Value-Agnostic Conversational Semantic Parsing

Value-Agnostic Conversational Semantic Parsing

Emmanouil Antonios Platanios, Adam Pauls, Subhro Roy, Yuchen Zhang, Alex Kyte, Alan Guo, Sam Thomson, Jayant Krishnamurthy, Jason Wolfe, Jacob Andreas, Dan Klein
Microsoft Semantic Machines
sminfo@microsoft.com

Abstract

Conversational semantic parsers map user utterances to executable programs given dialogue histories composed of previous utterances, programs, and system responses. Existing parsers typically condition on rich representations of history that include the complete set of values and computations previously discussed. We propose a model that *abstracts over values* to focus prediction on type- and function-level context. This approach provides a compact encoding of dialogue histories and predicted programs, improving generalization and computational efficiency. Our model incorporates several other components, including an atomic span copy operation and structural enforcement of well-formedness constraints on predicted programs, that are particularly advantageous in the low-data regime. Trained on the SMCALFLOW and TREEDST datasets, our model outperforms prior work by 7.3% and 10.6% respectively in terms of absolute accuracy. Trained on only a thousand examples from each dataset, it outperforms strong baselines by 12.4% and 6.4%. These results indicate that simple representations are key to effective generalization in conversational semantic parsing.

1 Introduction

Conversational semantic parsers, which translate natural language utterances into executable programs while incorporating conversational context, play an increasingly central role in systems for interactive data analysis (Yu et al., 2019), instruction following (Guu et al., 2017), and task-oriented dialogue (Zettlemoyer and Collins, 2009). An example of this task is shown in Figure 1. Typical models are based on an autoregressive sequence prediction approach, in which a detailed representation of the dialogue history is concatenated to the input sequence, and predictors condition on this sequence and all previously generated components of the output (Suhr et al., 2018). While this approach can capture arbitrary dependencies between inputs and outputs, it comes at the cost of sample- and computational inefficiency.

We propose a new "value-agnostic" approach to contextual semantic parsing driven by *type-based* representations of the dialogue history and *function-based* representations of the generated programs. Types and functions have long served as a foundation for formal reasoning about programs, but their use in neural semantic parsing has been limited, e.g., to constraining the hypothesis space (Krishnamurthy et al., 2017), guiding data augmentation (Jia and Liang, 2016), and coarsening in coarse-to-fine models (Dong and Lapata, 2018). We show that representing conversation histories and partial programs via the types and functions they contain enables fast, accurate, and sample-efficient contextual semantic parsing. We propose a neural encoder–decoder contextual semantic parsing model which, in contrast to prior work:

1. uses a compact yet informative representation of discourse context in the encoder that considers only the *types of salient entities* that were predicted by the model in previous turns or that appeared in the execution results of the predicted programs, and

2. conditions the decoder state on the *sequence of function invocations* so far, without conditioning on any concrete values passed as arguments to the functions.

Our model substantially improves upon the best published results on the SMCALFLOW (Semantic Machines et al., 2020) and TREEDST (Cheng et al., 2020) conversational semantic parsing datasets, improving model performance by 7.3% and 10.6%, respectively, in terms of absolute accuracy. In further experiments aimed at quantifying sample efficiency,

Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, pages 3666–3681
August 1–6, 2021.

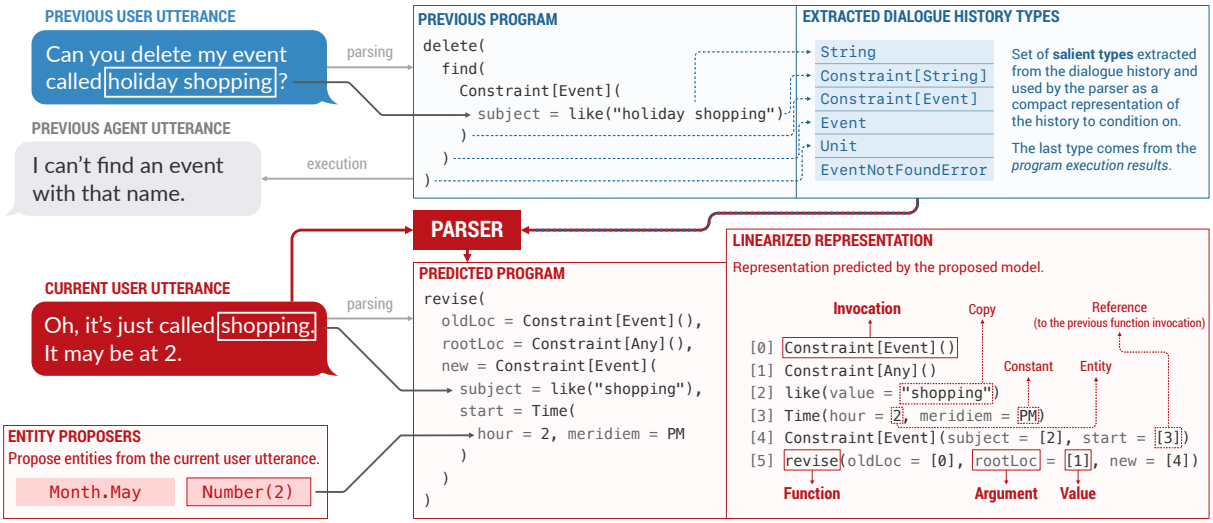

Figure 1: Illustration of the conversational semantic parsing problem that we focus on and the representations that we use. The previous turn user utterance and the previous program are shown in **blue** on the top. The dialogue history representation extracted using our approach is shown on the top right. The current turn user utterance is shown in **red** on the bottom left. The current utterance, the set of proposed entities, and the extracted dialogue history representation form the input to our parser. Given this input, the parser predicts a program that is shown on the bottom right (in **red** rectangles).

it improves accuracy by 12.4% and 6.4% respectively when trained on only a thousand examples from each dataset. Our model is also effective at non-contextual semantic parsing, matching state-of-the-art results on the JOBS, GEOQUERY, and ATIS datasets (Dong and Lapata, 2016). This is achieved while also reducing the test time computational cost by a factor of 10 (from 80ms per utterance down to 8ms when running on the same machine; more details are provided in Appendix H), when compared to our fastest baseline, which makes it usable as part of a real-time conversational system.

One conclusion from these experiments is that most semantic parses have structures that depend only weakly on the *values* that appear in the dialogue history or in the programs themselves. Our experiments find that hiding values alone results in a 2.6% accuracy improvement in the low-data regime. By treating types and functions, rather than values, as the main ingredients in learned representations for semantic parsing, we improve model accuracy and sample efficiency across a diverse set of language understanding problems, while also significantly reducing computational costs.

2 Proposed Model

Our goal is to map natural language *utterances* to *programs* while incorporating context from *dialogue histories* (i.e., past utterances and their associated programs and execution results). We model a program as a sequenceo of *function invocations*, each consisting of a function and zero or more *argument values*, as illustrated at the lower right of Figure 1. The argument values can be either literal values or references to results of previous function invocations. The ability to reference previous elements of the sequence, sometimes called a *target-side copy*, allows us to construct programs that involve re-entrancies. Owing to this referential structure, a program can be equivalently represented as a directed acyclic graph (see e.g., Jones et al., 2012; Zhang et al., 2019).

We propose a Transformer-based (Vaswani et al., 2017) encoder–decoder model that predicts programs by generating function invocations sequentially, where each invocation can draw its arguments from an inventory of values (§2.5)—possibly copied from the utterance—and the results of previous function invocations in the current program. The encoder (§2.2) transforms a natural language utterance and a dialogue history to a continuous representation. Subsequently, the decoder (§2.3) uses this representation to define an autoregressive distribution over function invocation sequences and chooses a high-probability sequence by performing beam search. As our experiments (§3) will show, a naïve encoding of the complete dialogue history and program results in poor model accuracy.

