Braid: Weaving Symbolic and Neural Knowledge into Coherent Logical Explanations

Aditya Kalyanpur, Tom Breloff, David Ferrucci,
Adam Lally, John Jantos
Elemental Cognition
New York, NY, USA
{adityak, tomb, davef, adaml, johnj}
@elementalcognition.com

Abstract

Traditional symbolic reasoning engines, while attractive for their precision and explicability, have a few major drawbacks: the use of brittle inference procedures that rely on exact matching (unification) of logical terms, an inability to deal with uncertainty, and the need for a precompiled rule-base of knowledge (the "knowledge acquisition" problem). These issues are particularly severe for the Natural Language Understanding (NLU) task, where we often use implicit background knowledge to understand and reason about text, resort to fuzzy alignment of concepts and relations during reasoning, and constantly deal with ambiguity in representations. To address these issues, we devise a novel FOL-based reasoner, called Braid, that supports probabilistic rules, and uses the notion of custom unification functions and dynamic rule generation to overcome the brittle matching and knowledge-gap problem prevalent in traditional reasoners. In this paper, we describe the reasoning algorithms used in Braid-BC (the backchaining component of Braid), and their implementation in a distributed task-based framework that builds proof/explanation graphs for an input query in a scalable manner. We use a simple QA example from a children's story to motivate Braid-BC's design and explain how the various components work together to produce a coherent logical explanation.

1 Introduction

KR&R systems work well for certain knowledge-rich domains that typically involve a (pre-defined) set of axioms or rules, use structured queries and datasets, and have a need for precise logical inference with explanations. Formal logic-based reasoning engines such as Cyc (Lenat, 1985) and Ergo (Benjamin Grosof and Bloomfield., 2018) have been successfully deployed in domains such as legal, healthcare and finance. One of the main advantages of using such systems is transparency – the underlying reasoning of the system is well-understood and can be justified to end-users.

However, there are several known drawbacks of logic-based approaches. For one, the inference procedures are highly brittle in that they require precise matching/unification of logical terms and formulae in order to construct a complete explanation. Secondly, traditional reasoners don't deal with uncertainty well (often treating rules as hard constraints), whereas rules in real-world applications are often probabilistic and contextual. Thirdly, all such systems suffer from the knowledge acquisition problem (i.e. how does one acquire the rules). Often, the rules are hand-coded, an approach which doesn't scale in general.

Our problem domain is Natural Language Understanding (NLU), an area where all the issues mentioned above come into play – the need to acquire and use implicit background knowledge to understand text, the application of rules differently based on the context, and the use of imperfect/fuzzy alignment of concepts and relations when doing reasoning. To address these issues, we devise a novel FOL-based reasoner, called Braid. Braid includes a backward and forward chainer, assumption based reasoner and a constraint solver. This paper only refers to the backward chaining component, which we refer to as Braid-BC.

Braid-BC supports rules with confidences, and uses the notion of *custom unification functions* and *dynamic rule generation* to overcome the brittle matching and knowledge-gap problem prevalent in traditional reasoning engines. The custom-unifiers can be based on any statistical techniques, as long as they can propose and score mappings between the terms of two logical propositions (the unifier interface is defined more precisely in Section 4.2).

For example, we use neural matching functions as unifiers. Their purpose is to help the reasoner find proofs even when (sub) goals, rule conditions and/or facts do not align perfectly.

The dynamic rule-generator (Section 6) is given a target proposition (goal) and a knowledge base (KB) as input, and outputs a scored list of hypothesized rules that could be used to prove that proposition. The purpose of rule-generation is to connect the dots when the knowledge required for an inference is missing from the static KB. We describe two DRG implementations - one using a neural (transformer-based) rule generation model that was fine-tuned on a dataset of crowd-sourced causal rules, known as GLUCOSE (which leverages our previous work (Mostafazadeh et al., 2020)), and the second that uses a rule-template based technique.

We describe the reasoning algorithms used in Braid-BC, and their implementation in a distributed task-based framework that builds proof/explanation graphs for an input query in a highly scalable manner. Our approach shares some similarities with the RETE framework (Charles, 1982) for matching production rules (e.g. reusing nodes for the same sub-goal proposition, message passing between nodes etc.) but makes several novel extensions: we primarily do backward chaining via a heuristic best-first search (i.e. A* search), leverage a Master-Worker architecture where the Master builds the main proof graph while Workers make local inferential updates, and define general functions for Unifiers and Provers that lets us plug in various reasoning strategies combining standard reasoning (e.g. syntactic resolution based) with statistical approaches (e.g. word-embedding based).

