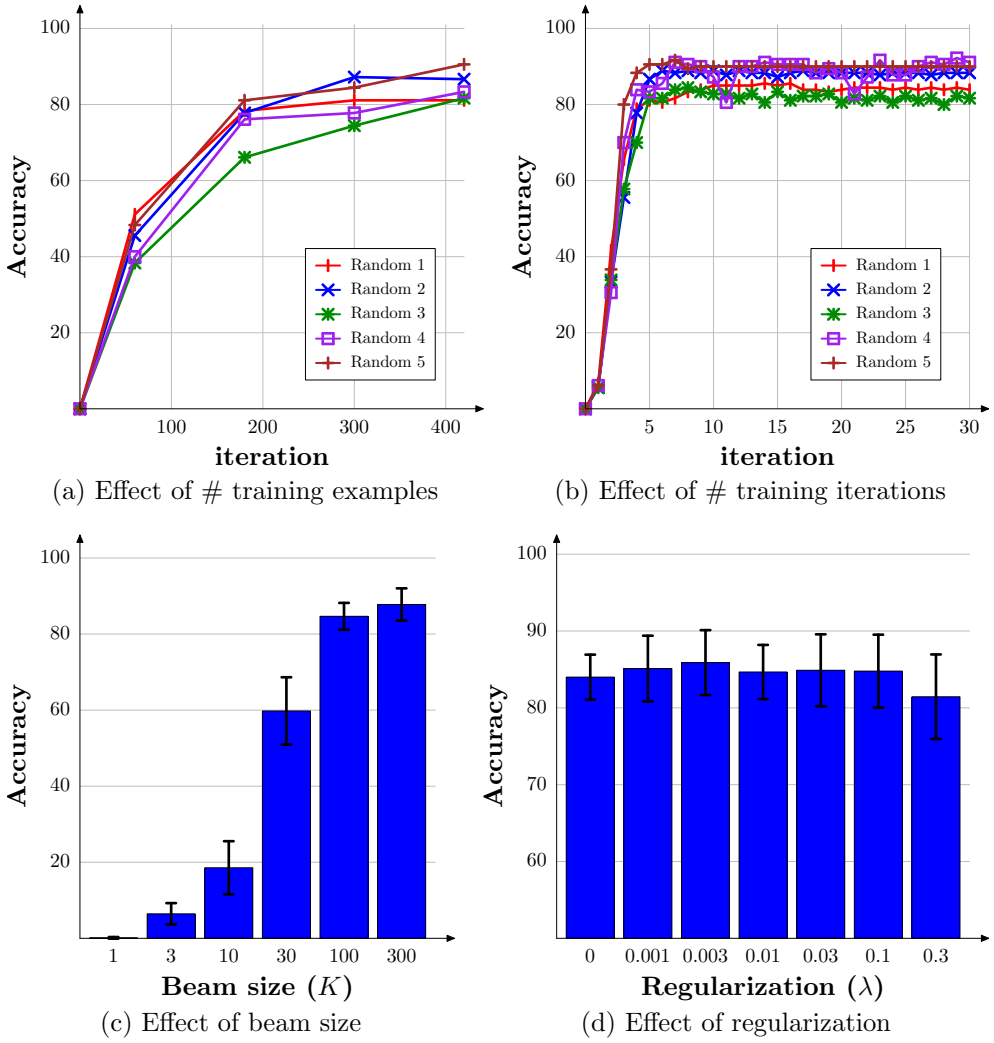
4.3.4 Effect of Various Settings

So far, we have used our approach with default settings (Section 4.1.2). How sensitive is the approach to these choices? Table 4.4 shows the impact of the feature templates. Figure 4.4 shows the effect of the number of training examples, number of training iterations, beam size, and regularization parameter. The overall conclusion is that there are no big surprises: Our default settings could be improved on slightly, but these differences are often smaller than the variation across different development splits.

Features	Accuracy
Pred	13.4 ± 1.6
Pred + PredRel	18.4 ± 3.5
Pred + PredRel + PredRelPred	23.1 ± 5.0
Pred + TriggerPred	61.3 ± 1.1
Pred + TriggerPred + Trace∗	76.4 ± 2.3
Pred + PredRel + PredRelPred + TriggerPred + Trace∗	84.7 ± 3.5

Table 4.4: There are two classes of feature templates: lexical features (TriggerPred,Trace*) and non-lexical features (PredRel,PredRelPred). The lexical features are relatively much more important for obtaining good accuracy (76.4% versus 23.1%), but adding the non-lexical features makes a significant contribution as well (84.7% versus 76.4%).