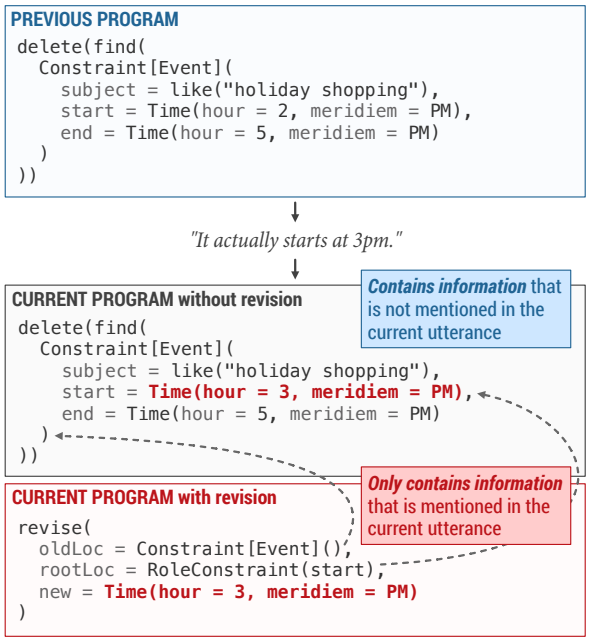

Figure 2: Illustration of the `revise` meta-computation operator (§2.1) used in our program representations. This operator can remove the need to copy program fragments from the dialogue history.

2.1 Preliminaries

Our approach assumes that programs have type annotations on all values and function calls, similar to the setting of Krishnamurthy et al. (2017).[1] Furthermore, we assume that program prediction is *local* in that it does not require program fragments to be copied from the dialogue history (but may still depend on history in other ways). Several formalisms, including the *typed references* of Zettlemoyer and Collins (2009) and the *meta-computation operators* of Semantic Machines et al. (2020), make it possible to produce local program annotations even for dialogues like the one depicted in Figure 2, which reuse past computations. We transformed the datasets in our experiments to use such meta-computation operators (see Appendix C).

We also optionally make use of *entity proposers*, similar to Krishnamurthy et al. (2017), which annotate spans from the current utterance with typed values. For example, the span *"one"* in *"Change it to one"* might be annotated with the value `1` of type `Number`. These values are scored by the decoder along with other values that it considers (§2.5) when predicting argument values for function invocations. Using entity proposers aims to help the model generalize better to previously unseen values that can be recognized in the utterance using hard-coded heuristics (e.g., regular expressions), auxiliary training data, or other runtime information (e.g., a contact list). In our experiments we make use of simple proposers that recognize numbers, months, holidays, and days of the week, but one could define proposers for arbitrary values (e.g., song titles). As described in §2.5, certain values can also be predicted directly without the use of an entity proposer.

[1]This requirement can be trivially satisfied by assigning all expressions the same type, but in practice defining a set of type declarations for the datasets in our experiments was not difficult (refer to Appendix C for details).

2.2 Encoder

The encoder, shown in Figure 3, maps a natural language utterance to a continuous representation. Like many neural sequence-to-sequence models, we produce a contextualized token representation of the utterance, $\boldsymbol{H}_{\text{utt}} \in \mathbb{R}^{U \times h_{\text{enc}}}$, where U is the number of tokens and h_{enc} is the dimensionality of their embeddings. We use a Transformer encoder (Vaswani et al., 2017), optionally initialized using the BERT pretraining scheme (Devlin et al., 2019). Next, we need to encode the dialogue history and combine its representation with $\boldsymbol{H}_{\text{utt}}$ to produce *history-contextualized* utterance token embeddings.

Prior work has incorporated history information by linearizing it and treating it as part of the input utterance (Cheng et al., 2018; Semantic Machines et al., 2020; Aghajanyan et al., 2020). While flexible and easy to implement, this approach presents a number of challenges. In complex dialogues, history encodings can grow extremely long relative to the user utterance, which: (i) increases the risk of overfitting, (ii) increases computational costs (because attentions have to be computed over long sequences), and (iii) necessitates using small batch sizes during training, making optimization difficult.

Thanks to the *predictive locality* of our representations (§2.1), our decoder (§2.3) never needs to retrieve values or program fragments from the dialogue history. Instead, context enters into programs primarily when programs use referring expressions that point to past computations, or revision expressions that modify them. Even though this allows us to dramatically simplify the dialogue history representation, effective generation of referring expressions still requires knowing something about the past. For example, for the utterance *"What's next?"* the model needs to determine what *"What"* refers to. Perhaps more interestingly, the presence of dates in recent

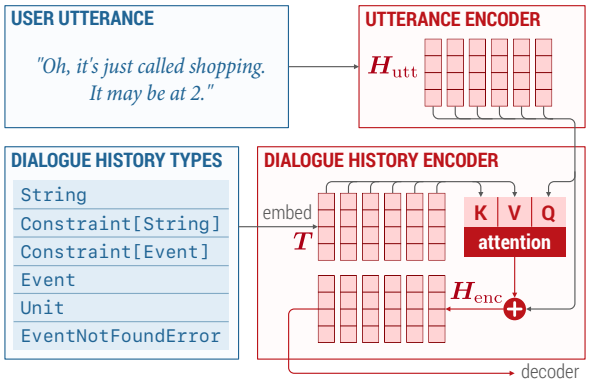

Figure 3: Illustration of our encoder (§2.2), using the example of Figure 1. The utterance is processed by a Transformer-based (Vaswani et al., 2017) encoder and combined with information extracted from the set of dialogue history types using multi-head attention.

turns (or values that *have* dates, such as meetings) should make the decoder more eager to generate referring calls that retrieve dates from the dialogue history; especially so if other words in the current utterance hint that dates may be useful and yet date values cannot be constructed directly from the current utterance. Subsequent steps of the decoder which are triggered by these other words can produce functions that consume the referred dates.

We thus hypothesize that it suffices to strip the dialogue history down to its constituent types, hiding all other information.[2] Specifically, we extract a set $\mathcal{T}$ of types that appear in the dialogue history up to m turns back, where $m = 1$ in our experiments.[3] Our encoder then transforms $\boldsymbol{H}_{\text{utt}}$ into a sequence of history-contextualized embeddings $\boldsymbol{H}_{\text{enc}}$ by allowing each token to attend over $\mathcal{T}$. This is motivated by the fact that, in many cases, dialogue history is important for determining the meaning of specific tokens in the utterance, rather than the whole utterance. Specifically, we learn embeddings $\boldsymbol{T} \in \mathbb{R}^{|\mathcal{T}| \times h_{\text{type}}}$ for the extracted types, where h_{type} is the embedding size, and use the attention mechanism of Vaswani et al. (2017) to contextualize $\boldsymbol{H}_{\text{utt}}$:

$$\boldsymbol{H}_{\text{enc}} \triangleq \boldsymbol{H}_{\text{utt}} + \text{MHA}(\underbrace{\boldsymbol{H}_{\text{utt}}}_{\text{Queries}}, \underbrace{\boldsymbol{T}}_{\text{Keys}}, \underbrace{\boldsymbol{T}}_{\text{Values}}), \quad (1)$$

where "MHA" stands for multi-head attention, and each head applies a separate linear transformation to the queries, keys, and values. Intuitively, each utterance-contextualized token is further contextualized in (1) by adding to it a mixture of embeddings of elements in $\mathcal{T}$, where the mixture coefficients depends only on that utterance-contextualized token. This encoder is illustrated in Figure 3. As we show in §3.1, using this mechanism performs better than the naïve approach of appending a set-of-types vector to $\boldsymbol{H}_{\text{utt}}$.

[2]For the previous example, if the type `List[Event]` appeared in the history then we may infer that *"What"* probably refers to an `Event`.

[3]We experimented with different values of m and found that increasing it results in worse performance, presumably due to overfitting.

