Task-Oriented Dialogue as Dataflow Synthesis

1 Introduction

Two central design decisions in modern conversational AI systems are the choices of state and action representations, which determine the scope of possible user requests and agent behaviors. Dialogue systems with fixed symbolic state representations (like slot filling systems) are easy to train but hard to extend (Pieraccini et al., 1992). Deep continuous state representations are flexible enough to represent arbitrary properties of the dialogue history, but so unconstrained that training a
neural dialogue policy “end-to-end” fails to learn appropriate latent states (Bordes et al., 2016).

This paper introduces a new framework for dialogue modeling that aims to combine the strengths of both approaches: structured enough to enable efficient learning, yet flexible enough to support open-ended, compositional user goals that involve multiple tasks and domains. The framework has two components: a new state representation in which dialogue states are represented as dataflow graphs; and a new agent architecture in which dialogue agents predict compositional programs that extend these graphs. Over the course of a dialogue, a growing dataflow graph serves as a record of common ground: an executable description of the entities that were mentioned and the actions and computations that produced them (Figure 1).

While this paper mostly focuses on representational questions, learning is a central motivation for our approach. Learning to interpret natural-language requests is simpler when they are understood to specify graph-building operations. Human speakers avoid repeating themselves in conversation by using anaphora, ellipsis, and bridging to build on shared context (Mitkov, 2014). Our framework treats these constructions by translating them into explicit metacomputation operators for reference and revision, which directly retrieve fragments of the dataflow graph that represents the shared dialogue state. This approach borrows from corresponding ideas in the literature on program transformation (Visser, 2001) and results in compact, predictable programs whose structure closely mirrors user utterances.

Experiments show that our rich dialogue state representation makes it possible to build better dialogue agents for challenging tasks. First, we release a newly collected dataset of around 40K natural dialogues in English about calendars, locations, people, and weather—the largest goal-oriented dialogue dataset to date. Each dialogue turn is annotated with a program implementing the user request. Many turns involve more challenging predictions than traditional slot-filling, with compositional actions, cross-domain interaction, complex anaphora, and exception handling (Figure 2). On this dataset, explicit reference mechanisms reduce the error rate of a seq2seq-with-copying model (See et al., 2017) by 5.9% on all turns and by 10.9% on turns with a cross-turn reference. To demonstrate breadth of applicability, we additionally describe how to automatically convert the simpler MultiWOZ dataset into a dataflow representation. This representation again enables a basic seq2seq model to outperform a state-of-the-art, task-specific model at traditional state tracking. Our results show that within the dataflow framework, a broad range of agent behaviors are both representable and learnable, and that explicit abstractions for reference and revision are the keys to effective modeling.

2 Overview: Dialogue and Dataflow

This section provides a high-level overview of our dialogue modeling framework, introducing the main components of the approach. Sections 3–5 refine this picture, describing the implementation and use of specific metacomputation operators.

We model a dialogue between a (human) user and an (automated) agent as an interactive programming task where the human and computer communicate using natural language. Dialogue state is represented with a dataflow graph. At each turn, the agent’s goal is to translate the most recent user utterance into a program. Predicted programs nondestructively extend the dataflow graph, construct any newly requested values or real-world side-effects, and finally describe the results to the user. Our approach is significantly different from a conventional dialogue system pipeline, which has separate modules for language understanding, dialogue state tracking, and dialogue policy execution (Young et al., 2013). Instead, a single learned model directly predicts executable agent actions and logs them in a graphical dialogue state.

Programs, graphs, and evaluation The simplest example of interactive program synthesis is question answering:

<table>
<thead>
<tr>
<th>User:</th>
<th>When is the next retreat?</th>
</tr>
</thead>
<tbody>
<tr>
<td>start(findEvent(EventSpec(</td>
<td></td>
</tr>
<tr>
<td>name='retreat',</td>
<td></td>
</tr>
<tr>
<td>start=after(now()))))</td>
<td></td>
</tr>
<tr>
<td>Agent:</td>
<td>It starts on April 27 at 9 am.</td>
</tr>
</tbody>
</table>

Here the agent predicts a program that invokes an API call (findEvent) on a structured input (EventSpec) to produce the desired query.¹ This

¹Note that what the agent predicts is not a formal representation of the utterance’s meaning, but a query that enables a contextually appropriate response (what Austin (1962) called the “perlocutionary force” of the utterance on
is a form of semantic parsing (Zelle, 1995).

The program predicted above can be rendered as a dataflow graph:

```
  'retreat'
    name
  EventSpec
    findEvent
      start
        after
        now

  DateTimeSpec(...)
```

Each function call in the program corresponds to a node labeled with that function. This node’s parents correspond to the arguments of the function call. The top-level call that returns the program’s result is depicted with a solid border. A dataflow graph is always acyclic, but is not necessarily a tree, as nodes may be reused.

Once nodes are added to a dataflow graph, they are evaluated in topological order. Evaluating a node applies its function to its parents’ values:

```
  'retreat'
    name
  EventSpec
    findEvent
      start
        after
        now
          DateTimeSpec(...)
```

Here we have annotated two nodes to show that the value of `after(now())` is a `DateTimeSpec` and the value of the returned `start` node is a specific `DateTime`. Evaluated nodes are shaded in our diagrams. Exceptions (see §5) block evaluation, leaving downstream nodes unevaluated.