We now consider the choice of optimization algorithm to update the parameters given candidate sets (see Figure 3.2). Thus far, we have been using L-BFGS (Nocedal, 1980), which is a batch algorithm: Each iteration, we construct the candidate sets $C^{(t)}(\mathbf{x})$ for all the training examples before solving the optimization problem $\text{argmax}_\theta \mathcal{O}(\theta, C^{(t)})$. We now consider an online algorithm, stochastic gradient descent (SGD) (Robbins and Monro,



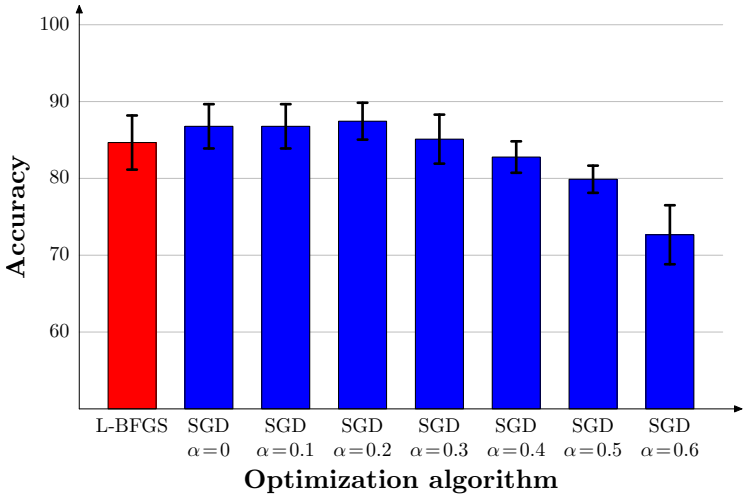
(a) Effect of # training examples

(b) Effect of # training iterations

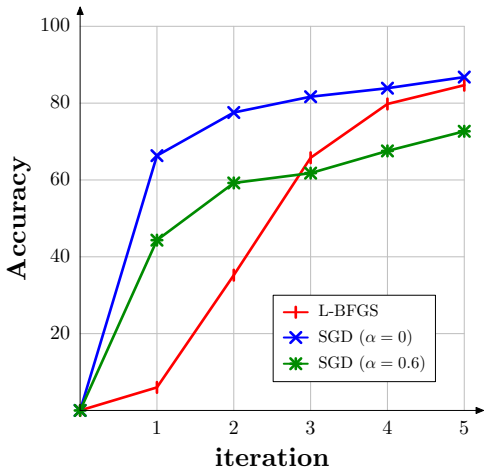
(c) Effect of beam size

(d) Effect of regularization

Figure 4.4: (a) The learning curve shows test accuracy as the number of training examples increases; about 300 examples suffices to get around 80% accuracy. (b) Although our algorithm is not guaranteed to converge, the test accuracy is fairly stable (with one exception) with more training iterations—hardly any overfitting occurs. (c) As the beam size increases, the accuracy increases monotonically, although the computational burden also increases. There is a small gain from our default setting of $K = 100$ to the more expensive $K = 300$. (d) The accuracy is relatively insensitive to the choice of the regularization parameter for a wide range of values. In fact, no regularization is also acceptable. This is probably because the features are simple, and the lexical triggers and beam search already provide some helpful biases.



(a) Effect of optimization algorithm



(b) Batch versus online

Figure 4.5: (a) Given the same number of iterations, compared to default batch algorithm (L-BFGS), the online algorithm (stochastic gradient descent) is slightly better for aggressive step sizes (small α) and worse for conservative step sizes (large α). (b) The online algorithm (with an appropriate choice of α) obtains a reasonable accuracy must faster than L-BFGS.

1951), which updates the parameters after computing the candidate set for each example. In particular, we iteratively scan through the training examples in a random order. For each example $(\mathbf{x}, y)$, we compute the candidate set using beam search. We then update the parameters in the direction of the gradient of the marginal log-likelihood for that example (see (3.8)) with step size $t^{-\alpha}$:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} + t^{-\alpha} \left(\frac{\partial \log p(y \mid \mathbf{x}; \tilde{\mathcal{Z}}_{L,\theta^{(t)}}, \theta)}{\partial \theta} \Big|_{\theta=\theta^{(t)}} \right). \tag{4.4}$$

The trickiest part about using SGD is selecting the correct step size: a small α leads to quick progress but also instability; a large α leads to the opposite. We let L-BFGS and SGD both take the same number of iterations (passes over the training set). Figure 4.5 shows that a very small α (less than 0.2) is best for our task, even though only values between 0.5 and 1 guarantee theoretical convergence. Our setting is slightly different since we are interleaving the SGD updates with beam search, which might also lead to unpredictable consequences. Furthermore, the non-convexity of the objective function exacerbates the unpredictability (Liang and Klein, 2009). Nonetheless, with a proper α, SGD converges much faster than L-BFGS and even to a slightly better solution.

Chapter 5

Discussion

The work we have presented in this thesis contains three important themes. The first theme is *semantic representation* (Section 5.1): How do we parametrize the mapping from utterances to their meanings? The second theme is *program induction* (Section 5.2): How do we efficiently search through the space of logical structures given a weak feedback signal? Finally, the last theme is *grounded language* (Section 5.3): How do we use constraints from the world to guide learning of language and conversely use language to interact with the world?