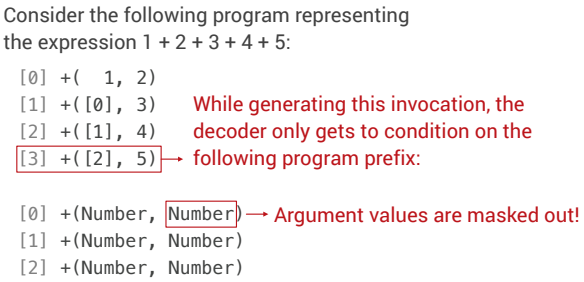

Figure 4: Illustration showing the way in which our decoder is *value-agnostic*. Specifically, it shows which part of the generated program prefix, our decoder conditions on while generating programs (§2.3).

2.3 Decoder: Programs

The decoder uses the history-contextualized representation $\boldsymbol{H}_{\text{enc}}$ of the current utterance to predict a distribution over the program $\boldsymbol{\pi}$ that corresponds to that utterance. Each successive "line" π_i of $\boldsymbol{\pi}$ invokes a function f_i on an argument value tuple $(v_{i1}, v_{i2}, \ldots, v_{iA_i})$, where A_i is the number of (formal) arguments of f_i. Applying f_i to this ordered tuple results in the invocation $f_i(a_{i1} = v_{i1}, a_{i2} = v_{i2}, \ldots)$, where $(a_{i1}, a_{i2}, \ldots, a_{iA_i})$ name the formal arguments of f_i. Each predicted *value* v_{ij} can be the result of a previous function invocation, a constant value, a value copied from the current utterance, or a proposed entity (§2.1), as illustrated in the lower right corner of Figure 1. These different argument *sources* are described in §2.5. Formally, the decoder defines a distribution of programs $\boldsymbol{\pi}$:

$$p(\boldsymbol{\pi} \mid \boldsymbol{H}_{\text{enc}}) = \prod_{i=1}^{P} p(\pi_i \mid f_{<i}, \boldsymbol{H}_{\text{enc}}), \quad (2)$$

where P is the number of function invocations in the program, and $f_{<i} \triangleq \{f_1, \ldots, f_{i-1}\}$. Additionally, we assume that argument values are conditionally independent given f_i and $f_{<i}$, resulting in:

$$p(\pi_i \mid f_{<i}) = \underbrace{p(f_i \mid f_{<i})}_{\substack{\text{function} \\ \text{scoring}}} \prod_{j=1}^{A_i} \underbrace{p(v_{ij} \mid f_{<i}, f_i)}_{\substack{\text{argument value} \\ \text{scoring}}}, \quad (3)$$

where we have elided the conditioning on $\boldsymbol{H}_{\text{enc}}$. Here, *functions depend only on previous functions*

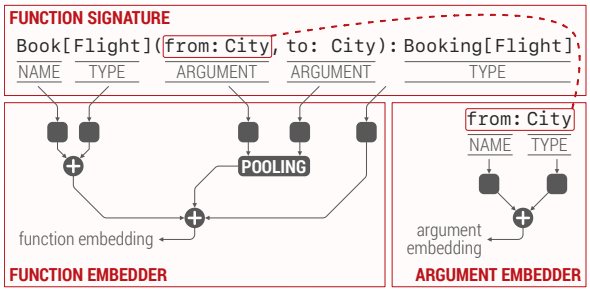

Figure 5: Illustration of our function encoder (§2.4), using a simplified example function signature.

(not their argument values or results) and *argument values depend only on their calling function* (not on one another or any of the previous argument values).[4] This is illustrated in Figure 4. In addition to providing an important inductive bias, these independence assumptions allow our inference procedure to efficiently score all possible function invocations at step i, given the ones at previous steps, at once (i.e., function and argument value assignments together), resulting in an efficient search algorithm (§2.6). Note that there is also a corresponding disadvantage (as in many machine translation models) that a meaningful phrase in the utterance could be independently selected for multiple arguments, or not selected at all, but we did not encounter this issue in our experiments; we rely on the model training to evade this problem through the dependence on $\boldsymbol{H}_{\text{enc}}$.

2.4 Decoder: Functions

In Equation 3, the sequence of functions $f_1, f_2, \ldots$ in the current program is modeled by $\prod_i p(f_i \,|\, f_{<i}, \boldsymbol{H}_{\text{enc}})$. We use a standard autoregressive Transformer decoder that can also attend to the utterance encoding $\boldsymbol{H}_{\text{enc}}$ (§2.2), as done by Vaswani et al. (2017). Our decoder generates sequences over the *vocabulary of functions*. This means that each function f_i needs an embedding $\boldsymbol{f}_i$ (used as both an input to the decoder and an output), which we construct *compositionally*.

We assume that each unique function f has a *type signature* that specifies a name n, a list of type parameters $\{\tau_1, \ldots, \tau_T\}$ (to support polymorphism),[5] a list of argument names and types $((a_1, t_1), \ldots, (a_A, t_A))$, and a result type r. An example is shown in Figure 5. We encode the function and argument names using the utterance encoder of §2.2 and learn embeddings for the types, to obtain $(\boldsymbol{n}, \boldsymbol{r})$, $\{\boldsymbol{\tau}_1, \ldots, \boldsymbol{\tau}_T\}$, and $\{(\boldsymbol{a}_1, \boldsymbol{t}_1), \ldots, (\boldsymbol{a}_A, \boldsymbol{t}_A)\}$. Then, we construct an embedding for each function as follows:

$$\boldsymbol{a} = \text{Pool}(\boldsymbol{a}_1 + \boldsymbol{t}_1, \ldots, \boldsymbol{a}_A + \boldsymbol{t}_A), \quad (4)$$

$$\boldsymbol{f} = \boldsymbol{n} + \text{Pool}(\boldsymbol{\tau}_1, \ldots, \boldsymbol{\tau}_T) + \boldsymbol{a} + \boldsymbol{r}, \quad (5)$$

where "Pool" is the max-pooling operation which is invariant to the arguments' order.

Our main motivation for this function embedding mechanism is the ability to take cues from the user utterance (e.g., due to a function being named similarly to a word appearing in the utterance). If the functions and their arguments have names that are semantically similar to corresponding utterance parts, then this approach enables zero-shot generalization.[6] However, there is an additional potential benefit from parameter sharing due to the compositional structure of the embeddings (see e.g., Baroni, 2020).

2.5 Decoder: Argument Values

This section describes the implementation of the argument predictor $p(v_{ij} \mid f_{<i}, f_i)$. There are four different kinds of *sources* that can be used to fill each available argument slot: *references* to previous function invocations, *constants* from a static vocabulary, *copies* that copy string values from the utterance, and *entities* that come from entity proposers (§2.1). Many sources might propose the same value, including multiple sources of the same kind. For example, there may be multiple spans in the utterance that produce the same string value in a program, or an entity may be proposed that is also available as a constant. To address this, we marginalize over the sources of each value:

$$p(v_{ij} \mid f_{<i}, f_i) = \sum_{s \in \mathcal{S}(v_{ij})} p(v_{ij}, s \,|\, f_{<i}, f_i), \quad (6)$$

where v_{ij} represents a possible value for the argument named a_{ij}, and $s \in \mathcal{S}(v_{ij})$ ranges over the possible sources for that value. For example, given the utterance *"Change that one to 1:30pm"* and the value `1`, the set $\mathcal{S}(\texttt{1})$ may contain entities that correspond to both *"one"* and *"1"* from the utterance.

[4] We also tried defining a *jointly normalized* distribution over entire function invocations (Appendix A), but found that it results in a higher training cost for no accuracy benefits.

[5] The type parameters could themselves be parameterized, but we ignore this here for simplicity of exposition.

[6] The data may contain *overloaded* functions that have the same name but different type signatures (e.g., due to optional arguments). The overloads are given distinct identifiers f, but they often share argument names, resulting in at least partially shared embeddings.