2 Related Work

2.1 Comparison with standard logical reasoners

State-of-the-art logical reasoning systems such as Cyc, Ergo, Pellet (Sirin et al., 2007), Vampire (Kovács and Voronkov, 2013), SPASS (Suda et al., 2009) etc. have a rich set of features beyond what Braid currently supports such as higher order logics (e.g. HiLog (Chen et al., 1993)), more sophisticated identity and negation reasoning, disjunction support and in some cases, explicit defeasible rule theories. However, they do not support rules with confidences, custom unification functions or dynamic rule-generation, features we believe are needed to overcome the brittleness of standard deductive reasoning.

Also, our distributed task-based reasoning framework which can scale to multiple cores/machines on a cluster has several novel aspects. The design shares some similarities with the RETE algorithm, in that propositional atoms in rules are nodes in an inference graph (referred to as alpha nodes in the RETE network), we have join-support nodes similar to beta nodes in the RETE network to compute joint solutions across conjunctions, nodes are shared across rules and are associated with bindings that flow based on the RETE network graph. However, the RETE algorithm is typically used for forward inference, whereas our reasoning paradigm is goal-driven backward chaining. Moreover, we extend the approach to include a heuristic search strategy to deal with confidences associated with fuzzy unifications and rules, support for existential quantifications, and dynamic rules. Our overall design has several benefits: reusing solutions for previously solved goals (also knowns as tabling), minimizing communication overhead by ensuring locality of inferential computations on Workers, only passing new solutions between nodes to limit updates, potential for efficient belief-revision / truth maintenance by storing a global proof support graph across all queries, and updating results incrementally when the KB changes.

2.2 Comparison with statistical relation learning

We contrast Braid with Markov Logic Networks (MLN) (Richardson and Domingos, 2006), a prominent SRL framework. An MLN KB is a collection of FOL rules with weights. The KB itself acts as a template for generating a Markov Network (an undirected graphical model where nodes are random variables and edges represent dependencies between variables) which is created by grounding the variables using the constants in the KB. In the final network, each node (random variable) is a ground literal from a rule in the KB, and each rule forms a clique, that has a weight w. The basic idea is that given a particular configuration of truth values for the variables (nodes), if a rule containing those nodes is satisfied and its weight is high, the probability of the world goes up, and vice versa (i.e. violating a rule which has a high weight causes a big drop in the probability of the world). Thus, rules thus act as soft constraints on the likelihood of the world. Given a KB of true facts along with the weighted rules, we can perform MAP inference to answer FOL queries (i.e. $P(query|data)$.

The main issue with MLNs is scalability due to the grounding step which is exponential in the size of the KB. Additionally, MLN's work with a static KB and do not support the dynamic addition of rules during the inference process.

2.3 Comparison with neuro-symbolic approaches

In recent times, there has been a growing interest in exploring neural-symbolic approaches such as Logic Tensor Networks (LTNs) (Serafini and d'Avila Garcez, 2016), which use distributed representations for logical symbols. For example, in LTNs, each constant is mapped to a real valued vector, each n-ary function is defined as a linear transform over its argument vectors, and predicates and clauses are associated with a 0-1 score (reflecting its truth value), computed by using a tensor network (where the inputs are the argument vectors). The representations are learned by optimizing for approximate satisfiability of the clauses in the KB. We share the same objective with such approaches, namely, overcoming the brittleness of standard reasoning algorithms using statistical methods and fuzzy reasoning.

However, one concern with the LTN model is scalability, as each clause is trained using a separate tensor model. It is also unclear how much training data is needed to learn reliable representations. Finally, it is not clear how such a model can produce precise explanations/proofs for its final decisions, as the reasoning process becomes opaque.

We take an alternate approach to combining neural-symbolic information. Instead of using distributed representations for FOL term/formulae and learning how to compose them using neural models, we use an FOL-resolution styled model as the underlying reasoning paradigm which lets us preserve explicability of the final results, while allowing for statistical methods (e.g. neural models) to be used for fuzzy matching/unifications and hypothesizing missing rules. Also, we believe our implementation based on a distributed task-framework and message passing is more scalable.