5.1 Semantic Representation

Since the late nineteenth century, philosophers and linguists have worked on elucidating the relationship between an utterance and its meaning. One of the pillars of formal semantics is Frege's principle of compositionality, that the meaning of an utterance is built by composing the meaning of its parts. What these parts are and how they are composed is the main question. The dominant paradigm, which stems from the seminal work of Richard Montague in the early 1970s (Montague, 1973), states that parts are lambda calculus expressions that correspond to syntactic constituents, and composition is function application.

Consider the compositionality principle from a statistical point of view, where we liberally construe compositionality as factorization. Factorization, the way a statistical model breaks into features, is necessary for generalization: It enables us to learn from previously seen examples and interpret new utterances. Projecting back to Frege's original principle, the parts are the features (Section 3.1.1), and composition is the DCS construction mechanism (Section 2.6) driven by parameters learned from training examples.

Taking the statistical view of compositionality, finding a good semantic representation becomes designing a good statistical model. But statistical modeling must also deal with the additional issue of language acquisition or learning, which presents complications: In absorbing training examples, our learning algorithm must inevitably traverse through intermediate

models that are wrong or incomplete. The algorithms must therefore tolerate this degradation, and do so in a computationally efficient way. For example, in the line of work on learning probabilistic CCGs (Zettlemoyer and Collins, 2005, 2007; Kwiatkowski et al., 2010), many candidate lexical entries must be entertained for each word even when polysemy does not actually exist (Section 2.6.4).

To improve generalization, the lexicon can be further factorized (Kwiatkowski et al., 2011), but this is all done within the constraints of CCG. DCS represents a departure from this tradition, which replaces a heavily-lexicalized constituency-based formalism with a lightly-lexicalized dependency-based formalism. We can think of DCS as a shift in linguistic coordinate systems, which makes certain factorizations or features more accessible. For example, we can define features on paths between predicates in a DCS tree which capture certain lexical patterns much more easily than on a lambda calculus expression or a CCG derivation.

The benefits of working with dependency-based logical forms also led to the independent development of a semantic representation called *natural logic form* (Alshawi et al., 2011), but the details of the two semantic formalisms and the goals are both different. Alshawi et al. (2011) focuses on parsing much complex sentences in an open-domain where a structured database or world does not exist. While they do equip their logical forms with a full model-theoretic semantics, the logical forms are actually closer to dependency trees: quantifier scope is left unspecified, and the predicates are simply the words.

Perhaps not immediately apparent is the fact that DCS draws an important idea from Discourse Representation Theory (DRT) (Kamp and Reyle, 1993)—not from its treatment of anaphora and presupposition which it is known for, but something closer to its core. This is the idea of having a logical form where all variables are existentially quantified and constraints are combined via conjunction—a Discourse Representation Structure (DRS) in DRT, or a basic DCS tree with only join relations. Computationally, such these logical structures conveniently encode CSPs. Linguistically, it appears that existential quantifiers play an important role and should be treated specially (Kamp and Reyle, 1993). DCS takes this core and focuses more on semantic compositionality and computation, while DRT focuses more on discourse and pragmatics.

In addition to the statistical view of DCS as a semantic representation, it is useful to think about DCS from the perspective of programming language design. Two programming languages can be equally expressive, but what matters is how simple it is to express a desired type of computation with in a given language. In some sense, we designed the DCS formal language to make it easy to represent computations expressed by natural language. An important part of DCS is the mark-execute construct, a uniform framework for dealing with the divergence between syntactic and semantic scope. This construct allows us to build simple DCS tree structures and still handle the complexities of phenomena such as quantifier scope variation. Compared to lambda calculus, think of DCS as a higher-level programming language tailored to natural language, which results in simpler programs (DCS trees). Simpler programs are easier for us to work with and easier for an algorithm to learn.

5.2 Program Induction

Searching over the space of programs is hard though. This is the central computational challenge of program induction, that of inferring programs (logical forms) from their behavior (denotations). This problem has been tackled by difference communities in various forms: program induction in AI, programming by demonstration in HCI, and program synthesis in programming languages. The core computational difficulty is that the supervision signal—the behavior—is a complex function of the program which cannot be easily inverted. What program generated the output *Arizona*, *Nevada*, and *Oregon*?

Perhaps somewhat counterintuitively, program induction is easier if we infer programs for not a single task but for multiple tasks. The intuition is that when the tasks are related, the solution to one task can help another task, both computationally in navigating the program space but statistically in choosing the appropriate program if there are multiple feasible possibilities (Liang et al., 2010). In our semantic parsing work, we want to infer a logical form for each utterance (task). Clearly the tasks are related because they use the same vocabulary to talk about the same domain.

Natural language also makes program induction easier by providing side information (words) which can be used to guide the search. There has been several papers that induce programs in this setting: Eisenstein et al. (2009) induces conjunctive formulae from natural language instructions, Piantadosi et al. (2008) induces first-order logic formulae using CCG in a small domain assuming observed lexical semantics, and Clarke et al. (2010) induces logical forms in semantic parsing. In the ideal case, the words would determine the program predicates, and the utterance would determine the entire program compositionally. But of course, this mapping is not given and must be learned.