The argument scoring mechanism considers the last-layer decoder state $\boldsymbol{h}^i_{\text{dec}}$ that was used to predict f_i via $p(f_i \mid f_{<i}) \propto \exp(\boldsymbol{f}_i^\top \boldsymbol{h}^i_{\text{dec}})$. We specialize this decoder state to argument a_{ij} as follows:

$$\boldsymbol{h}^{i,a_{ij}}_{\text{dec}} \triangleq \hat{\boldsymbol{h}}^i_{\text{dec}} \odot \tanh(\boldsymbol{f}_i + \boldsymbol{a}_{ij}), \quad (7)$$

where $\odot$ represents elementwise multiplication, $\boldsymbol{f}_i$ is the embedding of the current function f_i, $\boldsymbol{a}_{ij}$ is the encoding of argument a_{ij} as defined in §2.4, and $\hat{\boldsymbol{h}}_{\text{dec}}$ is a projection of $\boldsymbol{h}_{\text{dec}}$ to the necessary dimensionality. Intuitively, $\tanh(\boldsymbol{f}_i + \boldsymbol{a}_{ij})$ acts as a gating function over the decoder state, deciding what is relevant when scoring values for argument a_{ij}. This argument-specific decoder state is then combined with a value embedding to produce a probability for each (sourced) value assignment:

$$p(v, s \mid f_{<i}, f_i) \propto \exp\left\{\tilde{\boldsymbol{v}}^\top (\boldsymbol{h}^{i,a}_{\text{dec}} + \boldsymbol{w}_a^{\text{kind}(s)}) + b_a^{\text{kind}(s)}\right\}, \quad (8)$$

where a is the argument name a_{ij}, $\text{kind}(s) \in \{\texttt{REFERENCE}, \texttt{CONSTANT}, \texttt{COPY}, \texttt{ENTITY}\}$, $\tilde{\boldsymbol{v}}$ is the embedding of (v, s) which is described next, and $\boldsymbol{w}_a^k$ and b_a^k are model parameters that are specific to a and the kind of the source s.

References. References are pointers to the return values of previous function invocations. If the source s for the proposed value v is the result of the k^{th} invocation (where $k < i$), we take its embedding $\tilde{\boldsymbol{v}}$ to be a projection of $\boldsymbol{h}^k_{\text{dec}}$ that was used to predict that invocation's function and arguments.

Constants. Constants are values that are always proposed, so the decoder always has the option of generating them. If the source s for the proposed value v is a constant, we embed it by applying the utterance encoder on a string rendering of the value. The set of constants is automatically extracted from the training data (see Appendix B).

Copies. Copies are string values that correspond to substrings of the user utterance (e.g., person names). String values can only enter the program through copying, as they are not in the set of constants (i.e., they cannot be "hallucinated" by the model; see Pasupat and Liang, 2015; Nie et al., 2019). One might try to construct an approach based on a standard token-based copy mechanism (e.g., Gu et al., 2016). However, this would allow copying non-contiguous spans and would also require marginalizing over identical tokens as opposed to spans, resulting in more ambiguity. Instead, we propose a mechanism that enables the decoder to copy *contiguous spans* directly from the utterance. Its goal is to produce a score for each of the $U(U+1)/2$ possible utterance spans. Naïvely, this would result in a computational cost that is quadratic in the utterance length U, and so we instead chose a simple scoring model that avoids it. Similar to Stern et al. (2017) and Kuribayashi et al. (2019), we assume that the score for a span factorizes, and define the embedding of each span value as the concatenation of the contextual embeddings of the first and last tokens of the span, $\tilde{\boldsymbol{v}} = [\boldsymbol{h}^{k_{\text{start}}}_{\text{utt}}; \boldsymbol{h}^{k_{\text{end}}}_{\text{utt}}]$. To compute the copy scores we also concatenate $\boldsymbol{h}^{i,a}_{\text{dec}}$ with itself in Equation 8.

Entities. Entities are treated the same way as copies, except that instead of scoring all spans of the input, we only score spans proposed by the external *entity proposers* discussed in §2.1. Specifically, the proposers provide the model with a list of candidate entities that are each described by an utterance span and an associated value. The candidates are scored using an identical mechanism to the one used for scoring copies. This means that, for example, the string *"sept"* could be linked to the value `Month.September` even though the string representations do not match perfectly.

Type Checking. When scoring argument values for function f_i, we know the argument types, as they are specified in the function's signature. This enables us to use a *type checking* mechanism that allows the decoder to directly exclude values with mismatching types. For references, the value types can be obtained by looking up the result types of the corresponding function signatures. Additionally, the types are always pre-specified for constants and entities, and copies are only supported for a subset of types (e.g., `String`, `PersonName`; see Appendix B). The type checking mechanism sets $p(v_{ij} \mid f_{<i}, f_i) = 0$ whenever v_{ij} has a different type than the expected type for a_{ij}. Finally, because copies can correspond to multiple types, we also add a *type matching* term to the copy score. This term is defined as the inner product of the argument type embedding and a (learnable) linear projection of $\boldsymbol{h}^{k_{\text{start}}}_{\text{utt}}$ and $\boldsymbol{h}^{k_{\text{end}}}_{\text{utt}}$ concatenated, where k_{start} and k_{end} denote the span start and end indices.

2.6 Decoder: Search

Similar to other sequence-to-sequence models, we employ *beam search* over the sequence of function invocations when decoding. However, in contrast to other models, our assumptions (§2.3) allow us to

Dataset	SMCalFlow		TreeDST
	v1.1	v2.0	
Best Reported Result	66.5	68.2	62.2
Our Model	**73.8**	**75.3**	**72.8**

Table 1: Test set exact match accuracy comparing our model to the best reported results for SMCalFlow (Seq2Seq model from the public leaderboard; Semantic Machines et al., 2020) and TreeDST (TED-PP model; Cheng et al., 2020). The evaluation on each dataset in prior work requires us to repeat some idiosyncrasies that we describe in Appendix D.

efficiently implement beam search over *complete* function invocations, by leveraging the fact that:

$$\max_{\pi_i} p(\pi_i) = \max_{f_i} \left\{ p(f_i) \prod_{j=1}^{A_i} \max_{v_{ij}} p(v_{ij} \mid f_i) \right\}, \quad (9)$$

where we have omitted the dependence on $f_{<i}$. This computation is parallelizable and it also allows the decoder to avoid choosing a function if there are no high scoring assignments for its arguments (i.e., we are performing a kind of *lookahead*). This also means that the paths explored during the search are shorter for our model than for models where each step corresponds to a single decision, allowing for smaller beams and more efficient decoding.

3 Experiments

We first report results on SMCalFlow (Semantic Machines et al., 2020) and TreeDST (Cheng et al., 2020), two recently released large-scale conversational semantic parsing datasets. Our model makes use of type information in the programs, so we manually constructed a set of type declarations for each dataset and then used a variant of the Hindley-Milner type inference algorithm (Damas and Milner, 1982) to annotate programs with types. As mentioned in §2.1, we also transformed TreeDST to introduce meta-computation operators for references and revisions (more details can be found in Appendix C).[7] We also report results on non-conversational semantic parsing datasets in §3.2. We use the same hyperparameters across all experiments (see Appendix E), and we use BERT-medium (Turc et al., 2019) to initialize our encoder.

3.1 Conversational Semantic Parsing

Test set results for SMCalFlow and TreeDST are shown in Table 1. Our model significantly outperforms the best published numbers in each case.

[7] The transformed datasets are available at https://github.com/microsoft/task_oriented_dialogue_as_dataflow_synthesis/tree/master/datasets.