The above diagram saves space by summarizing the (structured) value of a node as a string. In reality, each evaluated node has a dashed result edge that points to the result of evaluating it:

```
2020
   apr
      start
        DateTime
          dayOfWeek
            DateTimeSpec(...)
```

That result is itself a node in the dataflow graph—often a new node added by evaluation. It may have its own result edge. A node’s value is found by transitively following result edges until we arrive at a node whose result is itself. Such a terminal node is either a primitive value (e.g., 2020), or a constructor (e.g., `DateTime`) whose parent nodes’ values specify its arguments. A constructor has the same (capitalized) name as the type it constructs.

**Reference and revision** We now sketch two metacomputation functions whose evaluation extends the graph in complex ways.

As a representation of dialogue state, the dataflow graph records entities that have been mentioned and the relationships among them. All nodes in the dataflow graph are eligible to be referenced by subsequent utterances. Suppose, for example, that the user continues the previous dialogue fragment with a follow-up question:

```
User: What day of the week is that?
dayOfWeek(refer())
```

```
Agent: It’s a Monday.
```

The user’s word *that* becomes the `refer` call in our predicted program, as it is a reference to “some salient previously mentioned node.” Evaluating `refer` here chooses the top-level node, `start`, from the previous turn. That node is then used as an argument to a `dayOfWeek` node (nodes existing from previous turns are shown here in lighter ink):

```
  'retreat'
    name
  EventSpec
    findEvent
      start
        after
        now
          DateTimeSpec(...)
```

and evaluating the latter node applies `dayOfWeek` to `start`’s value. This diagram is actually a simplification: we will show in §3 how the `refer` call itself is also captured in the dataflow graph.

The user may next ask a question that changes the upstream constraint on the event’s start time:

```
User: What about in 2021?
```

```
revise(
  new=DateTimeSpec(year=2021),
  oldLoc=Constraint[DateTimeSpec],
  rootLoc=RoleConstraint(output))
```

† that invokes an API will generally return its value directly as a terminal node. However, translating natural language to higher-level function calls, which have been defined to expand into lower-level library calls (reminiscent of macro expansion), is often more easily learnable and more maintainable than translating it directly to the expanded graph.
A “new” DateTimeSpec (representing in 2021) is to be substituted for some salient existing old node that has value type DateTimeSpec (in this case, the node after(now())). The revise operator non-destructively splices in this new sub-computation and returns a revised version of the most salient computation containing old (in this case, the subgraph for the previous utterance, rooted at dayOfWeek):

```
<table>
<thead>
<tr>
<th>retreat</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventSpec</td>
<td>findEvent</td>
</tr>
<tr>
<td>DateTimeSpec</td>
<td></td>
</tr>
<tr>
<td>2021</td>
<td>dayOfWeek</td>
</tr>
</tbody>
</table>
```

As in the refer example, the target program (though not the above subgraph) corresponds closely to the user’s new utterance, making it easy to predict. Like the utterance itself, the program does not specify the revised subgraph in full, but describes how to find and reuse relevant structure from the previous dataflow graph.

Given a dataset of turns expressed in terms of appropriate graph-manipulation programs, the learning problem for a dataflow agent is the same as for any other supervised contextual semantic parser. We want to learn a function that maps user utterances to particular programs—a well-studied task for which standard models exist. Details of the model used for our experiments in this paper are provided in §7.

Aside: response generation This paper focuses on language understanding: mapping from a user’s natural language utterance to a formal response, in this case the value of the outlined node returned by a program. Dialogue systems must also perform language generation: mapping from this formal response to a natural-language response. The dataset released with this paper includes output from a learned generation model that can describe the value computed at a previous turn, describe the structure of the computation that produced the value, and reference other nodes in the dataflow graph via referring expressions. Support for structured, computation-conditional generation models is another advantage of dataflow-based dialogue state representations. While a complete description of dataflow-based language generation is beyond the scope of this paper, we briefly describe the components of the generation system relevant to the understanding system presented here.

The generation model is invoked after the evaluation phase. It conditions on a view of the graph rooted at the most recent return node, so generated responses can mention both the previously returned value and the computation that produced it. As the generation model produces the natural language response, it extends the dataflow graph. For example, if after the user query “What’s the date of the next retreat?” the agent responds:

```
| Agent: | It starts on April 27 at 9 am, and runs for 8 hours. |
```

then it will also extend the dataflow graph to reflect that the event’s duration was mentioned:

```
<table>
<thead>
<tr>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 hours</td>
</tr>
</tbody>
</table>
```

The duration of the event is now part of the common ground in the conversation and available for future reference by either the agent or the user.

The generation model is also important for agent initiative:

```
<table>
<thead>
<tr>
<th>User:</th>
<th>Put an event on my calendar.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent:</td>
<td>What should it be called?</td>
</tr>
</tbody>
</table>
```

As discussed in detail in §5, questions of this kind can be generated in response to exceptions generated by underspecified user requests. In the accompanying dataset release, the agent’s utterances are annotated with their dataflow graphs as extended by the generation model.

3 Reference resolution

In a dialogue, entities that have been introduced once may be referred to again. In dataflow dialogues, the entities available for reference are given by the nodes in the dataflow graph. Entities are salient to conversation participants to different degrees, and their relative salience determines the ways in which they may be referenced (Lappin and Leass, 1994). For example, it generally refers to the most salient non-human entity, while more specific expressions like the Friday meeting are needed to refer to accessible but less salient entities. Not all references to entities are overt: if the agent says “You have a meeting tomorrow” and the user responds “What time?”, the agent must predict the implicit reference to a salient event.
Dataflow pointers We have seen that refer is used to find referents for referring expressions. In general, these referents may be existing dataflow nodes or new subgraphs for newly mentioned entities. We now give more detail about both cases.