5.3 Grounded Language

In recent years, there has been an increased interest in connecting language with the world.[1] One of the primary issues in grounded language is alignment—figuring out what fragments of utterances refer to what aspects of the world. In fact, semantic parsers trained on examples of utterances and annotated logical form (those discussed in Section 4.2.2) need to solve the task of aligning words to predicates. Some can learn from utterances paired with a set of logical forms, one of which is correct (Kate and Mooney, 2007; Chen and Mooney, 2008). Liang et al. (2009) tackles the even more difficult alignment problem of segmenting and aligning a discourse to a database of facts, where many parts of either side are irrelevant (Liang et al., 2009).

If we know how the world relates to language, we can leverage structure in the world to guide the learning and interpretation of language. We saw that type constraints from

[1]Here, *world* need not refer to the physical world, but could be any virtual world. The point is that the world has non-trivial structure and exists extra-linguistically.

the database/world reduces the set of candidate logical forms and leads to more accurate systems (Popescu et al., 2003; Liang et al., 2011). Even for syntactic parsing, information from the denotation of an utterance can be helpful (Schuler, 2003).

One of the exciting aspects about using the world for learning language is that it opens the door to many new types of supervision. We can obtain answers given a world, which are cheaper to obtain than logical forms (Clarke et al., 2010; Liang et al., 2011). Goldwasser et al. (2011) learns a semantic parser based on bootstrapping and estimating the confidence of its own predictions. Artzi and Zettlemoyer (2011) learns a semantic parser not from annotated logical forms, but from user interactions with a dialog system. Branavan et al. (2009, 2010, 2011) use reinforcement learning learn to follow natural language instructions from a reward signal. In general, supervision from the world is more indirectly related to the learning task, but it is often much more plentiful and natural to obtain.

The benefits can also flow from language to the world. For example, previous work learned to interpret language to troubleshoot a Windows machine (Branavan et al., 2009, 2010), win a game of Civilization (Branavan et al., 2011), play a legal game of solitaire (Eisenstein et al., 2009; Goldwasser and Roth, 2011), and navigate a map by following directions (Vogel and Jurafsky, 2010; Chen and Mooney, 2011). Even when the objective in the world is defined independently of language (e.g., in Civilization), language can provide a useful bias towards the non-linguistic end goal.

5.4 Conclusions

The main conceptual contribution of this thesis is a new semantic formalism, dependency-based compositional semantics (DCS), which has favorable linguistic, statistical, and computational properties. This enabled us to learn a semantic parser from question-answer pairs where the intermediate logical form (a DCS tree) is induced in an unsupervised manner. Our final question answering system was able to outperform current state-of-the-art systems despite requiring no annotated logical forms.

However, there is currently a large divide between our question answering system (which are natural language interfaces to databases) and open-domain question answering systems. The former focuses on understanding a question compositionally and computing the answer compositionally, while the latter focuses on retrieving and ranking answers from a large unstructured textual corpus. The former has depth; the latter has breadth. Developing methods that can both model the semantic richness of language and scale up to an open-domain setting remains an open challenge.

We believe that it is possible to push our approach in that direction. Neither DCS nor the learning algorithm is tied to having a clean rigid database, which could instead be a database generated from a noisy information extraction process. The key is to drive the learning with the desired behavior, the question-answer pairs. The latent variable is the logical form or program, which just tries to compute the desired answer by piecing together

whatever information is available. Of course, there are many open challenges ahead, but with the proper combination of linguistic, statistical, and computational insight, we hope to eventually build systems with both breadth and depth.

Bibliography

H. Alshawi, P. Chang, and M. Ringgaard. Deterministic statistical mapping of sentences to underspecified semantics. In *International Conference on Compositional Semantics (IWCS)*, 2011.

I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases – an introduction. *Journal of Natural Language Engineering*, 1:29–81, 1995.

Y. Artzi and L. Zettlemoyer. Bootstrapping semantic parsers from conversations. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2011.

J. Bos. A controlled fragment of DRT. In *Workshop on Controlled Natural Language*, pages 1–5, 2009.

S. Branavan, H. Chen, L. S. Zettlemoyer, and R. Barzilay. Reinforcement learning for mapping instructions to actions. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, Singapore, 2009. Association for Computational Linguistics.

S. Branavan, L. Zettlemoyer, and R. Barzilay. Reading between the lines: Learning to map high-level instructions to commands. In *Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2010.

S. Branavan, D. Silver, and R. Barzilay. Learning to win by reading manuals in a Monte-Carlo framework. In *Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2011.

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

D. L. Chen and R. J. Mooney. Learning to sportscast: A test of grounded language acquisition. In *International Conference on Machine Learning (ICML)*, pages 128–135. Omnipress, 2008.

D. L. Chen and R. J. Mooney. Learning to interpret natural language navigation instructions from observations. In *Association for the Advancement of Artificial Intelligence (AAAI)*, Cambridge, MA, 2011. MIT Press.

J. Clarke, D. Goldwasser, M. Chang, and D. Roth. Driving semantic parsing from the world's response. In *Computational Natural Language Learning (CoNLL)*, 2010.

M. Collins. *Head-Driven Statistical Models for Natural Language Parsing*. PhD thesis, University of Pennsylvania, 1999.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.