Dataset		SMCalFlow			TreeDST		
# Training Dialogues		1k	10k	33k	1k	10k	19k
w/o BERT	Seq2Seq	36.8	69.8	74.5	28.2	47.9	50.3
	Seq2Tree	43.6	69.3	77.7	23.6	46.9	48.8
	Seq2Tree++	48.0	71.9	78.2	74.8	75.4	86.9
	Our Model	**53.8**	**73.2**	**78.5**	**78.6**	**87.6**	**88.5**
w/ BERT	Seq2Seq	44.6	64.1	67.8	28.6	40.2	47.2
	Seq2Tree	50.8	74.6	78.6	30.9	50.6	51.6
	Our Model	**63.2**	**77.2**	**80.4**	**81.2**	**87.1**	**88.3**

(a) Baseline comparison.

Dataset		SMCalFlow			TreeDST		
# Training Dialogues		1k	10k	33k	1k	10k	19k
Our Model		63.2	77.2	80.4	81.2	87.1	88.3
Parser	Value Dependence	60.6	76.4	79.4	79.3	86.2	86.5
	No Name Embedder	62.8	76.7	80.3	81.1	87.0	88.1
	No Types	62.4	76.5	79.9	80.6	87.1	88.3
	No Span Copy	60.2	76.2	79.8	79.0	86.7	87.4
	No Entity Proposers	59.6	76.4	79.8	80.5	86.9	88.2
	All of the Above	58.9	75.8	77.3	72.9	80.2	80.6
History	No History	59.0	70.0	73.8	68.3	75.0	76.5
	Previous Turn	61.3	75.9	77.4	80.5	86.9	87.4
	Linear Encoder	63.0	76.5	80.2	81.2	87.1	88.3

(b) Ablation study.

Table 2: Validation set exact match accuracy across varying amounts of training data (each subset is sampled uniformly at random). The best results in each case are shown in **bold red and are underlined**.

In order to further understand the performance characteristics of our model and quantify the impact of each modeling contribution, we also compare to a variety of other models and ablated versions of our model. We implemented the following baselines:

- Seq2Seq: The OpenNMT (Klein et al., 2017) implementation of a pointer-generator network (See et al., 2017) that predicts linearized plans represented as S-expressions and is able to copy tokens from the utterance while decoding. This model is very similar to the model used by Semantic Machines et al. (2020) and represents the current state-of-the-art for SMCalFlow.[8]
- Seq2Tree: The same as Seq2Seq, except that it generates invocations in a top-down, pre-order program traversal. Each invocation is embedded as a unique item in the output vocabulary. Note that SMCalFlow contains re-entrant programs represented with LISP-style `let` bindings. Both the Seq2Tree and Seq2Seq are unaware of the special meaning of `let` and predict calls to `let` as any other function, and references to bound

[8] Semantic Machines et al. (2020) used linearized plans to represent the dialogue history, but our implementation uses previous user and agent utterances. We found no difference in performance.

variables as any other literal.

– Seq2Tree++: An enhanced version of the model by Krishnamurthy et al. (2017) that predicts typed programs in a top-down fashion. Unlike Seq2Seq and Seq2Tree, this model can only produce well-formed and well-typed programs. It also makes use of the same entity proposers (§2.1) similar to our model, and it can atomically copy spans of up to 15 tokens by treating them as additional proposed entities. Furthermore, it uses the linear history encoder that is described in the next paragraph. Like our model, re-entrancies are represented as references to previous outputs in the predicted sequence.

We also implemented variants of Seq2Seq and Seq2Tree that use BERT-base[9] (Devlin et al., 2019) as the encoder. Our results are shown in Table 2a. Our model outperforms all baselines on both datasets, showing particularly large gains in the low data regime, even when using BERT. Finally, we implemented the following ablations, with more details provided in Appendix G:

– Value Dependence: Introduces a unique function for each value in the training data (except for copies) and transforms the data so that values are always produced by calls to these functions, allowing the model to condition on them.
– No Name Embedder: Embeds functions and constants atomically instead of using the approach of §2.4 and the utterance encoder.
– No Types: Collapses all types to a single type, which effectively disables type checking (§2.5).
– No Span Copy: Breaks up span-level copies into token-level copies which are put together using a special `concatenate` function. Note that our model is *value-agnostic* and so this ablated model cannot condition on previously copied tokens when copying a span token-by-token.
– No Entity Proposers: Removes the entity proposers, meaning that previously entity-linked values have to be generated as constants.
– No History: Sets $\boldsymbol{H}_{\text{enc}} = \boldsymbol{H}_{\text{utt}}$ (§2.2).
– Previous Turn: Replaces the type-based history encoding with the previous turn user and system utterances or linearized system actions.
– Linear Encoder: Replaces the history attention mechanism with a linear function over a multi-hot embedding of the history types.

[9]We found that BERT-base worked best for these baselines, but was no better than the smaller BERT-medium when used with our model. Also, unfortunately, incorporating BERT in Seq2Tree++ turned out to be challenging due to the way that model was originally implemented.

Method		Dataset		
		JOBS	GEO	ATIS
Zettlemoyer and Collins (2007)		—	86.1	84.6
Wang et al. (2014)		90.7	90.4	**91.3**
Zhao and Huang (2015)		85.0	88.9	84.2
Saparov et al. (2017)		81.4	83.9	—
Neural Methods	Dong and Lapata (2016)	90.0	87.1	84.6
	Rabinovich et al. (2017)	**92.9**	87.1	85.9
	Yin and Neubig (2018)	—	88.2	86.2
	Dong and Lapata (2018)	—	88.2	87.7
	Aghajanyan et al. (2020)	—	89.3	—
	Our Model	91.4	**91.4**	90.2
	↳ No BERT	91.4	90.0	**91.3**

Table 3: Validation set exact match accuracy for single-turn semantic parsing datasets. Note that Aghajanyan et al. (2020) use BART (Lewis et al., 2020), a large pre-trained encoder. The best results for each dataset are shown in **bold red and are underlined**.

The results, shown in Table 2b, indicate that all of our features play a role in improving accuracy. Perhaps most importantly though, the "value dependence" ablation shows that our function-based program representations are indeed important, and the "previous turn" ablation shows that our type-based program representations are also important. Furthermore, the impact of both these modeling decisions grows larger in the low data regime, as does the impact of the span copy mechanism.

3.2 Non-Conversational Semantic Parsing

Our main focus is on conversational semantic parsing, but we also ran experiments on non-conversational semantic parsing benchmarks to show that our model is a strong parser irrespective of context. Specifically, we manually annotated the JOBS, GEOQUERY, and ATIS datasets with typed declarations (Appendix C) and ran experiments comparing with multiple baseline and state-of-the-art methods. The results, shown in Table 3, indicate that our model meets or exceeds state-of-the-art performance in each case.

4 Related Work

Our approach builds on top of a significant amount of prior work in neural semantic parsing and also context-dependent semantic parsing.

Neural Semantic Parsing. While there was a brief period of interest in using unstructured sequence models for semantic parsing (e.g., Andreas

et al., 2013; Dong and Lapata, 2016), most research on semantic parsing has used tree- or graph-shaped decoders that exploit program structure. Most such approaches use this structure as a constraint while decoding, filling in function arguments one-at-a-time, in either a top-down fashion (e.g., Dong and Lapata, 2016; Krishnamurthy et al., 2017) or a bottom-up fashion (e.g., Misra and Artzi, 2016; Cheng et al., 2018). Both directions can suffer from exposure bias and search errors during decoding: in top-down when there's no way to realize an argument of a given type in the current context, and in bottom-up when there are no functions in the programming language that combine the predicted arguments. To this end, there has been some work on global search with guarantees for neural semantic parsers (e.g., Lee et al., 2016) but it is expensive and makes certain strong assumptions. In contrast to this prior work, we use program structure not just as a decoder constraint but as a source of independence assumptions: the decoder explicitly decouples some decisions from others, resulting in good inductive biases and fast decoding algorithms.