Imagine a dialogue in which the dataflow graph contains the following fragment (which translates a mention of Easter or answers When is Easter?):

Suppose the user subsequently mentions the day after that. We wish to produce this computation:

In our framework, this is accomplished by mapping the day after that to + (refer(), Days(1)). The corresponding graph is not quite the one shown above, but it evaluates to the same value:

This shows how the refer() call is reified as a node in the dataflow graph. Its result is the salient findDateTime node from the previous turn—whose own result, a specific DateTime, now serves as the value of refer. We show both result edges here. Evaluating + adds a day to this old DateTime value to get the result of +, a new DateTime.

To enable dataflow graph manipulation with referring expressions, all that is required is an implementation of refer that can produce appropriate pointers for both simple references (that) and complex ones (the first meeting).

Constraints A call to refer is essentially a query that retrieves a node from the dialogue history, using a salience model discussed below. refer takes an optional argument: a constraint on the returned node. Indeed, the proper translation of that in the context the day after that would be refer(Constraint[DateTime]). Constraints are predicates built from boolean connectives and the constructions illustrated below:

- **Type constraints:** the meeting maps to the call refer(Constraint[Event]()), where the constraint matches all nodes with values of type Event.
- **Property constraints:** Assuming a structured Event type with a date property, and a Date type with a weekday property, the Thursday meeting maps to this nested call: refer(Constraint[Event](date=Constraint[DateTime](weekday=thurs)))
- **Role constraints:** A role constraint specifies a keyword and matches nodes that are used as keyword arguments with that keyword. For example, the month maps to refer(RoleConstraint(month)) and resolves to the constant node apr in the dialogues in §2, since that node was used as a named argument month=apr. We further allow keyword paths that select arguments of arguments: thurs in the previous bullet would satisfy refer(RoleConstraint([date,weekday])).

To interpret a natural language referring expression, the program prediction model only needs to translate it into a contextually appropriate constraint C. refer(C) is then evaluated using a separate salience retrieval model that returns an appropriate node. The following dialogue shows referring expressions in action:

User: What’s happening this morning?
findEvent(EventSpec(
  start=and(today(), during(morning())))))
Agent: You have a planning meeting at 9 am.
User: What do I have after that?
findEvent(EventSpec(start=after(end(
  refer(Constraint[Event]())))))
Agent: Your birthday lunch is at 12:30 pm.
User: How many people are coming to the 9 am meeting?
length(attendees(
  refer(Constraint[Event](start=am(9))))))
Agent: 5 people will be at the planning meeting.

Fortunately, this constraint need not be manually annotated. Given the rest of the program, it can be inferred automatically by Hindley-Milner type inference (Hindley, 1969; Milner, 1978), which establishes that this refer node must return a DateTime if the program is to type-check.

---

ef{3} refers and revise are not the only operations that take constraints as arguments. For example, constraint arguments to findEvent and createEvent specify what sort of event is to be retrieved or created. In these other contexts, to avoid distracting the reader, this paper uses EventSpec as an alias for Constraint[Event]. It similarly uses aliases PersonSpec and DateTimeSpec. (Our dataset does not.)
Here the 9 am meeting refers to the one that is salient from the first response, not an arbitrary one.

**Salience retrieval model** The salience retrieval model returns the most salient node satisfying the underlying constraint. Our dataflow framework is agnostic to the implementation of this model. A sophisticated model could select nodes via a machine-learned scoring function. In the experiments in this paper, however, we rank nodes using a hard-coded heuristic.

The heuristic chooses the root node $r$ of the previous user utterance, if it satisfies the constraint. More generally, the heuristic prefers nodes to the extent that they can be reached from $r$ in a small number of steps, where a step may move from a node to one of its input nodes, from an evaluated node to its result node, or from the root of an utterance to the root of an adjacent (user or system) utterance. If no satisfying node is found in the past several utterances, the heuristic falls back to generating code (see footnote 2) that will search harder for a satisfying salient entity, perhaps by querying a database. For example, our earlier Constraint[Event](start=am(9)) may return the expression `findEvent(EventSpec(start=am(9)))` if no 9 am meeting has been mentioned recently, and Constraint[Person](name='Adam') may return `findPerson(PersonSpec(name='Adam'))` if no Adam has been mentioned. (see footnote 4.)

### 4 Revision

Beyond referring to previously mentioned entities (nodes), task-oriented dialogues frequently refer to previously executed computations (subgraphs). This is one of the major advantages of representing the dialogue state as a dataflow graph of computations, not just a set of potentially salient entities.

**User:** What time on Tuesday is my planning meeting?

**Agent:** You meet with Grace at noon.

**User:** Sorry, I meant all-hands.

**Agent:** Your all-hands meeting is at 2:30 pm.

The second user utterance asks for the computation from the first user utterance to be repeated, but with all-hands in place of planning. The expected result is still a time, even though the second utterance makes no mention of time.