H. Daume, J. Langford, and D. Marcu. Search-based structured prediction. *Machine Learning Journal (MLJ)*, 75:297–325, 2009.

R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

J. Eisenstein, J. Clarke, D. Goldwasser, and D. Roth. Reading to learn: Constructing features from semantic abstracts. In *Empirical Methods in Natural Language Processing (EMNLP)*, Singapore, 2009.

R. Ge and R. J. Mooney. A statistical semantic parser that integrates syntax and semantics. In *Computational Natural Language Learning (CoNLL)*, pages 9–16, Ann Arbor, Michigan, 2005.

A. Giordani and A. Moschitti. Semantic mapping between natural language questions and SQL queries via syntactic pairing. In *International Conference on Applications of Natural Language to Information Systems*, 2009.

D. Goldwasser and D. Roth. Learning from natural instructions. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

D. Goldwasser, R. Reichart, J. Clarke, and D. Roth. Confidence driven unsupervised semantic parsing. In *Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2011.

J. Judge, A. Cahill, and J. v. Genabith. Question-bank: creating a corpus of parse-annotated questions. In *International Conference on Computational Linguistics and Association for Computational Linguistics (COLING/ACL)*, Sydney, Australia, 2006. Association for Computational Linguistics.

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.

H. Kamp, J. v. Genabith, and U. Reyle. Discourse representation theory. In *Handbook of Philosophical Logic*. 2005.

R. J. Kate and R. J. Mooney. Using string-kernels for learning semantic parsers. In *International Conference on Computational Linguistics and Association for Computational Linguistics (COLING/ACL)*, pages 913–920, Sydney, Australia, 2006. Association for Computational Linguistics.

R. J. Kate and R. J. Mooney. Learning language semantics from ambiguous supervision. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 895–900, Cambridge, MA, 2007. MIT Press.

R. J. Kate, Y. W. Wong, and R. J. Mooney. Learning to transform natural to formal languages. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 1062–1068, Cambridge, MA, 2005. MIT Press.

T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2010.

T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, and M. Steedman. Lexical generalization in CCG grammar induction for semantic parsing. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2011.

P. Liang and D. Klein. Online EM for unsupervised models. In *North American Association for Computational Linguistics (NAACL)*. Association for Computational Linguistics, 2009.

P. Liang, M. I. Jordan, and D. Klein. Learning semantic correspondences with less supervision. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, Singapore, 2009. Association for Computational Linguistics.

P. Liang, M. I. Jordan, and D. Klein. Learning programs: A hierarchical Bayesian approach. In *International Conference on Machine Learning (ICML)*. Omnipress, 2010.

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

W. Lu, H. T. Ng, W. S. Lee, and L. S. Zettlemoyer. A generative model for parsing natural language to meaning representations. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 783–792, 2008.

M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19:313–330, 1993.

S. Miller, D. Stallard, R. Bobrow, and R. Schwartz. A fully statistical approach to natural language interfaces. In *Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 1996.

R. Montague. The proper treatment of quantification in ordinary English. In *Approaches to Natural Language*, pages 221–242, 1973.

J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35:773–782, 1980.

S. Petrov, L. Barrett, R. Thibaux, and D. Klein. Learning accurate, compact, and interpretable tree annotation. In *International Conference on Computational Linguistics and Association for Computational Linguistics (COLING/ACL)*, pages 433–440. Association for Computational Linguistics, 2006.

S. T. Piantadosi, N. D. Goodman, B. A. Ellis, and J. B. Tenenbaum. A Bayesian model of the acquisition of compositional semantics. In *Proceedings of the Thirtieth Annual Conference of the Cognitive Science Society*, 2008.

A. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *International Conference on Intelligent User Interfaces (IUI)*, 2003.

M. F. Porter. An algorithm for suffix stripping. *Program*, 14:130–137, 1980.

H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22(3):400–407, 1951.

W. Schuler. Using model-theoretic semantic interpretation to guide statistical parsing and word recognition in a spoken language interface. In *Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2003.

M. Steedman. *The Syntactic Process*. MIT Press, 2000.

L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *European Conference on Machine Learning*, pages 466–477, 2001.

A. Vogel and D. Jurafsky. Learning to follow navigational directions. In *Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2010.

M. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1:1–307, 2008.

D. Warren and F. Pereira. An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics*, 8:110–122, 1982.

Y. W. Wong and R. J. Mooney. Learning for semantic parsing with statistical machine translation. In *North American Association for Computational Linguistics (NAACL)*, pages 439–446, New York City, 2006. Association for Computational Linguistics.

Y. W. Wong and R. J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *Association for Computational Linguistics (ACL)*, pages 960–967, Prague, Czech Republic, 2007. Association for Computational Linguistics.

W. A. Woods, R. M. Kaplan, and B. N. Webber. The lunar sciences natural language information system: Final report. Technical report, BBN Report 2378, Bolt Beranek and Newman Inc., 1972.

M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic proramming. In *Association for the Advancement of Artificial Intelligence (AAAI)*, Cambridge, MA, 1996. MIT Press.

