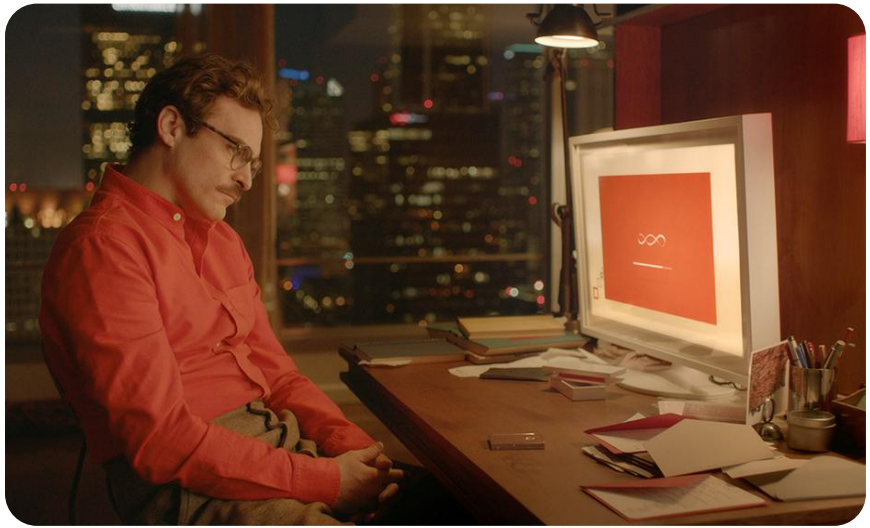
Putting The Conversation in Conversational AI

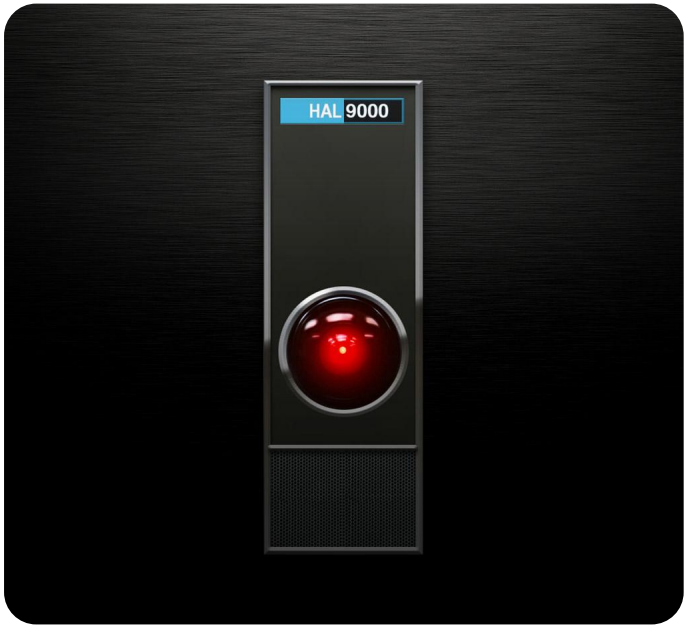
Ozlo | Scott Martin | 4/6/17

How to approach this?

Make it be your friend?



Teach it to read lips?



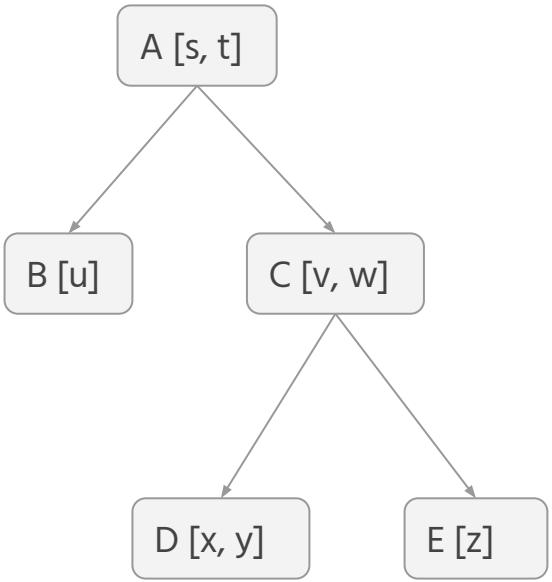
Some conversational design principles

- Allow the user to interact as freely as possible, while still maintaining an understanding of how to respond
- Give up some *system initiative* (think automated phone tree) in favor of more *user initiative*:
  - **Underspecification**: not providing enough information
  - **Overanswering**: providing more information than was requested
  - **Overspecification**: creating an ambiguous situation
- Mimic natural human interaction by allowing the user to change the subject, but come back to what we were talking about **without forgetting anything**