In the dataflow framework, we invoke a `revise` operator to construct the revised computation:

```
User: Sorry, I meant all-hands.
revise(rootLoc=RoleConstraint(output),
oldLoc=Constraint[String](),
new='all-hands')
```

Again, the content of the program closely reflects that of the corresponding utterance. The `revise` operator takes three arguments:

- rootLoc, a constraint to find the top-level node of the original computation;
- oldLoc, a constraint on the node to replace within the original computation;
- new, a new graph fragment to substitute there.

The `revise` node evaluates to the root of a modified copy of the original computation, in which new now fills the role at the "old" location. Normally rootLoc is RoleConstraint(output), which selects a node returned by a previous user utterance; thus, we `revise` that entire original request.

Revision is non-destructive—no part of the dialogue history is lost, so entities computed by the original target and its ancestors remain available for later reference. However, the copy shares nodes with the original computation where possible, to avoid introducing unnecessary duplicate nodes that would have to be considered by `refer`.

For the example dialogue at the beginning of this section, the first turn produces the light gray nodes below. The second turn adds the darker gray nodes, which specify the desired revision.

Finally, evaluating the `revise` node selects the salient locations that match the rootLoc and oldLoc constraints (indicated in the above drawing by temporary dotted lines), and constructs the revised subgraph (the new start node below and its ancestors). The result of evaluation (dashed arrow below) is the root of the revised subgraph.
Finally, evaluating these new nodes as well will establish that the value of the top-level revise is the start time of the ‘all-hands’ meeting on Tuesday.

In the following example, the second utterance asks to replace a date specification. However, the utterance appears in a context where the relevant DateTimeSpec-valued node to be “replaced” is an argument that has actually not yet been provided:

<table>
<thead>
<tr>
<th>User:</th>
<th>When is lunch?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent:</td>
<td>It’s at 1 pm.</td>
</tr>
<tr>
<td>User:</td>
<td>What about tomorrow?</td>
</tr>
<tr>
<td>revise(</td>
<td>rootLoc=RoleConstraint(output),</td>
</tr>
<tr>
<td>oldLoc=Constraint[DateTimeSpec],</td>
<td>new=tomorrow())</td>
</tr>
</tbody>
</table>

The revision replaces the missing start argument to the previous EventSpec (whose absence had resulted in some default behavior) with an explicit argument (the DateTimeSpec returned by tomorrow()). To achieve this, when the salience retrieval model is run with an oldLoc constraint, it must be able to return missing arguments that satisfy that constraint. Missing arguments are implicitly present, with special value missing of the appropriate type. In practice they are created on demand.

Evaluating the revise node results in the new, revised subgraph pointed to by the dashed arrow, (which can then be evaluated):

Relatively, a user utterance sometimes modifies a previously mentioned constraint such as an EventSpec (see footnote 4). To permit this and more, we allow a more flexible version of revise to (non-destructively) transform the subgraph at oldLoc by applying a function, rather than by substituting a given subgraph new. Such functions are similar to rewrite rules in a term rewriting system (Klop, 1990), with the oldLoc argument supplying the condition. Our dataset (§6) makes heavy use of reviseConstraint calls, which modify a constraint as directed, while weakening it if necessary so that it remains satisfiable. For example, if a 3:00–3:30 meeting is onscreen and the user says make it 45 minutes or make it longer, then the agent can no longer preserve previous constraints start=3:00 and end=3:30; one must be dropped.

While the examples in this section involve a single update, real-world dialogues (§6) can involve single user requests built up over as many as five turns with unrelated intervening discussion. Revisions of revisions or of constraints on reference are also seamlessly handled: revise can take another revise or a refer node as its target, leading to a longer chain of result edges (dashed lines) to follow. Coordination of interactions among this many long-range dependencies remains a challenge even for modern attentional architectures (Bordes et al., 2016). With revise all the needed information is in one place; as experiments will show, this is crucial for good performance in more challenging dialogues.

5 Recovery

Sometimes users make requests that can be fulfilled only with the help of followup exchanges, if at all. Requests might be incomplete:

| User: | Book a meeting for me. |
| Agent: | When should the meeting start? |

referentially ambiguous:

| User: | Who is coming to the planning meeting? |
| Agent: | Susan Chu and Susan Brown. |
| User: | What is Susan’s email? |

or have no identifiable referent (a presupposition failure):

| User: | When is my first meeting on February 30? |

Our solution is to treat such discourse failures as exceptions. In principle, they are no different from other real-world obstacles to fulfilling the user’s request (server errors, declined credit cards, and
other business logic). To be useful, a dialogue model must have some way to recover from all these exceptions, describing the problem to the user and guiding the dialogue past it.

Our dialogue manager consists mainly of an exception recovery mechanism. This contrasts with traditional slot-filling systems, where a scripted policy determines which questions to ask the user and in which order. Scripted policies are straightforward but cannot handle novel compositional utterances. Contextual semantic parsers treat compositionality, but provide no dialogue management mechanism at all. Our dataflow-based approach allows the user to express complex compositional intents, but also allows the agent to reclaim the initiative when it is unable to make progress. Specifically, the agent can elicit interactive repairs of the problematic user plan: the user communicates such repairs through the reference and revision mechanisms described in preceding sections.

