Appendix

Data Collection

Data collection was done in two stages. In the first stage, we collected if-then-because commands from humans subjects. In the second stage, a team of annotators annotated the data with commonsense presumptions. Below we explain the details of the data collection and annotation process.

In the data collection stage, we asked a pool of human subjects to write commands that follow the general format: if ⟨ state holds ⟩ then ⟨ perform action ⟩ because ⟨ i want to achieve goal ⟩. The subjects were given the following instructions at the time of data collection:

" Imagine the two following scenarios:

Scenario 1: Imagine you had a personal assistant that has access to your email, calendar, alarm, weather and navigation apps, what are the tasks you would like the assistant to perform for your day-to-day life? And why?

Scenario 2: Now imagine you have an assistant/friend that can understand anything. What would you like that assistant/friend to do for you?

Our goal is to collect data in the format "If .... then .... because ...." "

After the data was collected, a team of annotators annotated the commands with additional presumptions that the human subjects have left unspoken. These presumptions were either in the *if*-clause and/or the *then*-clause and examples of them are shown in Tables 1 and 4

Logic Templates

As explained in the main text, we uncovered 5 different logic templates, that reflect humans' reasoning, from the data after data collection. The templates are listed in Table 5. In what follows, we will explain each template in detail using the examples of each template listed in Tab. 5.

In the blue template (Template 1), the `state` results in a "bad state" that causes the *not* of the goal. The speaker asks for the `action` in order to avoid the bad state and achieve the `goal` . For instance, consider the example for the blue template in Table 5. The `state` of snowing a lot at night, will result in a bad state of traffic slowdowns which in turn causes the speaker to be late for work. In order to overcome this bad state. The speaker would like to take the `action` , waking up earlier, to account for the possible slowdowns cause by snow and get to work on time.

In the orange template (Template 2), performing the `action` when the `state` holds allows the speaker to achieve the `goal` and not performing the `action` when the `state` holds prevents the speaker from achieving the `goal` . For instance, in the example for the orange template in Table 5 the speaker would like to know who the attendees of a meeting are when the speaker is walking to that meeting so that the speaker is prepared for the meeting and that if the speaker is not reminded of this, he/she will not be able to properly prepare for the meeting.

In the green template (Template 3), performing the `action` when the `state` holds allows the speaker to take a hidden `action` that enables him/her to achieve the desired `goal` . For example, if the speaker is reminded to buy flower bulbs close to the Fall season, he/she will buy and plant the flowers (hidden `action` s) that allows the speaker to have a pretty spring garden.

In the purple template (Template 4), the `goal` that the speaker has stated is actually a goal that they want to *avoid*. In this case, the `state` causes the speaker's `goal` , but the speaker would like to take the `action` when the `state` holds to achieve the opposite of the `goal` . For the example in Tab. 1, if the speaker has a trip coming up and he/she buys perishables the perishables would go bad. In order for this not to happen, the speaker would like to be reminded not to buy perishables to avoid them going bad while he/she is away.

The rest of the statements are categorized under the "other" category. The majority of these statements contain conjunction in their `state` and are a mix of the above templates. A reasoning engine could potentially benefit from these logic templates when performing reasoning. We provide more detail about this in the Extended Discussion section in the Appendix.

Prolog Background

Prolog (Colmerauer 1990) is a declarative logic programming language. A Prolog program consists of a set of predicates. A predicate has a name (functor) and $N \geq 0$ arguments. N is referred to as the arity of the predicate. A predicate with functor name F and arity N is represented as $F(T_1, \ldots, T_N)$ where T_i's, for $i \in [1, N]$, are the arguments that are arbitrary Prolog terms. A Prolog term is either an atom, a variable or a compound term (a predicate with arguments). A variable starts with a capital letter (e.g., Time) and atoms start with small letters (e.g. monday). A predicate defines a relationship between its arguments. For example, isBefore(monday, tuesday) indicates that the relationship between Monday and Tuesday is that, the former is before the latter.