How I got pulled into this

- Trying to build a domain-general dialog system on my first computer was humbling
- Dialog is hard! Many complex, interacting parts that are mostly invisible
- Lesson learned: start with the simplest parts, and try to get those right

A simple task decomposition scheme



Tasks may have

- Slots that may be parameterized in advance
- Dependencies on other tasks

Intents are mapped to tasks by rank

Dialog

Highly ranking tasks matched

User: *fried chicken*

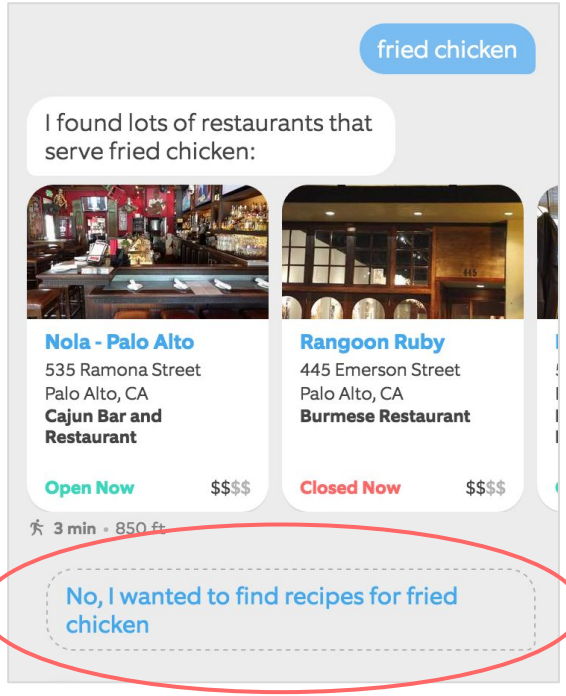
- This utterance yields two competing, highly ranked task decompositions

findRestaurant [dish=fried chicken] *0.99*

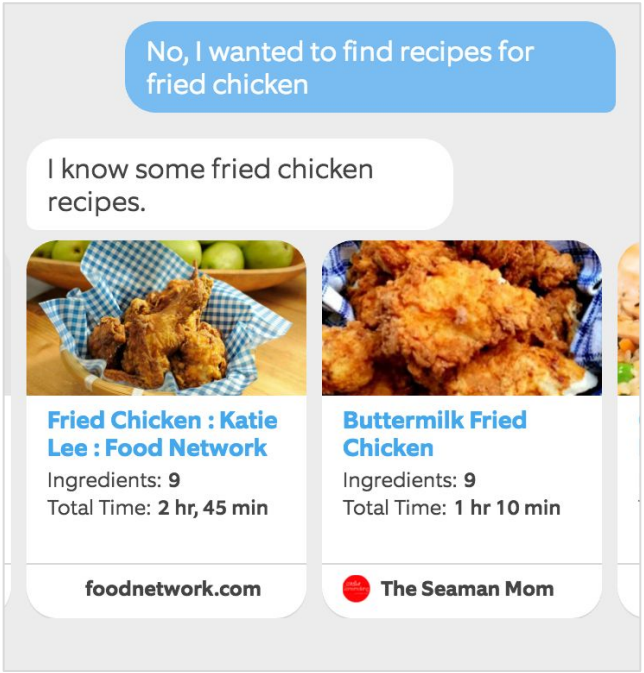
findRecipes [dish=fried chicken] *0.924747*