Perhaps closest to our work is that of Dong and Lapata (2018), which is also about decoupling decisions, but uses a dataset-specific notion of an abstracted program *sketch* along with different independence assumptions, and underperforms our model in comparable settings (§3.2). Also close are the models of Cheng et al. (2020) and Zhang et al. (2019). Our method differs in that our beam search uses larger steps that predict functions together with their arguments, rather than predicting the argument values serially in separate dependent steps. Similar to Zhang et al. (2019), we use a *target-side* copy mechanism for generating references to function invocation results. However, we extend this mechanism to also predict constants, copy spans from the user utterance, and link externally proposed entities. While our span copy mechanism is novel, it is inspired by prior attempts to copy spans instead of tokens (e.g., Singh et al., 2020). Finally, bottom-up models with similarities to ours include SMBOP (Rubin and Berant, 2020) and BUSTLE (Odena et al., 2020).

Context-Dependent Semantic Parsing. Prior work on conversational semantic parsing mainly focuses on the decoder, with few efforts on incorporating the dialogue history information in the encoder. Recent work on context-dependent semantic parsing (e.g., Suhr et al., 2018; Yu et al., 2019) conditions on explicit representations of user utterances and programs with a neural encoder. While this results in highly expressive models, it also increases the risk of overfitting. Contrary to this, Zettlemoyer and Collins (2009), Lee et al. (2014) and Semantic Machines et al. (2020) do not use context to resolve references at all. They instead predict context-independent logical forms that are resolved in a separate step. Our approach occupies a middle ground: when combined with local program representations, types, even without any value information, provide enough information to resolve context-dependent meanings that cannot be derived from isolated sentences. The specific mechanism we use to do this "infuses" contextual type information into input sentence representations, in a manner reminiscent of *attention flow* models from the QA literature (e.g., Seo et al., 2016).

5 Conclusion

We showed that abstracting away values while encoding the dialogue history and decoding programs significantly improves conversational semantic parsing accuracy. In summary, our goal in this work is to think about types in a new way. Similar to previous neural and non-neural methods, types are an important source of constraints on the behavior of the decoder. Here, for the first time, they are also the primary ingredient in the *representation* of both the parser actions and the dialogue history.

Our approach, which is based on type-centric encodings of dialogue states and function-centric encodings of programs (§2), outperforms prior work by 7.3% and 10.6%, on SMCALFLOW and TREEDST, respectively (§3), while also being more computationally efficient than competing methods. Perhaps more importantly, it results in even more significant gains in the low-data regime. This indicates that choosing our representations carefully and making appropriate independence assumptions can result in increased accuracy and computational efficiency.

6 Acknowledgements

We thank the anonymous reviewers for their helpful comments, Jason Eisner for his detailed feedback and suggestions on an early draft of the paper, Abulhair Saparov for helpful conversations and pointers about semantic parsing baselines and prior work, and Theo Lanman for his help in scaling up some of our experiments.

References

Armen Aghajanyan, Jean Maillard, Akshat Shrivastava, Keith Diedrick, Michael Haeger, Haoran Li, Yashar Mehdad, Veselin Stoyanov, Anuj Kumar, Mike Lewis, and Sonal Gupta. 2020. Conversational Semantic Parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5026–5035. Association for Computational Linguistics.

Jacob Andreas, Andreas Vlachos, and Stephen Clark. 2013. Semantic Parsing as Machine Translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 47–52, Sofia, Bulgaria. Association for Computational Linguistics.

Marco Baroni. 2020. Linguistic Generalization and Compositionality in Modern Artificial Neural Networks. *Philosophical Transactions of the Royal Society B*, 375(1791):20190307.

Jianpeng Cheng, Devang Agrawal, Héctor Martínez Alonso, Shruti Bhargava, Joris Driesen, Federico Flego, Dain Kaplan, Dimitri Kartsaklis, Lin Li, Dhivya Piraviperumal, Jason D. Williams, Hong Yu, Diarmuid Ó Séaghdha, and Anders Johannsen. 2020. Conversational Semantic Parsing for Dialog State Tracking. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8107–8117. Association for Computational Linguistics.

Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. 2018. Learning an Executable Neural Semantic Parser. *Computational Linguistics*, 45(1):59–94. Publisher: MIT Press.

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA. Association for Computing Machinery.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43, Berlin, Germany. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2018. Coarse-to-Fine Decoding for Neural Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742, Melbourne, Australia. Association for Computational Linguistics.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany. Association for Computational Linguistics.

Kelvin Guu, Panupong Pasupat, E. Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. *ArXiv*, abs/1704.07926.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12–22, Berlin, Germany. Association for Computational Linguistics.

Bevan Jones, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, and Kevin Knight. 2012. Semantics-Based Machine Translation with Hyperedge Replacement Grammars. In *Proceedings of COLING 2012*, pages 1359–1376, Mumbai, India. The COLING 2012 Organizing Committee.

Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs.LG]*. ArXiv: 1412.6980.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada. Association for Computational Linguistics.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural Semantic Parsing with Type Constraints for Semi-Structured Tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark. Association for Computational Linguistics.

Tatsuki Kuribayashi, Hiroki Ouchi, Naoya Inoue, Paul Reisert, Toshinori Miyoshi, Jun Suzuki, and Kentaro Inui. 2019. An Empirical Study of Span Representations in Argumentation Structure Parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4691–4698, Florence, Italy. Association for Computational Linguistics.

Kenton Lee, Yoav Artzi, Jesse Dodge, and Luke Zettlemoyer. 2014. Context-dependent Semantic Parsing for Time Expressions. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages

1437–1447, Baltimore, Maryland. Association for Computational Linguistics.

Kenton Lee, Mike Lewis, and Luke Zettlemoyer. 2016. Global Neural CCG Parsing with Optimality Guarantees. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2366–2376, Austin, Texas. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880. Association for Computational Linguistics.

Dipendra Kumar Misra and Yoav Artzi. 2016. Neural Shift-Reduce CCG Semantic Parsing. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1775–1786, Austin, Texas. Association for Computational Linguistics.

Feng Nie, Jin-Ge Yao, Jinpeng Wang, Rong Pan, and Chin-Yew Lin. 2019. A Simple Recipe towards Reducing Hallucination in Neural Surface Realisation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2673–2679, Florence, Italy. Association for Computational Linguistics.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, and Charles Sutton. 2020. BUSTLE: Bottom-up Program Synthesis Through Learning-guided Exploration. *arXiv:2007.14381 [cs, stat]*. ArXiv: 2007.14381.

Panupong Pasupat and Percy Liang. 2015. Compositional Semantic Parsing on Semi-Structured Tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1470–1480, Beijing, China. Association for Computational Linguistics.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Ohad Rubin and Jonathan Berant. 2020. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing. *arXiv:2010.12412 [cs]*. ArXiv: 2010.12412.

Abulhair Saparov, Vijay Saraswat, and Tom Mitchell. 2017. Probabilistic Generative Grammar for Semantic Parsing. In *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017)*, pages 248–259, Vancouver, Canada. Association for Computational Linguistics.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada. Association for Computational Linguistics.

Semantic Machines, Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-Oriented Dialogue as Dataflow Synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. 2016. Bidirectional Attention Flow for Machine Comprehension. *arXiv:1611.01603 [cs.CL]*. ArXiv: 1611.01603.

Abhinav Singh, Patrick Xia, Guanghui Qin, Mahsa Yarmohammadi, and Benjamin Van Durme. 2020. CopyNext: Explicit Span Copying and Alignment in Sequence to Sequence Models. In *Proceedings of the Fourth Workshop on Structured Prediction for NLP*, pages 11–16. Association for Computational Linguistics.

Mitchell Stern, Jacob Andreas, and Dan Klein. 2017. A Minimal Span-Based Neural Constituency Parser. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 818–827, Vancouver, Canada. Association for Computational Linguistics.