A more directly related neuro-symbolic approach is that taken by NL-Prolog (Weber et al., 2019) in which the authors use a Prolog-like system to do back-chaining from a query to find proofs, though the entities/predicates have distributed representations (allowing for weak unification), and the inference rules are learned during training by specializing generic templates like `P1(?X1, ?X3) :- P2(?X1, ?X2), P3(?X2, ?X3).` On the surface, the notions of weak unification and dynamic rule induction in NL-Prolog seem very similar to that in Braid. However, there are some important fundamental differences: NL-Prolog is at its core an E2E differentiable system whose explanations are not fully transparent (e.g. the learned rule predicates have distributed representations and are not directly interpretable). Also, the embeddings for entities/predicates once learned during training are fixed at test time and independent of the local context. Finally, there are various hyperparameter choices such as number of rule template instances to create which can impact the final result and it is unclear how to set these appropriately.

On the other hand, Braid at its core is a symbolic reasoning engine that produces transparent logical explanations but uses distributed representations for doing fuzzy unification and rule hypothesizing/scoring. The entity/predicate embeddings used in fuzzy unification come from a transformer-based deep learning model which looks at the local context in which the entities/predicates appear (since they are derived from words in the text). Also, the dynamically induced rules are fully interpretable (unlike in NL-Prolog), and their confidence scores are estimated via pre-trained language models.

3 Motivating Example

We describe an example to illustrate the challenges that a logical reasoning engine faces when answering questions in an NLU setting. The text below is from a children's story (Grade: K) on the ReadWorks website (http://readworks.org).

Consider the following short story:

"Fernando and Zoey go to a plant sale. They buy mint plants. They like the minty smell of leaves. Zoey puts her plant near a sunny window. The plant looks green and healthy!"

The question we would like to answer is:

"Why does Zoey place the plant near the window?"

Questions of this nature are part of our **Template of Understanding**, defined in our prior work (Dunietz et al., 2020), that we use to test an AI system's deep understanding of a narrative story.

Figure 1 shows the logical interpretation of a sentence in the story. For each sentence, we generate multiple probabilistic interpretations (each associated with a confidence) by running a suite of NLP components on the text, including a syntactic and semantic parser (Kalyanpur et al., 2020), a word sense disambiguator and a co-reference resolution component. We omit details about the NLP stack, as they are not relevant for this paper.

```
Zoey puts her plant near a sunny window
1. put(e2)
2. agent (e2, Zoey)
3. theme(e2, plant)
4. destination(e2, window, near)
5. hasProperty(window, sunny)
```

Figure 1: Logical interpretation of a story sentence

In the figure, *e2* is a constant which denotes an event/action, while *Zoey, plant* etc. are constants derived from story terms. On the other hand, predicates such as *put, agent, theme* etc. come from our lexical ontology called Hector. The Hector ontology is a collection of frames (concepts and relations) derived from FrameNet (Ruppenhofer et al., 2006) and NOAD (Stevenson and Lindberg, 2010) that aims to capture the core meaning behind text.

The question interpretation is shown in Figure 2. We assume that we have run a co-reference algorithm on the question and story text, and so the same constants are used (when co-referential) in the question and story interpretations. The specific question representation (line 5) uses the *motivates* relation and is querying for Zoey's goal which explains her performing the place action.

```
Question: Why does Zoey place the plant near the window?
1. place(e3)
2. agent(e3, Zoey)
3. theme(e3, plant)
4. destination(e3, window, near)
5. ?motivates(Zoey, e3, ?goal)
```

Figure 2: Logical interpretation of question

Answering this seemingly straightforward question requires a lot of implicit background knowledge, such as that a plant near a sunny window gets exposed to light, that plants need light to be healthy, and that Zoey wants the plant to stay alive (which motivated her action). Also, in this particular example, the reasoner needs to realize that the put action (in the story) and the place action (in the question) are similar/synonymous in this context.

In the subsequent sections, we shall see how Braid-BC resolves these issues via dynamic rule generation (to bring in background knowledge) and fuzzy unification (to overcome the verb action mismatch).