Task ranking allows cross-domain ambiguity

Highest ranking task

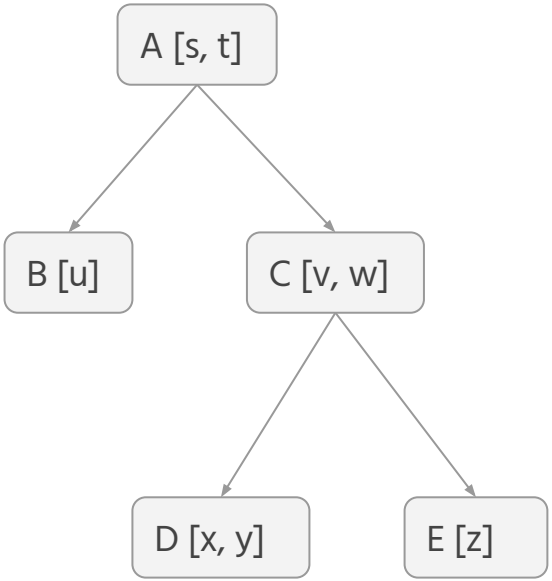


Alternative task



Task execution

Specification



Execution stack

E [z]

D [x, y]

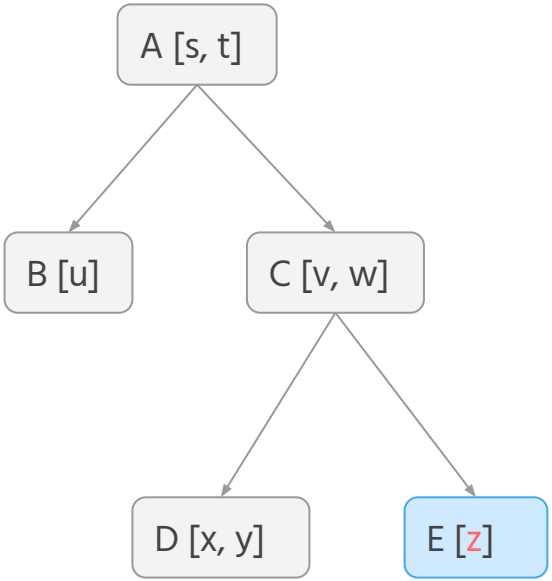
C [v, w]

B [u]

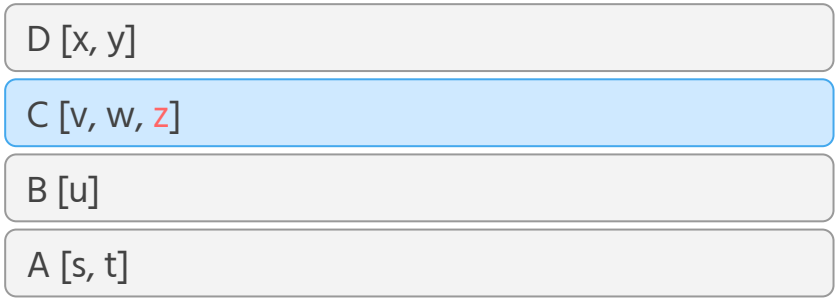
A [s, t]