L. S. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence (UAI)*, pages 658–666, 2005.

L. S. Zettlemoyer and M. Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP/CoNLL)*, pages 678–687, 2007.

Appendix A

Details

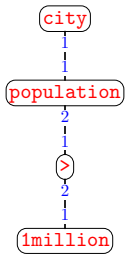
A.1 Denotation Computation

This section describes the details of how we compute the denotations of basic DCS trees according to (2.15). We could simply convert DCS trees into SQL or relational algebra and use an off-the-shelf database management system (DBMS). After all, our bottom-up approach to computing the denotation of a DCS tree is not necessarily optimal. For example, if the predicate at the root contains only one tuple, then a top-down approach would most likely be preferable, and a global query optimizer might choose that query execution plan. While this might have been a viable route, we use a custom implementation for several reasons:

1. We wanted to present a simple algorithm with a transparent computational complexity;
2. DCS trees can be built from custom predicates (e.g., for generalized quantification and superlatives), which are more cumbersome to incorporate into standard engines;
3. We need to compute denotations not of one DCS tree but of a forest of DCS trees which share many subtrees. Exploiting this structure requires a particular form of multiple query optimization which is readily incorporated into our construction mechanism.

In (2.15), we have reduced the computation of denotations to two operations, join and project, but there is still much to be said regarding how these operations are implemented. For example, for two finite sets of tuples, A and B, their join, $A \bowtie_{j,j'} B$ can either be computed by enumerating the elements of A and querying B for each one, or by enumerating B and querying A. We can estimate the costs of these operations and choose the cheaper option. We also build various indices to make these operations more efficient. Many of these ideas are based on standard database query optimization techniques.

Recall that our worlds are only conceptually databases, for they contain infinite sets of tuples, which arise due to predicates such as >. Performing join and project operations on



	Set-expression	Denotation
city	explicit-predicate1	$\{(x) : \texttt{city}(x) \wedge \exists y.\texttt{population}(x, y) \wedge y > 10^6\}$
population	explicit-predicate2	$\{(x, y) : \texttt{population}(x, y) \wedge y > 10^6\}$
>	$>\{\cdot_2 = 10^6\}$	$\{(x, 10^6) : x > 10^6\}$
1million	1million	$\{(10^6)\}$

Figure A.1: Shows the computation and representation of the denotations for the DCS tree corresponding to *city with population greater than 1 million*. For each node, we compute the denotation for the subtree rooted at that node, which is represented by set-expressions.

these infinite sets can produce other infinite sets. To handle these manipulations, we need a way of representing these infinite sets symbolically.

A.1.1 Set-expressions

We define a language for representing sets of tuples and show how to perform join and project operations on expressions in that language (*set-expressions*). As a running example, consider computing the denotation for the DCS tree corresponding to *city with population greater than 1 million*, shown in Figure A.1.

Our language for representing sets is given by the following grammar:

$$\begin{aligned}
\text{set} \quad ::= \quad & \text{predicate} && \text{[base case]} && \text{(A.1)}\\
| \quad & \text{set}\{\cdot_{\mathbf{i}} = \mathbf{v}\} && \text{[select]} && \text{(A.2)}\\
| \quad & \text{set}[\mathbf{i}] && \text{[project]} && \text{(A.3)}\\
| \quad & \text{set} \times \text{set} && \text{[product]} && \text{(A.4)}\\
| \quad & \text{set} \cup \text{set} && \text{[union]} && \text{(A.5)}
\end{aligned}$$

Perhaps not surprisingly, set-expressions bear a striking similarity to relational algebra. Each set-expression e denotes the set $[\![e]\!]_w$ specified below for each case:

- Predicate (e.g., `population`): a predicate $p \in \mathcal{P}$ represents its semantics $[\![p]\!]_w = w(p)$.
- Filter (e.g., $>\{\cdot_2 = 10^6\}$): $[\![e\{\cdot_{\mathbf{i}} = \mathbf{v}\}]\!]_w = \{x \in [\![e]\!]_w : x[\mathbf{i}] = \mathbf{v}\}$.
- Project (e.g., `population`[2]): $[\![e[\mathbf{i}]]\!]_w = \{x_{\mathbf{i}} : x \in [\![e]\!]_w\}$.
- Product (e.g., `population` $\times$ `area`): $[\![e_1 \times e_2]\!]_w = \{x + y : x \in [\![e_1]\!]_w, y \in [\![e_2]\!]_w\}$.

- Union (e.g., `city` $\cup$ `river`): $[\![e_1 \cup e_2]\!]_w = [\![e_1]\!]_w \cup [\![e_2]\!]_w$.

At this point, it is worth stressing the difference between three kinds of objects: DCS trees, set-expressions, and the actual sets of tuples in the world. DCS trees have denotations given by (2.15), which are sets of tuples. These denotations are computed by manipulating set-expressions, which themselves denote the same sets of tuples in the world.