**Exceptions in execution**  In the dataflow graph framework, failure to interpret a user utterance is signaled by exceptions, which occur during evaluation. The simplest exceptions result from errors in function calls and constructors:

```
User: What do I have on February 30?
findEvent(EventSpec(
    start=DateTimeSpec(month=feb, day=30)))
```

Evaluation of the dataflow graph specified by this program cannot be completed. The DateTimeSpec constructor generates an exception, and descendants of that node remain unevaluated.

```
<table>
<thead>
<tr>
<th>feb</th>
<th>month</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
```

An exception is essentially just a special result (possibly a structured value) returned by evaluation. It appears in the dataflow graph, so the agent can condition on it when predicting programs in future turns. When an exception occurs, the generation model (§2) is invoked on the exceptional node. This can be used to produce prompts like:

```
Agent: There is no 30th of February. Did you mean some other date?
```

At this point, recovering from the exception looks like any other revision step: the user supplies a new value, and the agent simply needs to patch it into the right location in the dataflow graph. There are several answers the user could make, indicating repairs at different locations:

```
User: I mean February 28.
revise(rootLoc=RoleConstraint(output),
    oldLoc=Constraint[DateTimeSpec()],
    new=DateTimeSpec(month=Feb, day=28))
```

```
User: I mean March.
revise(rootLoc=RoleConstraint(output),
    oldLoc=Constraint[Month],
    new=mar)
```

The fact that exception recovery looks like any other turn-level prediction is another key advantage of dataflow-based state representations. In the above examples, the user specified a revision that would enable them to continue, but they also would have been free to try another utterance (*List all my meetings in February*) or to change goals altogether (*Never mind, let’s schedule a vacation*).

Because of its flexibility, our exception-handling mechanism is suitable for many situations that have not traditionally been regarded as exceptions. For example, an interactive slot-filling workflow can be achieved via a sequence of underspecified constructors, each triggering an exception and eliciting a revision from the user:

```
User: Create a meeting.
createEvent()
--→ UnderconstrainedException!(name)
Agent: What should it be called?
User: Planning meeting.
revise(rootLoc=RoleConstraint(output),
    oldLoc=RoleConstraint(name),
    new='Planning meeting')
--→ UnderconstrainedException!(start)
Agent: When should it start?
```

The agent predicted that the user intended to revise the missing name because an exception involving the name path appeared in the dialogue history on the previous turn.

Recovery behaviors are enabled by the phase separation between constructing the dataflow graph (which is the job of program synthesis from natural language) and evaluating its nodes. The dataflow graph always contains a record of the user’s current goal, even when the goal could not be successfully evaluated. This goal persists
across turns and remains accessible to reference, and thus can be interactively refined and clarified using the same metacomputation operations as user-initiated revision. Exception handling influences the course of the dialogue, without requiring a traditional hand-written or learned “dialogue policy” that reasons about full dialogue states. Our policy only needs to generate language (recall §2) that reacts appropriately to any exception or exceptions in the evaluation of the most recent utterance’s program, just as it reacts to the ordinary return value in the case where evaluation succeeds.

6 Data

To validate our approach, we crowdsourced a large English dialogue dataset, SMCalFlow, featuring task-oriented conversations about calendar events, weather, places, and people. Figure 2 has an example. SMCalFlow has several key characteristics: 

Richly annotated: Agent responses are executable programs, featuring API calls, function composition, and complex constraints built from strings, numbers, dates and times in a variety of formats. They are not key-value structures or database queries, but instead full descriptions of the runtime behavior needed to react to the user in a real, grounded dialogue system.

Open-ended: We did not constrain crowdworkers to scripts. Instead, they were given general information about agent capabilities and were encouraged to interact freely. A practical dialogue system must also recognize and respond to out-of-scope requests. Our dataset includes many such examples (see the fourth user turn in Figure 2).

Cross-domain: SMCalFlow spans four major domains: calendar, weather, places, and people. Cross-domain interaction is pervasive (Figure 2).

To cover a rich set of back-end capabilities while encouraging worker creativity, we designed a wide range of scenarios to guide dialogue construction. There are over 100 scenarios of varying topic and granularity. Dialogues are collected via a Wizard-of-Oz process. Every dialogue is associated with a scenario. At each turn, a crowdworker acting as the user is presented with a dialogue as context and is asked to append a new utterance. An annotator acting as the agent labels the utterance with a program (which may include refer and revise) and then selects a natural-language response from a set of candidates produced by the language generation model described in §2. The

User: Can you remind me to go to the airport tomorrow morning at 8am?
createCommitEventWrapper(
    createPreflightEventWrapper(
        EventBuilder(
            subject=’go to the airport’,
            start=dateAtTime(date=tomorrow(),
                time=numberAM(8))))))
Agent: Does this look right? [showing an event card with start at 8am and the event name]
User: Make the location LaGuardia Airport.
reviseConstraint(
    rootLoc=RoleConstraint(output),
    oldLoc=Constraint[EventBuilder](),
    newEventBuilder(
        location=LocationKeyphrase('LaGuardia Airport')))
Agent: How is this? [showing an updated card]
User: What can I expect the weather there to be tomorrow morning?
weatherQueryApi(place=findPlace(PlaceSpec(keyphrase=refer(
    Constraint[LocationKeyphrase]()))),
    time=DateTimeSpec(date=tomorrow(),
        constraint=morning()))
Agent: It will be partly cloudy with a high of 77 °F and a low of 67 °F in LaGuardia Airport between 6:00 AM and 12:00 PM.
User: How is the traffic on the way there?
fenceNavigation()
Agent: I can’t answer questions about transit.
User: That’s fine. Book that please.
confirmAndReturnAction()
Agent: I’ve put that on your calendar. [showing the confirmed event card]

Figure 2: A sample annotated dialogue in SMCalFlow. Turn 1 features free-text subject and date/time. Turn 2 features reviseConstraint. Turn 3 features cross-domain interaction via refer and nested API calls (findPlace and weatherQueryApi are both real-world APIs). Turn 4 features an out-of-scope utterance that is parried by a category-appropriate “fencing” response. Turn 5 confirms a proposal after intervening turns.

annotation interface includes an autocomplete feature based on existing annotations. Annotators also populate databases of people and events to ensure that user requests have appropriate responses. The process is iterated for a set number of turns or until the annotator indicates the end of conversation. A single dialogue may include turns from multiple crowdworkers and annotators.