Runtime inheritance of slot values

Specification

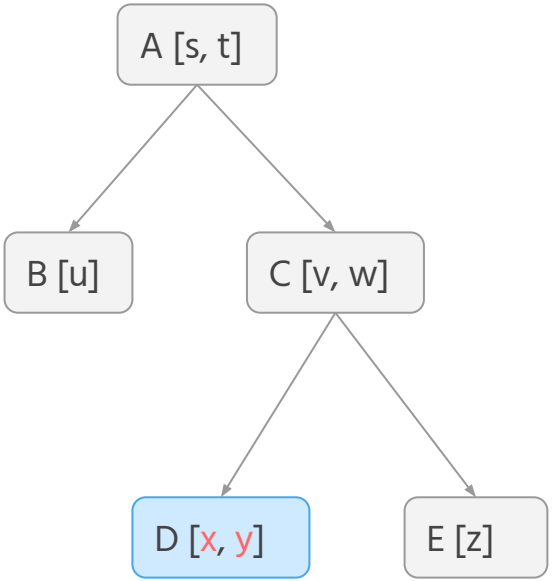


Execution stack



Runtime inheritance of slot values

Specification



Execution stack

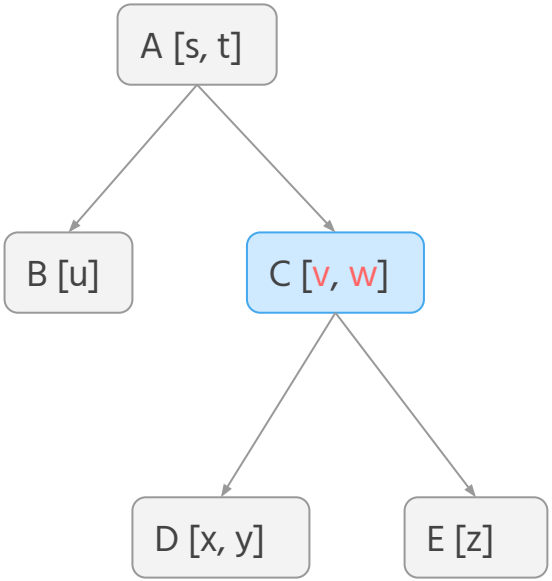
C [v, w, x, y, z]

B [u]

A [s, t]

Runtime inheritance of slot values

Specification



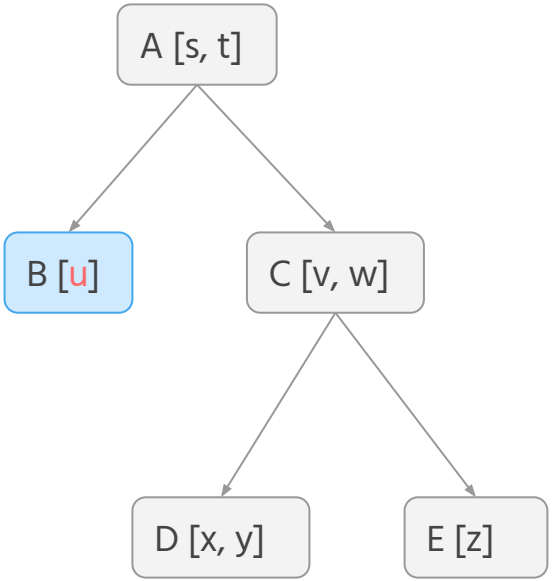
Execution stack

B [u]

A [s, t, v, w, x, y, z]

Runtime inheritance of slot values

Specification

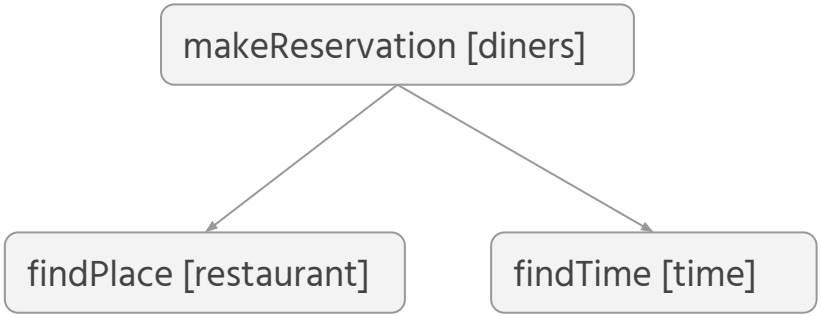


Execution stack

A [s, t, u, v, w, x, y, z]

A meal reservation task decomposition

Specification



Execution stack

findTime [time=?]

findPlace [restaurant=?]

makeReservation [diners=?]

Underspecification triggers a dialog

Dialog

User: *Make a reservation on Friday at 8*
Ozlo: *Where would you like to eat?*

- Underspecification is handled by running subtasks until we reach one that has an unfilled slot

Execution stack

findPlace [restaurant=?]

makeReservation [diners=?, time=Friday 8 PM]

Underspecification triggers a dialog

Dialog

User: *Make a reservation on Friday at 8*
Ozlo: *Where would you like to eat?*
User: *at Evvia*
Ozlo: *How many people will be eating?*

- Execution continues until all of the root task's slots are properly filled

Execution stack

makeReservation [diners=?, time=Friday 8 PM, restaurant=Evvia]

Underspecification triggers a dialog

Dialog

User: *Make a reservation on Friday at 8*
Ozlo: *Where would you like to eat?*
User: *at Evvia*
Ozlo: *How many people will be eating?*
User: *two people*
Ozlo: *Ok, I'll make a reservation for two at Evvia on Friday at 8*

Execution stack

makeReservation [diners=2, time=Friday 8 PM, restaurant=Evvia]