A predicate is defined by a set of clauses. A clause is either a Prolog *fact* or a Prolog *rule*. A Prolog rule is denoted with Head :− Body., where the Head is a predicate, the Body is a conjunction ($\wedge$) of predicates, :− is logical implication, and period indicates the end of the clause. The previous rule is an if-then statement that reads "*if* the Body holds *then* the Head holds". A fact is a rule whose body always holds, and is indicated by Head. , which is equivalent to Head :− true. Rows 1-4 in Table 6 are rules and rows 5-8 are facts.

Prolog can be used to logically "prove" whether a specific query holds or not (For example, to prove that isAfter(wednesday,thursday)? is false or that status(i, dry, tuesday)? is true using the Program in Table 6). The proof is performed through *backward chaining*, which is a backtracking algorithm that usually employs a depth-first search strategy implemented recursively. In each step of the recursion, the input is a query (goal) to prove and the output is the proof's success/failure. in order to prove a query, a rule or fact whose head *unifies* with the query is retrieved from the Prolog program. The proof continues recursively for each predicate in the body of the retrieved rule and succeeds if all the statements in the body of a rule are true. The base case (leaf) is when a fact is retrieved from the program.

Table 4: Example if-then-because commands in the data and their annotations. Annotations are tuples of (index, missing text) where index shows the starting word index of where the missing text should be in the command. Index starts at 0 and is calculated for the original utterance.

Utterance	Annotation
If the temperature ($\cdot^{\downarrow}$) is above 30 degrees ($\cdot^{\downarrow}$) then remind me to put the leftovers from last night into the fridge because I want the leftovers to stay fresh	(2, inside) (7, Celsius)
If it snows ($\cdot^{\downarrow}$) tonight ($\cdot^{\downarrow}$) then wake me up early because I want to arrive to work early	(3, more than two inches) (4, and it is a working day)
If it's going to rain in the afternoon ($\cdot^{\downarrow}$) then remind me to bring an umbrella ($\cdot^{\downarrow}$) because I want to stay dry	(8, when I am outside) (15, before I leave the house)

Table 5: Different reasoning templates of the statements that we uncovered, presumably reflecting how humans logically reason. $\wedge$, $\neg$, :– indicate logical and, negation, and implication, respectively. $action_h$ is an action that is hidden in the main utterance and $action\,(state)$ indicates performing the $action$ when the $state$ holds.

Logic template	**Example**	**Count**
1. $(\neg(\texttt{goal}) :\!\!- \texttt{state}) \wedge$ $(\texttt{goal} :\!\!- \texttt{action}(\texttt{state}))$	If it snows tonight then wake me up early because I want to arrive to work on time	65
2. $(\texttt{goal} :\!\!- \texttt{action}(\texttt{state})) \wedge$ $(\neg(\texttt{goal}) :\!\!- \neg(\texttt{action}(\texttt{state})))$	If I am walking to a meeting then remind me who else is there because I want to be prepared for the meeting	50
3. $(\texttt{goal} :\!\!- \texttt{action}_h) \wedge$ $(\texttt{action}_h :\!\!- \texttt{action}(\texttt{state}))$	If we are approaching Fall then remind me to buy flower bulbs because I want to make sure I have a pretty Spring garden.	17
4. $(\texttt{goal} :\!\!- \texttt{state}) \wedge$ $(\neg(\texttt{goal}) :\!\!- \texttt{action}(\texttt{state}))$	If I am at the grocery store but I have a trip coming up in the next week then remind me not to buy perishables because they will go bad while I am away	5
5. other	If tomorrow is a holiday then ask me if I want to disable or change my alarms because I don't want to wake up early if I don't need to go to work early.	23

At the heart of backward chaining is the *unification* operator, which matches the query with a rule's head. Unification first checks if the functor of the query is the same as the functor of the rule head. If they are the same, unification checks the arguments. If the number of arguments or the arity of the predicates do not match unification fails. Otherwise it iterates through the arguments. For each argument pair, if both are grounded atoms unification succeeds if they are exactly the same grounded atoms. If one is a variable and the other is a grounded atom, unification grounds the variable to the atom and succeeds. If both are variables unification succeeds without any variable grounding. The backwards chaining algorithm and the unification operator is depicted in Figure 3.