Annotators are provided with detailed guidelines containing example annotations and information about available library functions. Guidelines also specify conventions for pragmatic issues like
the decision to annotate next as after at the beginning of §2. Crowdworkers are recruited from Amazon Mechanical Turk with qualification requirements such as living in the United States and with a work approval rate higher than 95%.

Data is split into training, development, and test sets. We review every dialogue in the test set with two additional annotators. 75% of turns pass through this double review process with no changes, which serves as an approximate measure of inter-annotator consensus on full programs.

For comparison, we also produce a version of the popular MultiWOZ 2.1 dataset (Budzianowski et al., 2018; Eric et al., 2019) with dataflow-based annotations. MultiWOZ is a state tracking task, so in its original format the dataset annotates each turn with a dialogue state rather than an executable representation. To obtain an equivalent (program-based) representation for MultiWOZ, at each user turn we automatically convert the annotation to a dataflow program. Specifically, we represent each non-empty dialogue state as a call to an event booking function, $\text{find}$, whose argument is a constraint that specifies the desired type of booking along with values for some of that type’s slots. Within a dialogue, any turn that initiates a new type of booking is re-annotated as a call to $\text{find}$. Turns that merely modify some of the slots are re-annotated as $\text{reviseConstraint}$ calls. Within either kind of call, any slot value that does not appear as a substring of the user’s current utterance (all slot values in MultiWOZ are utterance substrings) is re-annotated as a call to $\text{refer}$ with an appropriate type constraint, provided that the reference resolution heuristic would retrieve the correct string from earlier in the dataflow. This covers references like the same day. Otherwise, our re-annotation retains the literal string value.

Data statistics are shown in Table 1. To the best of our knowledge, SMCalFlow is the largest annotated task-oriented dialogue dataset to date. Compared to MultiWOZ, it features a larger user vocabulary, a more complex space of state-manipulation primitives, and a long tail of agent programs built from numerous function calls and deep composition.

7 Experiments

We evaluate our approach on SMCalFlow and MultiWOZ 2.1. All experiments use the OpenNMT (Klein et al., 2017) pointer-generator network (See et al., 2017), a sequence-to-sequence model that can copy tokens from the source sequence while decoding. Our goal is to demonstrate that dataflow-based representations benefit standard neural model architectures. Dataflow-specific modeling might improve on this baseline, and we leave this as a challenge for future work.

For each user turn $i$, we linearize the target program into a sequence of tokens $z_i$. This must be predicted from the dialogue context—namely the concatenated source sequence $x_{i-c} \ldots x_{i-1} x_i z_{i-1} x_i$ (for SMCalFlow) or $y_{i-c} \ldots y_{i-1} y_i z_{i-1} x_i$ (for MultiWOZ 2.1). Here $c$ is a context window size, $x_j$ is the user utterance at user turn $j$, $y_j$ is the agent’s natural-language response, and $z_j$ is the linearized agent program. Each sequence $x_j$, $y_j$, or $z_j$ begins with a separator token that indicates the speaker (user or agent). Our formulation of context for MultiWOZ is standard (e.g., Wu et al., 2019). We take the source and target vocabularies to consist of all words that occur in (respectively) the source and target sequences in training data, as just defined.

The model is trained using the Adam optimizer (Kingma and Ba, 2015) with the maximum likelihood objective. We use 0.001 as the learning rate. Training ends when there have been two different epochs that increased the development loss.

We use Glove800B-300d (cased) and Glove6B-300d (uncased) (Pennington et al., 2014) to initialize the vocabulary embeddings for the SMCalFlow and MultiWoZ experiments, respectively. The context window size $c$, hidden layer size $d$, number of hidden layers $l$, and dropout rates $r$ are selected based on the agent action accuracy (for SMCalFlow) or dialogue-level exact match (for MultiWoZ) on the development set from \{2, 4, 10\}, \{256, 300, 320, 384\}, \{1, 2, 3\}, \{0.3, 0.5, 0.7\} respectively. Approximate 1-best decoding uses a beam of size 5.