A one-shot query is a special case of a dialog

Dialog

User: *make a reservation for 2 at Evvia on Friday at 8*

- We can gather all the parameters at once, essentially a complete dialog in a single turn
- Execution proceeds just as it would with a multi-turn dialog

Initial execution stack

findTime [time=Friday 8 PM]

findPlace [restaurant=Evvia]

makeReservation [diners=2]

Final execution stack

makeReservation [diners=2, time=Friday 8 PM, restaurant=Evvia]

Overanswering gives more flexibility

Dialog

User: *I want to make a reservation for dinner*

- We don't want to require a rigid order for the user's answers
- Slot values may be specified in **any order**, because all tasks on the stack get to inspect the input

Execution stack

findTime [time=?]

findPlace [restaurant=?]

makeReservation [diners=?]

Overanswering gives more flexibility

Dialog

User: *I want to make a reservation for dinner*
Ozlo: *Ok, what time should I make the reservation for?*
User: *It's for two people on Friday at 8.*

- Each task on the stack is allowed to inspect the input on every turn in case its values were provided

Execution stack

findPlace [restaurant=?]

makeReservation [diners=2, time=Friday 8 PM]

Overanswering gives more flexibility

Dialog

User: *I want to make a reservation for dinner*
Ozlo: *Ok, what time should I make the reservation for?*
User: *It's for two people on Friday at 8.*
Ozlo: *Where would you like to eat?*
User: *At Evvia.*
Ozlo: *Ok, I'll make a reservation for two at Evvia on Friday at 8*

Execution stack

```
makeReservation [diners=2, time=Friday 8 PM,
restaurant=Evvia]
```

Overspecification triggers disambiguation

Dialog

User: *Make a reservation at Evvia*
Ozlo: *What time should I make the reservation?*
User: *On Friday at 8 at Rangoon Ruby.*
Ozlo: Where do you want to eat? At Evvia or at Rangoon Ruby?

- If too many values are provided for a slot, we pause to clarify what the user meant

Execution stack

makeReservation [diners=?, time=Friday 8 PM, restaurant={Evvia, Rangoon Ruby}]

Overspecification triggers disambiguation

Dialog

User: *Make a reservation at Evvia*
Ozlo: *What time should I make the reservation?*
User: *On Friday at 8 at Rangoon Ruby.*
Ozlo: Where do you want to eat? At Evvia or at Rangoon Ruby?
User: *Actually, at Evvia.*
Ozlo: *How many people will be eating?*

- Once the ambiguity is resolved, we proceed with the execution

Execution stack

makeReservation [diners=?, time=Friday 8 PM, restaurant=Evvia]

Dialogs can be induced from data

- We can create tasks from Ozlo's data schemas to handle dialogs about a certain domain
- These tasks use the same execution mechanism as hand-coded tasks, but are automatically generated

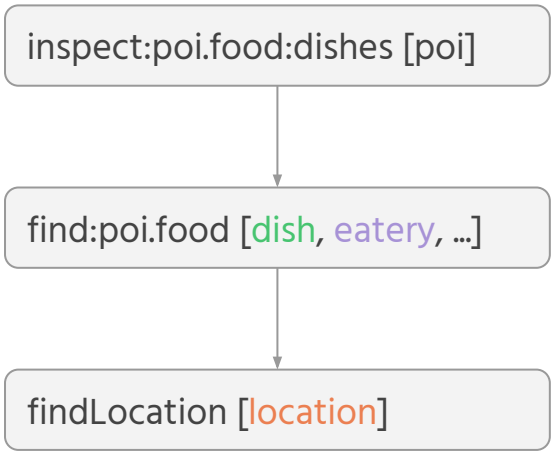
Example schemas

```
poi

address
coordinates
...
```

```
poi.food extends poi

name
cuisines
dishes
dietary constraints
eatery types
...
```

Dialogs can be induced from data

Generated specification for dishes field



Example schemas

```
poi

address
coordinates
...
```

```
poi.food extends poi

name
cuisines
dishes
dietary constraints
eatery types
...
```

Dialogs can be induced from data

Dialog

User: *Does that cafe on University have croissants?*

- We first find the embedded location, then look for cafes at that location

Execution stack

findLocation [location=University Ave]

find:poi.food [dish=croissant, eatery=cafe]

inspect:poi.food:dishes [poi=?]