A.1.2 Join and Project on Set-Expressions

We now define join and project operations that operate on set-expressions rather than sets of tuples. To distinguish these operations, we use $\mathbf{J}(e_1, j, j', e_2)$ and $\mathbf{P}(\mathbf{i}, e)$ for the operations that work on set-expressions, reserving $A \bowtie_{j,j'} B$ and $A[\mathbf{i}]$ for the operations that work on sets of tuples.

For convenience, define a caching function $\mathbf{C}_w(A)$, which takes a set of tuples A and returns a fresh new predicate p with denotation A (think of $\mathbf{C}_w$ as a technical trick which implements the inverse of $[\![\cdot]\!]_w$):

$$\mathbf{C}_w(A) \stackrel{\text{def}}{=} p \text{ such that } w(p) = A. \tag{A.6}$$

We also define a reduction function $\mathbf{R}(e)$, which takes a set-expression e and caches the result if e denotes a finite set:

$$\mathbf{R}(e) \stackrel{\text{def}}{=} \begin{cases} \mathbf{C}_w([\![e]\!]_w) & \text{if } |[\![e]\!]_w| < \infty \\ e & \text{otherwise.} \end{cases} \tag{A.7}$$

The idea is that we want to reduce set-expressions to predicates whenever possible.

We implement the join operation $\mathbf{J}$ using two operations: (i) enumerate $\mathbf{E}(e)$, which returns the set of tuples denoted by e; and (ii) select $\mathbf{S}(\mathbf{i}, \mathbf{v}, e)$, which returns a set-expression denoting the subset $[\![e]\!]_w$ which matches the select condition. Using enumerate and select, we implement the join of two set-expressions e_1 and e_2 by enumerating the tuples in e_1 and using each one to select e_2 or vice-versa:

$$\mathbf{J}(e_1, j, j', e_2) \stackrel{\text{def}}{=} \bigcup_{x \in \mathbf{E}(e_1)} \mathbf{C}_w(\{x\}) \times \mathbf{S}(j', e_{1j}, e_2) \quad \text{or} \quad \bigcup_{x \in \mathbf{E}(e_2)} \mathbf{S}(j, e_{2j'}, e_1) \times \mathbf{C}_w(\{x\}). \tag{A.8}$$

Now, it suffices to define enumerate ($\mathbf{E}$), select ($\mathbf{S}$), and project ($\mathbf{P}$) for each type of set-expression. First, the enumerate operation $\mathbf{E}$ simply computes the semantics of set-expressions, with the added caveat that the set of tuples must be finite:

$$\mathbf{E}(e) \stackrel{\text{def}}{=} [\![e]\!]_w \text{ if } |[\![e]\!]_w| < \infty. \tag{A.9}$$

The select operation $\mathbf{S}$ is defined for each type of set-expression as follows:

- Predicate: $\mathbf{S}(\mathbf{i}, \mathbf{v}, p) = \mathbf{R}(p\{\cdot_{\mathbf{i}} = \mathbf{v}\})$. If we know that the resulting set of tuples is finite, then we simply return a predicate corresponding to those tuples. For finite predicates such as $p = \texttt{city}$, the result is always finite. For infinite predicates where the select condition is restrictive enough, such as $p = \texttt{count}$ with $\mathbf{i} = (1)$ and $\mathbf{v} = (\{(1), (2), (3)\})$, then the resulting set is still finite—in this case, $\{(\{(1), (2), (3)\}, 3)\}$. Otherwise, we represent the computation symbolically and return $p\{\cdot_{\mathbf{i}} = \mathbf{v}\}$.
- Select: $\mathbf{S}(\mathbf{i}, \mathbf{v}, e\{\cdot_{\mathbf{j}} = \mathbf{t}\}) = \mathbf{R}(p\{\cdot_{\mathbf{i}+\mathbf{j}} = \mathbf{v} + \mathbf{t}\})$ (simply concatenate the select conditions).
- Project: $\mathbf{S}(\mathbf{i}, \mathbf{v}, e[\mathbf{j}]) = \mathbf{R}(e\{\cdot_{\mathbf{j}_\mathbf{i}} = \mathbf{v}\}[\mathbf{j}])$ (push the select operation inside the project).
- Product: $\mathbf{S}(\mathbf{i}, \mathbf{v}, e_1 \times e_2) = \mathbf{R}(\mathbf{S}(\mathbf{i}_\mathbf{j}, \mathbf{v}_\mathbf{j}, e_1) \cap \mathbf{S}(\mathbf{i}_{-\mathbf{j}}, \mathbf{v}_{-\mathbf{j}}, e_2))$, where $\mathbf{j} = \{j : i_j < \textsc{Arity}(\llbracket e_1 \rrbracket_w)\}$ are the indices corresponding to the e_1 part.
- Union: $\mathbf{S}(\mathbf{i}, \mathbf{v}, e_1 \cup e_2) = \mathbf{R}(\mathbf{S}(\mathbf{i}, \mathbf{v}, e_1) \cup \mathbf{S}(\mathbf{i}, \mathbf{v}, e_2))$.