Parsing

The goal of our parser is to extract the `state`, `action` and `goal` from the input utterance and convert them to their logical forms $S(X)$, $A(Y)$, and $G(Z)$, respectively. The parser is built using Spacy (Honnibal and Montani 2017). We implement a relation extraction method that uses Spacy's built-in dependency parser. The language model that we used is the en_coref_lg$-3.0.0$ released by Hugging face[8]. The predicate name is typically the sentence verb or the sentence root. The predicate's arguments are the subject, objects, named entities and noun chunks extracted by Spacy. The output of the relation extractor is matched against the knowledge base through rule-based mechanisms including string matching to decide weather the parsed logical form exists in the knowledge base. If a match is found, the parser re-orders the arguments to match the order of the arguments of the predicate retrieved from the knowledge base. This re-ordering is done through a type coercion method. In order to do type coercion, we use

[8]https://github.com/huggingface/neuralcoref-models/releases/download/en_coref_lg-3.0.0/en_coref_lg-3.0.0.tar.gz

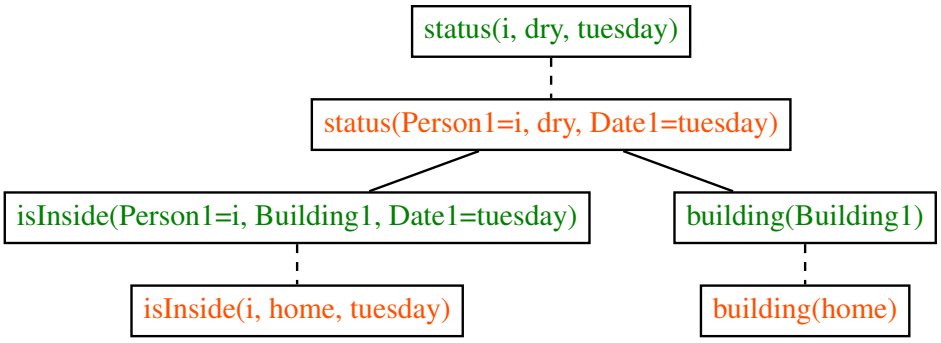

Figure 3: Sample simplified proof tree for query status(i, dry, tuesday). dashed edges show successful unification, orange nodes show the head of the rule or fact that is retrieved by the unification operator in each step and green nodes show the query in each proof step. This proof tree is obtained using the Prolog program or $\mathcal{K}$ shown in Tab. 6. In the first step, unification goes through all the rules and facts in the table and retrieves rule number 2 whose head unifies with the query. This is because the query and the rule head's functor name is status and they both have 3 arguments. Moreover, the arguments all match since Person1 grounds to atom i, grounded atom dry matches in both and variable Date1 grounds to tuesday. In the next step, the proof iterates through the predicates in the rule's body, which are isInside(i, Building1, tuesday) and building(Building1), to recursively prove them one by one using the same strategy. Each of the predicates in the body become the new query to prove and proof succeeds if all the predicates in the body are proved. Note that once the variables are grounded in the head of the rule they are also grounded in the rule's body.

Table 6: Examples of the commonsense rules and facts in $\mathcal{K}$

1	isEarlierThan(Time1,Time2) :- isBefore(Time1,Time3), isEarlierThan(Time3,Time2).
2	status(Person1, dry, Date1) :- isInside(Person1, Building1, Date1), building(Building1).
3	status(Person1, dry, Date1) :- weatherBad(Date1, _), carry(Person1, umbrella, Date1), isOutside(Person1, Date1).
4	notify(Person1, corgi, Action1) :- email(Person1, Action1).
5	isBefore(monday, tuesday).
6	has(house, window).
7	isInside(i, home, tuesday).
8	building(home).

the types released by Allen AI in the Aristo tuple KB v1.03 Mar 2017 Release (Dalvi Mishra, Tandon, and Clark 2017) and have added more entries to it to cover more nouns. The released types file is a dictionary that maps different nouns to their types. For example, doctor is of type person and Tuesday is of type date. If no match is found, the parsed predicate will be kept as is and CORGI tries to evoke relevant rules conversationally from humans in the *user feedback loop* in Figure 1.