Dialogs can be induced from data

Dialog

User: *Does that cafe on University have croissants?*
Ozlo: *Which coffeeshop did you mean? Blue Bottle Coffee, Cafe Venetia, or Cafe Epi?*

- Finding more than one result triggers disambiguation

Execution stack

inspect:poi.food:dishes [poi={Blue Bottle Coffee, Cafe Venetia, Cafe Epi}, location=University Ave, dish=croissant, eatery=cafe]

Dialogs can be induced from data

Dialog

User: *Does that cafe on University have croissants?*
Ozlo: *Which coffeeshop did you mean? Blue Bottle Coffee, Cafe Venetia, or Cafe Epi?*
User: *Cafe Epi*
Ozlo: *Cafe Epi, a coffee shop on University Ave, serves several varieties of croissant.*

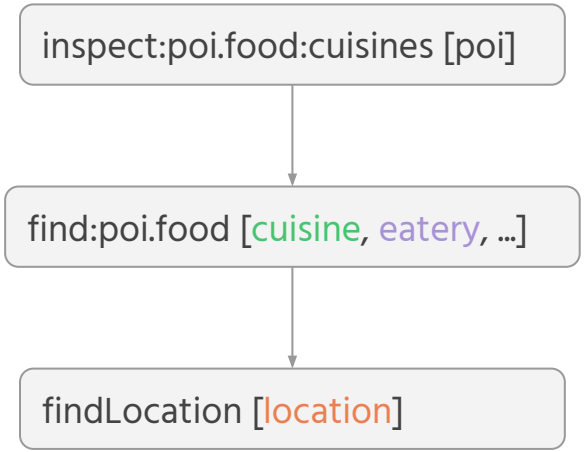
- Once the ambiguity is resolved, we can generate a useful response

Execution stack

inspect:poi.food:dishes [poi=Cafe Epi, location=University Ave, dish=croissant, eatery=cafe]

Other attributes work automatically too

Generated decomposition for cuisines field



Schemas

```
poi

address
coordinates
…
```

```
poi.food extends poi

name
cuisines
dishes
dietary constraints
eatery types
…
```

Other attributes work automatically too

Dialog

User: *Which of those restaurants on Emerson serve Greek food?*
Ozlo: *Evvia Estiatorio is a restaurant on Emerson Street that serves Greek food.*

Execution stack

inspect:poi.food:cuisines [poi=Evvia, location=Emerson St, cuisine=Greek, eatery=restaurant]

Taking stock

A more natural, freely flowing conversation

- User has more initiative, and is less constrained to interact how the system prefers
- Allows for underspecification, overanswering, overspecification
- Ambiguity resolution both between and within domains

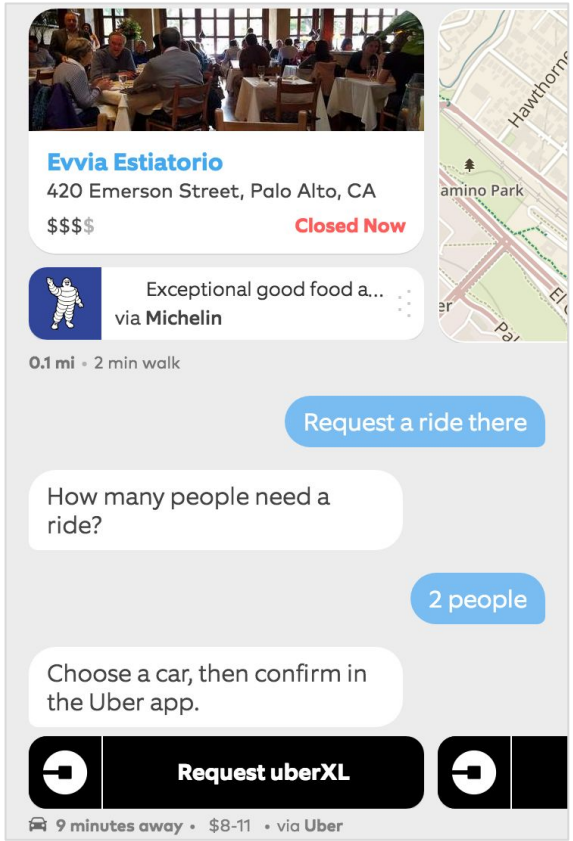
Ergonomic software design and development