4 Braid-BC

In this section, we describe the overall Braid-BC framework – a scalable, parallelized infrastructure for constructing deductive proof/explanation graphs for a given query and KB.

4.1 Background

Braid's backchaining algorithm is based on SLD resolution (Kowalski and Kuehner, 1971), the same inference procedure used in Prolog, with additional modifications to deal with existential quantification, custom unification and dynamic rules.

SLD resolution is a sound and complete inference procedure for Horn logic programs. Given a set of Horn clauses and a goal clause, SLD resolution works by creating a search tree of alternative derivations for the goal. It starts by setting the initial goal clause as the root and selecting a literal `L` in the clause to check a derivation for via backward reasoning. It then selects a definite clause whose positive literal unifies with `L` via variable-substitution `S` and creates a new child node for this clause substituting `S` into each of the remaining literals. It repeats this procedure by treating the child node as a new goal-clause and builds out the search tree. A leaf node is successful if it represents the empty clause. Any linear chain (path) from a successful leaf node to the root constitutes a valid proof for the original goal.

While the above is a conceptual description of how SLD resolution works, practical reasoners like Prolog work by doing a DFS of the search space and doing back-tracking when a reasoning path fails, to try an alternate path at an early choice point in the search tree. Additionally, reasoners use optimizations like memoization (or tabling) to cache and retrieve previously solved sub-goals.

For our use-case, there are several reasons why Prolog doesn't suffice. Our logic support existential quantification which makes the rules non-Horn clauses. The rules have confidences which need to be taken into account when constructing the search space. As a result, we use a best-first-search expansion strategy. Since our goal is to develop a highly scalable reasoner, we explore multiple paths in parallel. This means possibly evaluating antecedents of the same rule in parallel and doing a join operation on the bindings to find a consistent solution set, which is different from the way (standard) Prolog evaluates clauses.

Lastly, as mentioned earlier, our aim is to overcome the brittleness of standard logical reasoning algorithms via features like custom unification and dynamic rule-generation, which is beyond the capabilities of Prolog.

We now describe two core logical functions of Braid-BC: Unifiers and Provers.

4.2 Unifiers

One of the core functions in any FOL-based reasoner is unification. The standard unification function (*syntactic unification*) takes a pair of predicate logic formulae P1, P2, and checks if there exists a mapping of variables from one to the other which makes the two formulae equal. For example, the formulae *hasPossession(Zoey, ?y)* and *hasPossession(?x, plant)* unify with the mapping *[?x=Zoey, ?y=plant]*.

We generalize the notion of unification to be any FOL formulae matching function, defined as follows:

`unify(P1, P2, K)` $\rightarrow$ `{UR1...URn}`

where P1, P2 are predicate logic formulae, K is the knowledge base (which acts as context for the matching) and {UR1..URn} are a set of Unification Results, where each UR contains the following information: a substitution that maps variables, entities *or even the predicate* in one formula to the other; score (0-1) which reflects the confidence of the mapping; and additional metadata used by the unifier function when making its decision (which can be exposed by Braid in the final explanation).

Consider the example described in Section 3, where the story interpretation contains the proposition *put(e1)* while the question interpretation has the proposition *place(e3)*. Under standard unification, both these propositions would not unify, as they use different predicates and arguments. However, we have designed a custom unification function that considers word/phrase similarity (note: *place* and *put* are constants derived from story text) to align the two formulae, using the additional context that they share the same agent (*Zoey*), theme (*plant*) and destination (i.e. *near the window*) to boost the match score. Such a function may return `unify(put(e1), place(e2))` $\rightarrow$ `({put=place, e1=e2}, 0.9)`.

For fuzzy unification, we use a transformer-based neural model (Vaswani et al., 2017) to compute distributed representations for entities/predicates, since they also consider the local context of the words from which the entities/predicates are derived into account.

4.3 Provers

Similar to how SLD resolution works by building out a search tree for a given query/goal, Braid works by constructing a proof graph by using unification methods to backchain on clauses (rules/facts) in the KB.

To support various reasoning algorithms, we define the notion of a Prover, a function which given a (sub) goal and the KB, performs a "single step" expansion of the graph along a particular reasoning path.