Alane Suhr, Srinivasan Iyer, and Yoav Artzi. 2018. Learning to Map Context-Dependent Sentences to Executable Formal Queries. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2238–2249, New Orleans, Louisiana. Association for Computational Linguistics.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-Read Students Learn Better: On the Importance of Pre-training Compact Models. *arXiv:1908.08962 [cs.CL]*. ArXiv: 1908.08962.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All

you Need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc.

Adrienne Wang, Tom Kwiatkowski, and Luke Zettlemoyer. 2014. Morpho-syntactic Lexical Generalization for CCG Semantic Parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1284–1295, Doha, Qatar. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. SParC: Cross-Domain Semantic Parsing in Context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4511–4523, Florence, Italy. Association for Computational Linguistics.

Luke Zettlemoyer and Michael Collins. 2007. Online Learning of Relaxed CCG Grammars for Parsing to Logical Form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 678–687, Prague, Czech Republic. Association for Computational Linguistics.

Luke Zettlemoyer and Michael Collins. 2009. Learning Context-Dependent Mappings from Sentences to Logical Form. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 976–984, Suntec, Singapore. Association for Computational Linguistics.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019. AMR Parsing as Sequence-to-Graph Transduction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 80–94, Florence, Italy. Association for Computational Linguistics.

Kai Zhao and Liang Huang. 2015. Type-Driven Incremental Semantic Parsing with Polymorphism. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1416–1421, Denver, Colorado. Association for Computational Linguistics.

A Invocation Joint Normalization

Instead of the distribution in Equation 3, we can define a distribution over f_i and $\{v_{ij}\}_{j=1}^{A_i}$ that factorizes in the same way but is also jointly normalized:

$$p(\pi_i \mid f_{<i}) \propto h(f_i) \prod_{j=1}^{A_i} g(f_i, v_{ij}), \qquad (10)$$

where h and g are defined as presented in §2.4 and §2.5, respectively, before normalization. This model has the same cost as the locally normalized model at test time but is significantly more expensive at training time as we need to score all possible function invocations, as opposed to always conditioning on the gold functions. It can in principle avoid some of the exposure bias problems of the locally normalized model, but we observed no accuracy improvements in our experiments.

B Value Sources

In our model, the type of a value determines what sources it can be generated from. We enforce that values of certain types can only be copied or entity-linked. Any values that do not fall under these constraints are added to a static vocabulary of constants, and the model is always permitted to generate them, as long as they pass type checking. Values that fall under these constraints are not added to this vocabulary so that they cannot be "hallucinated" by the model. The specific constraints that we use are described in the following paragraphs.

Types that must be copied: Types for which the model is only allowed to construct values directly from string literals copied from the utterance. In §2.5 we noted that strings can be copied from the utterance to become string literals in the generated program. For certain types t, arguments of type t may also be willing to accept copied strings; in this case we generate a constructor call that constructs a t object from the string literal. For SMCALFLOW, these *copyable types* are `String`, `PersonName`, `RespondComment`, and `LocationKeyphrase`. For the other datasets it is just `String`. We declare training examples where a value of a copyable type appears in the program, but is not a substring of the corresponding utterance, as likely annotation errors and ignore them during training (but not during evaluation). Even though such examples are very rare for SMCALFLOW (∼0.5% of the examples), they turned out to be relatively frequent in TREEDST (∼6% of the examples), as we discuss in Appendix C.

Types that must be entity-linked: Types for which argument values can only be picked from the set of proposed entities (§2.1) and cannot be otherwise hallucinated from the model, or directly copied from the utterance. The `Number` type is treated in a special way for all datasets, where numbers 0, 1, and 2 are allowed to be hallucinated, but all other numbers must be entity-linked. Furthermore, for SMCALFLOW the set of types that must be entity-linked also contains the `Month`, `DayOfWeek`, and `Holiday` types. Based on this, we can detect probable annotation errors.

C Dataset Preparation

We now describe how we processed the datasets to satisfy the requirements mentioned in §2.1. We have made the processed datasets available at https://github.com/microsoft/task_oriented_dialogue_as_dataflow_synthesis/tree/master/datasets.

C.1 Type Declarations

We manually specified the necessary type declarations by inspection of all functions in the training data. In some cases, we found it helpful to transform the data into an equivalent set of function calls that simplified the resulting programs, while maintaining a one-to-one mapping with the original representations. For example, SMCALFLOW contains a function called `get` that takes in an object of some type and a `Path`, which specifies a field of that object, and acts as an accessor. For example, the object could be an `Event` and the specified path may be `"subject"`. We transform such invocations into invocations of functions that are instantiated separately for each unique combination of the object type and the provided path. For the aforementioned example, the corresponding new function would be defined as:

```
def Event.subject(obj: Event): String
```

All such transformations are invertible, so we can convert back to the original format after prediction.

C.2 Meta-Computation Operators

The meta-computation operators are only required for the conversational semantic parsing datasets, and SMCALFLOW already makes use of them. Therefore, we only had to convert TREEDST. To this end, we introduced two new operators:

```
def refer[T](): T

def revise[T, R](
  root: Root[T],
  path: Path[R],
  revision: R => R,
): T
```

`refer` goes through the programs and system actions in the dialogue history, starting at the most recent turn, finds the first sub-program that evaluates to type `T`, and replaces its invocation with that sub-program. Similarly, `revise` finds the first program whose root matches the specified `root`, walks down the tree along the specified `path`, and applies the provided `revision` on the sub-program rooted at the end of that path. It then replaces its invocation with this revised program. We performed an automated heuristic transformation of TREEDST so that it makes use of these meta-operators. We only applied the extracted transformations when executing them on the transformed programs using the gold dialogue history resulted in the original program (i.e., before applying any of our transformations). Therefore, when using the gold dialogue history, this transformation is also guaranteed to be invertible. We emphasize that we execute these meta-computation operators before computing accuracy so that our final evaluation results are comparable to prior work.

C.3 Annotation Errors

While preparing the datasets for our experiments using our automated transformations, we noticed that they contain some inconsistencies. For example, in TREEDST, the tree fragment:

```
...restaurant.book.restaurant.book...
```

seemed to be interchangeable with:

```
...restaurant.object.equals...
```

The annotation and checking mechanisms we employ impose certain regularity requirements on the data that are violated by such examples. Therefore, we had three choices for such examples: (i) we could add additional type declarations, (ii) we could discard them, or (iii) we could collapse the two annotations together, resulting in a lossy conversion. We used our best judgment when choosing among these options, preferring option (iii) where it was possible to do so automatically. We believe that all such cases are annotation errors, but we cannot know for certain without more information about how the TREEDST dataset was constructed. Overall, about 122 dialogues (0.4%) did not pass our checks for SMCALFLOW, and 585 dialogues (3.0%) for TREEDST. When converting back to the original format, we tally an error for each discarded example, and select the most frequent version of any lossily collapsed annotation.

Our approach also provides two simple yet effective consistency checks for the training data: (i) running type inference using the provided type declarations to detect ill-typed examples, and (ii) using the constraints described Appendix B to detect other forms of annotation errors. We found that these two checks together caught 68 potential annotation errors (<0.5%) in SMCALFLOW and ∼1,000 potential errors (∼6%) in TREEDST. TREEDST was particularly interesting as we found a whole class of examples where user utterances were replaced with system utterances.

Note that our model does not technically require any of these checks. It is possible to generate type signatures that permit arbitrary function/argument pairs based on observed data and to configure our model so that any observed value may be generated as a constant (i.e., not imposing the constraints described in Appendix B). In practice we found that constraining the space of programs provides useful sanity checks in addition to accuracy gains.

C.4 Non-Conversational Semantic Parsing

We obtained the JOBS, GEOQUERY, and ATIS datasets from the repository of Dong and Lapata (2016). For each dataset, we defined a *library* that specifies function and type declarations.