The project operation $\mathbf{P}$ is defined for each type of set-expression as follows:

- Predicate: $\mathbf{P}(\mathbf{i}, p) = \mathbf{R}(p[\mathbf{i}])$.
- Select: $\mathbf{P}(\mathbf{i}, e\{\cdot_{\mathbf{j}} = \mathbf{v}\}) = e\{\cdot_{\mathbf{j}} = \mathbf{v}\}[\mathbf{i}]$ (leave the project outside the select).
- Project: $\mathbf{P}(\mathbf{i}, e[\mathbf{j}]) = \mathbf{R}(e[\mathbf{j}_\mathbf{i}])$ (accumulate the two project operations).
- Product: $\mathbf{P}(\mathbf{i}, e_1 \times e_2) = \mathbf{R}(\mathbf{P}(\mathbf{i}_\mathbf{j}, e_1) \times \mathbf{P}(\mathbf{i}_{-\mathbf{j}}, e_2))$, where $\mathbf{j} = \{j : i_j < \textsc{Arity}(\llbracket e_1 \rrbracket_w)\}$ are the indices corresponding to the e_1 part.
- Union: $\mathbf{P}(\mathbf{i}, e_1 \cup e_2) = \mathbf{R}(\mathbf{P}(\mathbf{i}, e_1) \cup \mathbf{P}(\mathbf{i}, e_2))$.

The basic strategy when operating on set-expressions is to try to push the select operations as far in as possible, followed by project operations. Whenever we detect that a set-expression denotes a finite set, we evaluate the set-expression to get a concrete set of tuples and assign a new predicate to it. Note that it is not necessarily optimal to always construct a new predicate when possible. However, note that this does result in caching the results of the computation, which can be reused later.

At the root of the DCS tree, we obtain a set-expression. If the set-expression is a single predicate, we happily return the corresponding set of tuples as the final answer. Otherwise, we say that we have failed to compute the denotation, because the set is probably infinite. This happens mostly in strange cases, such as with the DCS tree $\langle \emptyset; \frac{1}{1}\!:\langle \texttt{count}; \frac{2}{1}\!:\langle \texttt{3} \rangle \rangle \rangle$, whose denotation is all sets (of primitive values) with cardinality 3. Though this denotation is semantically meaningful, it never arises in our intended applications. Thus, the failure to compute a denotation serves as a pragmatic filter on the set of DCS trees (see Section 2.6.3).

Let us walk through the example in Figure A.1. At the bottom of the DCS tree, we start with the set-expression corresponding to the constant predicate `1million`. We then

join `>` with `1million`, which results in the set-expression $\texttt{>}\{\cdot_2 = 10^6\}$. Note that we cannot reduce this set-expression to a predicate because it has an infinite denotation. Next, we join $\texttt{>}\{\cdot_2 = 10^6\}$ with `population`, which does produce a finite set, which is cached as `explicit-predicate2`. Finally, `explicit-predicate2` is joined with `city`, which yields the answer, which can be stored in `explicit-predicate1`.

Local Query Optimization The computation of the denotation is driven by the join operation (A.8). There are two points worth making. First, the core computation is in taking a set-expression (usually a predicate p) and performing a select operation ($\mathbf{S}(j, x, p)$). To avoid naïvely enumerating the tuples in $\mathbf{E}(p)$ and checking the select condition on each tuple, we perform some indexing to improve efficiency. Specifically, let $A = w(p)$ be the set of tuples. For each component $j = 1, \ldots, \textsc{Arity}(A)$, we maintain a map I_A from each component j and value v to the set of matching tuples in A; formally,

$$I_A(j, x) = \{t \in A : t_j = x\}. \tag{A.10}$$

Now, each $\mathbf{S}(j, x, p)$ operation simply returns $I_{w(p)}(j, x)$ using a single lookup.

Second, when joining two set-expressions e_1 and e_2 (A.8), there is the choice of whether to enumerate e_1 or e_2. We first estimate the costs of each option and choose the cheaper one. Specifically, let $N_A(j)$ be the average size of $I_A(j, x)$, where v is drawn from a uniform distribution over $A[j]$:

$$N_A(j) = \frac{1}{|A[j]|} \sum_{x \in A[j]} |I_A(j, x)|. \tag{A.11}$$

More sophisticated query optimization techniques can be used to improve this estimate, but we do not pursue them here. For example, suppose we would like to join A and B, which both have arity 1. Then the only possible join relation is $\frac{1}{1}$, which corresponds to set intersection. The cost for the enumerate-A-query-B option is $|A|$ and the cost form the enumerate-B-query-A option is $|B|$ (recall that querying is constant time). We would choose the former option if $|A| < |B|$ and the latter otherwise.

The other common case is when A has arity 2 and B has arity 1; assume the join relation is $\frac{j}{1}$ with $j \in \{1, 2\}$. The enumerate-A-query-B option is the same as before, with a cost of $|A|$. The enumerate-B-query-A option is a bit different, because querying A is no longer just a simple set-containment check. Instead, each query to A returns a set of arity 2 tuples. We take the union of these results: $\cup_{x \in B} I_A(j, x)$; this operation has an estimated cost of $|B| N_A(j)$.