Quantitative evaluation Table 2 shows results for the SMCalFlow dataset. We report program accuracy: specifically, exact-match accuracy of the predicted program after inlining metacomputation (i.e., replacing all calls to metacomputation operations with the concrete program fragments they return). We also compare to baseline mod-

\footnote{Specifically, we inline all $\text{refer}$ calls and $\text{revise}$ calls that involve direct substitution of the kind described in §4. We preserve $\text{reviseConstraint}$ calls to avoid penalizing...}
calls, they are 10, 320, 2, and 0.7. Even on inlined metacomputation. These

Table 2

<table>
<thead>
<tr>
<th># Dialogues</th>
<th># User Turns</th>
<th>User Vocab. Size</th>
<th>Library Size</th>
<th>Utterance Length</th>
<th>Program Length</th>
<th>Program Depth</th>
<th>OOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMCalFlow</td>
<td>41,517</td>
<td>155,923</td>
<td>17,397</td>
<td>338</td>
<td>(5, 8, 11)</td>
<td>(11, 40, 57)</td>
<td>(5, 9, 11)</td>
</tr>
<tr>
<td>MultiWOZ 2.1</td>
<td>10,419</td>
<td>71,410</td>
<td>4,105</td>
<td>35</td>
<td>(9, 13, 17)</td>
<td>(2, 29, 39)</td>
<td>(2, 6, 6)</td>
</tr>
</tbody>
</table>

Table 1: Dataset statistics. “Library Size” counts distinct function names (e.g., `findEvent`) plus keyword names (e.g., `start`). “Length” and “Depth” columns show (.25, .50, .75) quantiles. For programs, “Length” is the number of function calls and “Depth” is determined from a tree-based program representation. “OOS” counts the out-of-scope utterances. MultiWOZ statistics were calculated after applying the data processing of Wu et al. (2019). Vocabulary size is less than reported by Goel et al. (2019) because of differences in tokenization (see code release).

Table 2: SMCalFlow results. Agent action accuracy is significantly higher than a baseline without metacomputation, especially on turns that involve a reference (Ref. Turns) or revision (Rev. Turns) to earlier turns in the dialogue ($p < 10^{-6}$, McNemar’s test).

For the full representation, $c$, $d$, $l$, and $r$ are 2, 384, 2, and 0.5, respectively. For the inline variant, they are 2, 384, 3, and 0.5. Turn-level exact match accuracy is around 73% for the development set and 67% for the test set. Inlining metacomputation, which forces the model to explicitly resolve cross-turn computation, reduces accuracy by 5.9% overall, 10.9% on turns involving references, and 9.1% on turns involving revision. Dataflow-based metacomputation operations are thus essential for good model performance in all three cases.

We further evaluate our approach on dialogue state tracking using MultiWOZ 2.1. Table 3 shows results. For the full representation, the selected model uses $c = 2$, $d = 384$, $l = 2$, and $r = 0.7$. For the inline `refer` variant, they are 4, 320, 3, and 0.3. For the variant that inlines both `refer` and `revise` calls, they are 10, 320, 2, and 0.7. Even without metacomputation, prediction of program-based representations gives results comparable to the existing state of the art, TRADE, on the standard “Joint Goal” metric (turn-level exact match).

<table>
<thead>
<tr>
<th># of Turns</th>
<th>Full dev test</th>
<th>Ref. Turns dev test</th>
<th>Rev. Turns dev test</th>
</tr>
</thead>
<tbody>
<tr>
<td>inline</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: MultiWOZ 2.1 test set results. TRADE (Wu et al., 2019) results are from the public implementation. “Joint Goal” (Budzianowski et al., 2018) is average dialogue state exact-match, “Dialogue” is average dialogue-level exact-match, and “Prefix” is the average number of turns before an incorrect prediction. Within each column, the best result is boldfaced, along with all results that are not significantly worse ($p < 0.05$, paired permutation test). Moreover, all of “Dataflow,” “inline `refer`,” and “inline both” have higher dialogue accuracy than TRADE ($p < 0.005$).

(Our dataflow representation for MultiWOZ is designed so that dataflow graph evaluation produces native MultiWOZ slot-value structures.) However, Joint Goal does not fully characterize the effectiveness of a state tracking system in real-world interactions, as it allows the model to recover from an error at an earlier turn by conditioning on gold agent utterances after the error. We thus evaluate on dialogue-level exact match and prefix length (the average number of turns until an error). On these metrics the benefit of dataflow over past approaches is clearer. Differences within dataflow model variants are smaller here than in Table 2. For the Joint Goal metric, the no-metacomputation baseline is better; we attribute this to the comparative simplicity of reference in the MultiWOZ dataset. In any case, casting the state-tracking problem as one of program prediction with appropriate primitives gives a state-of-the-art state-tracking model for MultiWOZ using only off-the-shelf sequence prediction tools.\textsuperscript{3}

\textsuperscript{3}A note on reproducibility: Dependence on internal libraries prevents us from releasing a full salience model implementation and inlining script for SMCalFlow. The accompanying data release includes both inlined and non-inlined
Error category | Count
---|---
**Underprediction** | 21
**Entity linking** | 21
  - Hallucinated | 7
  - Wrong type | 7
  - Wrong field | 2
  - Boundary mismatch | 5
**Fencing** | 22
  - Should have fenced | 9
  - Shouldn’t have fenced | 8
  - Wrong message | 5
**Ambiguity** | 23
  - Wrong in context | 8
  - Acceptable (same semantics) | 8
  - Acceptable (different semantics) | 7
**Miscellaneous** | 10
  - Used wrong function | 4
  - Other / Multiple | 6
**Error in gold** | 3

Table 4: Manual classification of 100 model errors on the SMCalFlow dataset. The largest categories are underprediction (omitting steps from agent programs), entity linking (errors in extraction of entities from user utterances), fencing (classifying a user request as out-of-scope), and ambiguity (user utterances with multiple possible interpretations). See §7 for discussion.

**Error analysis** Beyond the quantitative results shown in Tables 2–3, we manually analyzed 100 SMCalFlow turns where our model mispredicted. Table 4 breaks down the errors by type.