D Evaluation Details

To compare with prior work for SMCALFLOW (Semantic Machines et al., 2020) and TREEDST (Cheng et al., 2020), we replicated their setups. For SMCALFLOW, we predict plans always conditioning on the gold dialogue history for each utterance, but we consider any predicted plan wrong if the `refer_are_correct` flag is set to `false`. This flag is meant to summarize the accuracy of a hypothetical model for resolving calls to `refer`, but is not relevant to the problem of program prediction. We also canonicalize plans by sorting keyword arguments and normalizing numbers (so that 30.0 and 30 are considered equivalent, for example). For TREEDST, our model predicts programs that use the `refer` and `revise` operators, and we execute them against the dialogue history that consists of predicted programs and gold (oracle) system

actions (following Cheng et al. (2020)) when converting back to the original tree representation. We canonicalize the resulting trees by lexicographically sorting the children of each node.

For our baseline comparisons and ablations (shown in Tables 2a and 2b), we decided to ignore the `refer_are_correct` flag for SMCALFLOW because it assumes that `refer` is handled by some other model and for these experiments we are only interested in evaluating program prediction. Also, for TREEDST we use the gold plans for the dialogue history in order to focus on the semantic parsing problem, as opposed to the dialogue state tracking problem. For the non-conversational semantic parsing datasets we replicated the evaluation approach of Dong and Lapata (2016), and so we also canonicalize the predicted programs.

E Model Hyperparameters

We use the same hyperparameters for all of our conversational semantic parsing experiments. For the encoder, we use either BERT-medium (Turc et al., 2019) or a non-pretrained 2-layer Transformer (Vaswani et al., 2017) with a hidden size of 128, 4 heads, and a fully connected layer size of 512, for the non-BERT experiments. For the decoder we use a 2-layer Transformer with a hidden size of 128, 4 heads, and a fully connected layer size of 512, and set h_{type} to 128, and h_{arg} to 512. For the non-conversational semantic parsing experiments we use a hidden size of 32 throughout the model as the corresponding datasets are very small. We also use a dropout of 0.2 for all experiments.

For training, we use the Adam optimizer (Kingma and Ba, 2017), performing global gradient norm clipping with the maximum allowed norm set to 10. For batching, we *bucket* the training examples by utterance length and adapt the batch size so that the total number of tokens in each batch is 10,240. Finally, we average the log-likelihood function over each batch, instead of summing it.

Experiments with BERT. We use a pre-training phase for 2,000 training steps, where we freeze the parameters of the utterance encoder and only train the dialogue history encoder and the decoder. Then, we train the whole model for another 8,000 steps. This because our model is not simply adding a linear layer on top of BERT, and so, unless initialized properly, we may end up losing some of the information contained in the pre-trained BERT model. During the pre-training phase, we linearly warm up the learning rate to 2×10^{-3} during the first 1,000 steps. We then decay it exponentially by a factor of 0.999 every 10 steps. During the full training phase, we linearly warm up the learning rate to 1×10^{-4} during the first 1,000 steps, and then decay it exponentially in the same fashion.

Experiments without BERT. We use a single training phase for 30,000 steps, where we linearly warm up the learning rate to 5×10^{-3} during the first 1,000 steps, and then we decay it exponentially by a factor of 0.999 every 10 steps. We need a larger number of training steps in this case because none of the model components have been pre-trained. Also, the encoder is now much smaller, meaning that we can afford a higher learning rate.

Even though these hyperparameters may seem very specific, we emphasize that our model is robust to the choice of hyperparameters and this setup was chosen once and shared across all experiments.

F Baseline Models

Seq2Seq. This model predicts linearized, tokenized S-expressions using the OpenNMT implementation of a Transformer-based (Vaswani et al., 2017) pointer-generator network (See et al., 2017). For example, the following program:

```
+(length("some string"), 1)
```

would correspond to the space-separated sequence:

```
( + ( length " some string " ) 1 )
```

In contrast to the model proposed in this paper, in this case tokens that belong to functions and values (i.e., that are outside of quotes) can also be copied directly from the utterance. Furthermore, there is no guarantee that this baseline will produce a well-formed program.

Seq2Tree. This model uses the same underlying implementation as our Seq2Seq baseline—also with no guarantee that it will produce a well-formed program—but it predicts a different sequence. For example, the following program:

```
+(+(1, 2), 3)
```

would be predicted as the sequence:

```
+(<NT>, 3)
+(1, 2)
```

Each item in the sequence receives a unique embedding in the output vocabulary and so, `"+(1,2)"` and `"+(<NT>, 3)"` share no parameters. `<NT>` is

a special placeholder symbol that represents a substitution point when converting the linearized sequence back to a tree. Furthermore, copies are not inlined into invocations, but broken out into token sequences. For example, the following program:

```
+(length("some string"), 1)
```

would be predicted as the sequence:

```
+(<NT>, 1)
length(<NT>)
"
some
string
"
```

Seq2Tree++. This is a re-implementation of Krishnamurthy et al. (2017) with some differences: (i) our implementation's entity linking embeddings are computed over spans, including type information (as in the original paper) and a span embedding computed based on the LSTM hidden state at the start and end of each entity span, (ii) copies are treated as entities by proposing all spans up to length 15, and (iii) we use the linear dialogue history encoder described in §3.1.

G Ablations

The "value dependence" and the "no span copy" ablations are perhaps the most important in our experiments, and so we provide some more details about them in the following paragraphs.

Value Dependence. The goal of this ablation is to quantify the impact of the dependency structure we propose in Equation 3. To this end, we first convert all functions to a *curried* form, where each argument is provided as part of a separate function invocation. For example, the following invocation:

```
[0] event(subject = s0, start = t0, end = t1)
```

is transformed to the following program fragment:

```
[0] value(s0)
[1] event_0(subject = [0])
[2] value(t0)
[3] event_1(curried = [1], start = [2])
[4] value(t1)
[5] event_2(curried = [3], end = [4])
```

When choosing a function, our decoder does not condition on the argument values of the previous invocations. In order to enable such conditioning without modifying the model implementation, we also transform the `value` function invocations whose underlying values are not copies, such that there exists a unique function for each unique value. This results in the following program:

```
[0] value_s0(s0)
[1] event_0(subject = [0])
[2] value_t0(t0)
[3] event_1(curried = [1], start = [2])
[4] value_t1(t1)
[5] event_2(curried = [3], end = [4])
```

Note that we keep the `value_s0`, `value_t0`, and `value_t1` function arguments because they allow the model to marginalize over multiple possible value sources (§2.5). The reason we do not transform the `value` functions that correspond to copies is because we attempted doing that on top of the span copy ablation, but it performed poorly and we decided that it may be a misrepresentation. Overall, this ablation offers us a way to obtain a bottom-up parser that maintains most properties of the proposed model, except for its dependency structure.

No Span Copy. In order to ablate the proposed span copy mechanism we implemented a data transformation that replaces all copied values with references to the result of a `copy` function (for spans of length 1) or the result of a `concatenate` function called on the results of 2 or more calls to `copy`. For example, the function invocation:

```
[0] event(subject = "water the plant")
```

is converted to:

```
[0] copy("water")
[1] copy("the")
[2] concatenate([0], [1])
[3] copy("plant")
[4] concatenate([2], [3])
[5] event(subject = [4])
```

When applied on its own and not combined with other ablation, the single token copies are further inlined to produce the following program:

```
[0] concatenate("water", "the")
[1] concatenate([0], "plant")
[2] event(subject = [1])
```

H Computational Efficiency

For comparing model performance we computed the average utterance processing time across all of the SMCalFlow validation set, using a single Nvidia V100 GPU. The fastest baseline required about 80ms per utterance, while our model only required about 8ms per utterance. This can be attributed to multiple reasons, such as the facts that: (i) our independence assumptions allow us to predict the argument value distributions in parallel, (ii) we avoid enumerating all possible utterance spans when computing the normalizing constant for the argument values, and (iii) we use ragged tensors to avoid unnecessary padding and computation.