- Simple task decomposition and execution model allows for modularity and reuse
- Straightforward to add support for more domains and more functionality
- Task decompositions for a wide range of domains can be automatically derived from properly structured data

Coming soon to a computer near you

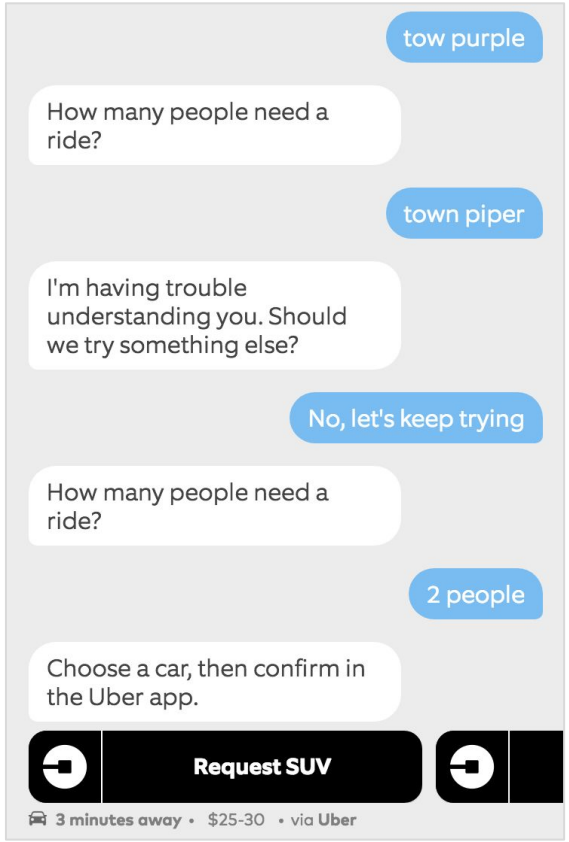
- Release as Ozlo's **Converse** API, likely later this year

Parts are visible in the Ozlo testing app now



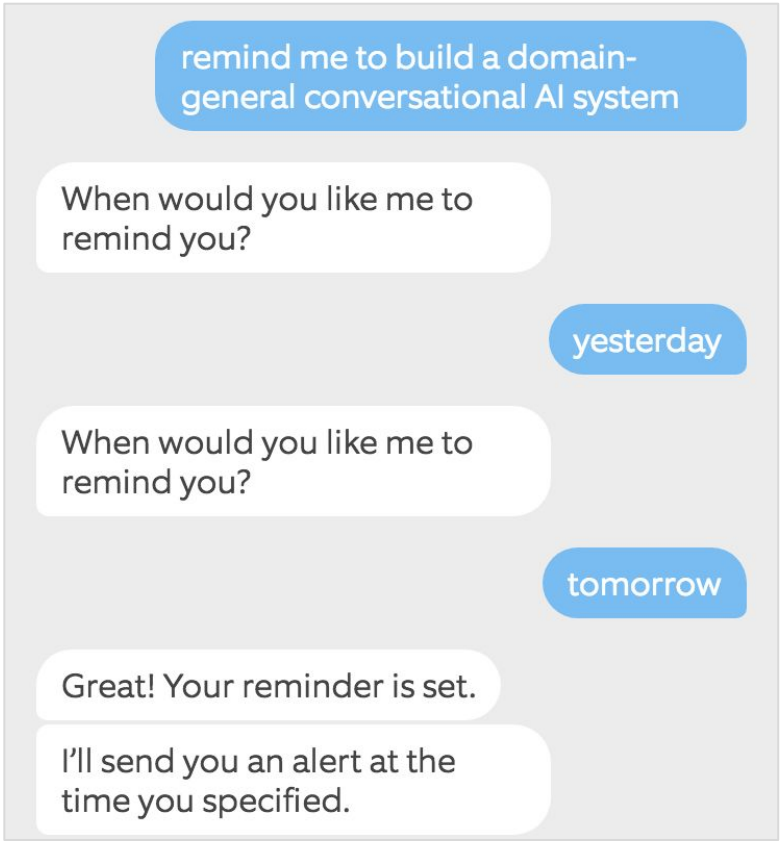
- In certain domains, we already enable dialogs using this technology
- One example is the ability to request an Uber ride after searching for a restaurant