Three categories involve straightforward parsing errors. In underprediction errors, the model fails to predict some computation (e.g., a search constraint or property extractor) specified in the user request. This behavior is not specific to our system: under-length predictions are also well-documented in neural machine translation systems (Murray and Chiang, 2018). In entity linking errors, the model correctly identifies the presence of an entity mention in the input utterance, but uses it incorrectly in the input plan. Sometimes the entity that appears in the plan is hallucinated, appearing nowhere in the utterance; sometimes the entity is cast to a wrong type (e.g., locations interpreted as event names) used in the wrong field or extracted with wrong boundaries. In fencing errors, the model interprets an out-of-scope user utterance as an interpretable command, or vice-versa.

The fourth category, ambiguity errors, is more interesting. In these cases, the predicted plan corresponds to an interpretation of the user utterance that would be acceptable in some discourse context. In a third of these cases, this interpretation is ruled out by either dialogue context (e.g., interpreting what’s next? as a request for the next list item rather than the event with the next earliest start time) or commonsense knowledge (make it at 8 means 8 a.m. for a business meeting and 8 p.m. for a dance party). In the remaining cases, the predicted plan expresses an alternative computation that produces the same result, or an alternative interpretation that is also contextually appropriate.

8 Related work

The view of dialogue as an interactive process of shared plan synthesis dates back to Grosz and Sidner’s earliest work on discourse structure (1986; 1988). That work represents the state of a dialogue as a predicate recognizing whether a desired piece of information has been communicated or change in world state effected. Goals can be refined via questions and corrections from both users and agents. The only systems to attempt full versions of this shared-plans framework (e.g., Allen et al., 1996; Rich et al., 2001) required inputs that could be parsed under a predefined grammar. Subsequent research on dialogue understanding has largely focused on two simpler subtasks:

**Contextual semantic parsing** approaches focus on complex language understanding without reasoning about underspecified goals or agent initiative. Here the prototypical problem is iterated question answering (Hemphill et al., 1990; Yu et al., 2019b), in which the user asks a sequence of questions corresponding to database queries, and results of query execution are presented as structured result sets. Vlachos and Clark (2014) describe a semantic parsing representation targeted at more general dialogue problems. Most existing methods interpret context-dependent user questions (What is the next flight to Atlanta? When does it land?) by learning to copy subtrees (Zettlemoyer and Collins, 2009; Iyyer et al., 2017; Suhr et al., 2018) or tokens (Zhang et al., 2019) from previously-generated queries. In contrast, our approach refines reuse with explicit graph operators.

**Slot-filling** approaches (Pieraccini et al., 1992) model simpler utterances in the context of full,
teractive dialogues. It is assumed that any user intent can be represented with a flat structure consisting of a categorical dialogue act and a mapping between a fixed set of slots and string-valued fillers. Existing fine-grained dialogue act schemes (Stolcke et al., 2000) can distinguish among a range of communicative intents not modeled by our approach, and slot-filling representations have historically been easier to predict (Zue et al., 1994) and annotate (Byrne et al., 2019). While recent variants support interaction between related slots (Budzianowski et al., 2018) and fixed-depth hierarchies of slots (Gupta et al., 2018), modern slot-filling approaches remain limited in their support for semantic compositionality. By contrast, our approach supports user requests corresponding to general compositional programs.

More recent end-to-end dialogue agents attempt to map directly from conversation histories to API calls and agent utterances using neural sequence-to-sequence models without a representation of dialogue state (Bordes et al., 2016; Yu et al., 2019a). While promising, models in these papers fail to outperform rule- or template-driven baselines. Neelakantan et al. (2019) report greater success on a generation-focused task, and promising results have also been obtained from hybrid neuro-symbolic dialogue systems (Zhao and Eskenazi, 2016; Williams et al., 2017; Wen et al., 2017; Gao et al., 2019). Much of this work is focused on improving agent modeling for existing representation schemes like slot filling. We expect that many modeling innovations (e.g., the neural entity linking mechanism proposed by Williams et al.) could be used in conjunction with the new representational framework we have proposed in this paper.

Like slot-filling approaches, our framework is aimed at modeling full dialogues in which agents can ask questions, recover from errors, and take actions with side effects, all backed by an explicit state representation. However, our notions of “state” and “action” are much richer than in slot-filling systems, extending to arbitrary compositions of primitive operators. We use semantic parsing as a modeling framework for dialogue agents that can construct compositional states of this kind. While dataflow-based representations are widely used to model execution state for programming languages (Kam and Ullman, 1976), this is the first work we are aware of that uses them to model conversational context and dialogue.

9 Conclusions

We have presented a representational framework for task-oriented dialogue modeling based on dataflow graphs, in which dialogue agents predict a sequence of compositional updates to a graphical state representation. This approach makes it possible to represent and learn from complex, natural dialogues. Future work might focus on improving prediction by introducing learned implementations of refer and revise that, along with the program predictor itself, could evaluate their hypotheses for syntactic, semantic, and pragmatic plausibility. The representational framework could itself be extended, e.g., by supporting declarative user goals and preferences that persist across utterances. We hope that the rich representations presented here—as well as our new dataset—will facilitate greater use of context and compositionality in learned models for task-oriented dialogue.

Acknowledgments

We would like to thank Tatsunori Hashimoto, Jianfeng Gao, and the anonymous TACL reviewers for feedback on early drafts of this paper.

References