We would like to note that we refrained from using a grammar parser, particularly because we want to enable open-domain discussions with the users and save the time required for them to learn the system's language. As a result, the system will learn to adapt to the user's language over time since the background knowledge will be accumulated through user interactions, therefore it will be adapted to that user. A negative effect, however, is that if the parser makes a mistake, error will propagate onto the system's future knowledge. This is an interesting future direction that we are planning to address.

Inference

The inference algorithm for our proposed neuro-symbolic theorem prover is given in Alg. 1. In each step t of the proof, given a query Q, we calculate $\mathbf{q}_t$ and $\mathbf{r}_t$ from the trained model to compute $\mathbf{r}_{t+1}$. Next, we choose k entries of $\mathbf{M}^{rule}$ corresponding to the top k entries of $\mathbf{r}_{t+1}$ as candidates for the next proof trace. k is set to 5 and is a tuning parameter. For each rule in the top k rules, we attempt to do variable/argument unification by computing the cosine similarity between the arguments of Q and the arguments of the rule's head. If all the corresponding pair of arguments in Q and the rule's head have a similarity higher than threshold, $T_1 = 0.9$, unification succeeds, otherwise it fails. If unification succeeds, we move to prove the body of that rule. If not, we move to the next rule.

Extended Discussion

Table 7 shows the performance breakdown with respect to the logic templates in Table 5. Currently, CORGI uses a general theorem prover that can prove all the templates. The large variation in performance indicates that taking into account the different templates would improve the performance. For example, the low performance on the green template is expected, since CORGI currently does not support the extraction of a hidden `action` from the user, and interactions only support extraction of missing `goal` s. This interesting observation indicates that, even within the same benchmark, we

Algorithm 1 Neuro-Symbolic Theorem Prover

Input: goal $G(Z)$, $\mathbf{M^{rule}}$, $\mathbf{M^{var}}$, Model parameters, threshold T_1, T_2, k
Output: Proof P

```
r_0 ← 0                                              ▷ 0 is a vector of 0s
P = PROVE(G(Z), r_0, [])
function PROVE(Q, r_t, stack)
    embed Q using the character RNN to obtain q_t
    input q_t and r_t to the model and compute r_{t+1} (Equation (2))
    compute c_t (Equation (2))
    {R^1 ... R^k} ← From M^{rule} retrieve k entries corresponding to the top k entries of r_{t+1}
    for i ∈ [0, k] do
        SU ← SOFT_UNIFY(Q, head(R^i))
        if SU == False then
            continue to i + 1
        else
            if c_t > T_2 then
                return stack
            add R^i to stack
            PROVE(Body(R^i), r_{t+1}, stack)             ▷ Prove the body of R^i
    return stack
function SOFT_UNIFY(G, H)
    if arity(G) ≠ arity(H) then
        return False
    Use M^{var} to compute cosine similarity S_i for all corresponding variable pairs in G and H
    if S_i > T_1   ∀   i ∈ [0, arity(G)] then
        return True
    else
        return False
```

might need to develop several reasoning strategies to solve reasoning problems. Therefore, even if CORGI adapts a general theorem prover, accounting for logic templates in the conversational knowledge extraction component would allow it to achieve better performance on other templates.

Table 7: Number of successful reasoning tasks vs number of attempts under different scenarios. In CORGI's Oracle unification, soft unification is 100% accurate. LT stands for Logic Template and LTi refers to template i in Table 5.

CORGI	LT1	LT2	LT3	LT5
Oracle Unification	24%	38%	11%	0%