Parts are visible in the Ozlo testing app now



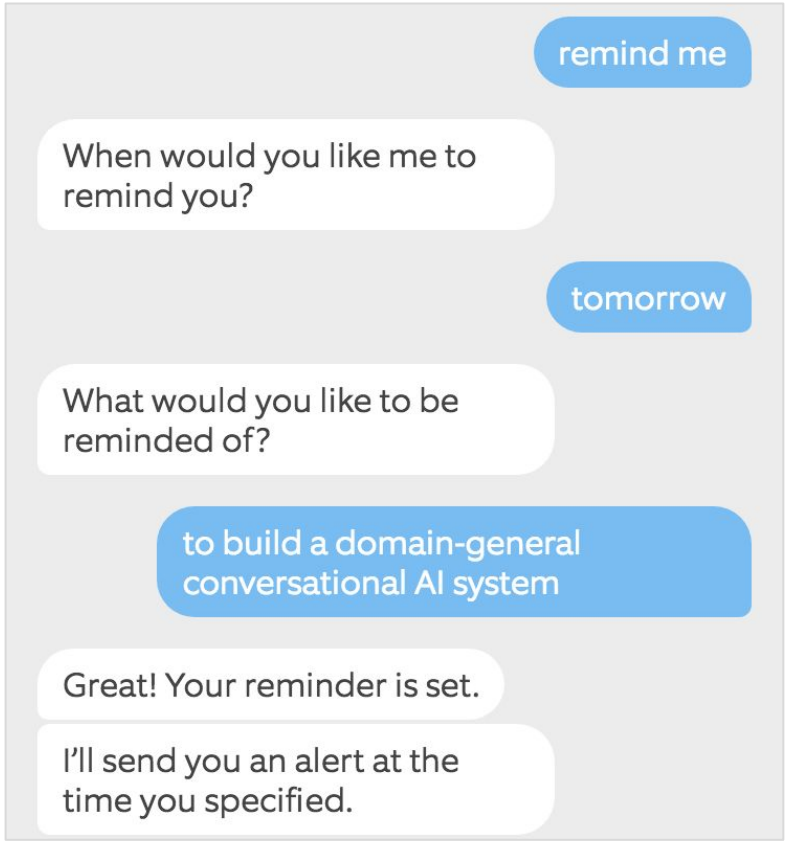
- After trying to disambiguate a few times, we give the user the chance to give up

Parts are visible in the Ozlo testing app now



- We carry on limited dialogs in order to set reminders, asking *what* and *when*

Parts are visible in the Ozlo testing app now



- The task execution model can be seen by not specifying what you want to be reminded of

Coming attraction: meta-conversations

Dialog

User: *Make a reservation on Friday at 8*
Ozlo: *Where would you like to eat?*
User: *at Evvia*
Ozlo: *How many people will be eating?*
User: *Actually, will it be raining then?*
Ozlo: *Rain is not forecast for Friday evening in Palo Alto.*

- This model allows the user to change the subject while in the middle of a task

Execution stack

weather [time=Friday 8 PM, condition=rain]

makeReservation [diners=?, time=Friday 8 PM, restaurant=Evvia]

Coming attraction: meta-conversations

Dialog

User: *Make a reservation on Friday at 8*
Ozlo: *Where would you like to eat?*
User: *at Evvia*
Ozlo: *How many people will be eating?*
User: *Actually, will it be raining then?*
Ozlo: *Rain is not forecast for Friday evening in Palo Alto.*
Ozlo: *Would you like to continue making a reservation at Evvia for Friday?*

- Since we know when a task from outside the selected task decomposition was run, we can make conversation about the conversation

Execution stack

```
makeReservation [diners=?, time=Friday 8 PM,
restaurant=Evvia]
```

Caveat: *Converse* probably won't work on this

Thanks for listening!

Questions?

Shout outs

- Dialog induction is joint work with Todd Agulnick and Mike Hanson
- The task that makes reminders was written by Felix Yuan
- Heather Tompkins helped make these slides look fantastic