`prover(G, K)` $\rightarrow$ `PD`

where G is the input goal, K is the knowledge base, and PD is a partial proof-derivation DAG which has the following properties:

- PD has two types of nodes: goal nodes and support nodes.
- Support nodes provide justification for some goal node and play a key role in the flow of information (e.g. solution bindings) in the overall proof graph, as we shall see later.
- G is the root of the graph
- Goal and support nodes in the graph are interleaved, i.e. a goal node can only have support nodes as its children (and vice versa).

For example, a *Rule-Based Prover* finds rules in the input KB whose respective consequents unify with the goal (using any implementation of the Unification interface defined earlier), and then output a partial proof-tree which has the goal as its root, a "rule-support" child node for each such rule that satisfies the condition above, and an additional edge from each support node to the corresponding antecedent of the rule (which form new sub-goals). Note that such a prover does not need to prove the antecedents.

The main advantage of this design is its scalability for deployment in a distributed setting – each prover performs a local computation, without requiring knowledge about the overall proof-graph, which enables parallelization across cores and machines. Also, the communication between the master Braid algorithm (which constructs the entire proof graph) and each individual prover is kept to a minimum.

4.4 Distributed Proof Graph Builder

Braid-BC uses a task-based framework where a central "Master" task builds the entire proof graph for the input goal by communicating with a set of "Worker" tasks, each of which use provers to perform local graph-based reasoning (see Figure 3 for the architecture diagram).

The master algorithm is described in Figure 4. As such, it is a generic, parallelized, graph building approach that continuously modifies a central graph (in this case, a Braid graph) based on asynchronous updates coming in from (remote) workers.

Nodes in the Braid graph are either goal nodes or support nodes. Each node is associated with a collection of Unification-Results that represent bindings flowing into the node via its child edges. Crucially, atomic goal nodes are reused across the graph (i.e. there is only one node per atomic goal proposition in the entire graph) which allows us to reuse solutions found earlier for the same goal proposition (caching).

Goal nodes also have a State, which can have one of three values: `Success, Failure and Unknown`. Also, each support node is associated with a Prover which specifies the logic for handling new binding updates coming into that nodes. Finally, since adjoining nodes may have propositions involving different variables, edges containing nodes are associated with variable-variable mappings.

The key functions of the master algorithm are merging local graph changes into the central master graph (Function: `mergeUpdate`) and determining the next goal to focus on and expand as part of the reasoning process (Function: `getNextSubGoal`)

- `mergeUpdate` takes the next graph update coming from the worker and merges its contents into the main graph. The event might be ignored (if we have already merged an equivalent event) or it may be changed to maintain internal consistency. For example, if we are merging a new edge from the local graph sent by the worker, and there is an equivalent destination node in the graph, we may connect the new source node to the existing destination node with a newly constructed edge which represents the proper mapping between nodes.
- `getNextSubGoal` uses a Goal Selection Strategy to determine which goal node(s) in the graph to expand next. When a worker becomes available, the strategy selects the next-best goal to expand and sends this goal, along with pertinent expansion parameters, to the worker. The default ordering strategy uses the following features: an aggregation of the product of confidences of the supports in the path from the node to a query, the minimum path distance from the node to a query, a measure of goal complexity (computed using syntactic heuristics such as the number of nested formula in the goal), and a measure of goal "plausibility" (estimated by using a statistical language model to score text plausibility on the textual version of the goal proposition).

Algorithm: Full Proof Graph Builder (runs on Master)
Input: Query Q, KB K
Output: Proof Graph PG

```
 1. PG = {Q}
 2. graphUpdates = [] // subscribe to workers which send local updates
 3. propagations = [] // queue up propagations
 4. done = False

 5. while (!done)

      // process all queued graph updates
 6.   while ((u = next(graphUpdates)) != null)
 7.      for (node in mergeUpdate(u))
 8.        if (changed(node))
 9.          p = propagation(node)
10.          push(propagations, p)

      // check if workers are available, if so, send work
11.   if (workerAvailable)
12.     sg = getNextSubgoal(PG) // use custom reasoning strategy
13.     if (sg == null)
14.       done = True
15.     else
16.       sendWork(sg)

      // process queued propagations
17.   while ((p = next(propagation)) != null)
18.     propagate(p)
```

Function: Propagate
Input: p is a triple: (Node n, (new) State s, (new) UnificationResults U)

```
 1. for parent in parents(n)
 2.   if isGoal(parent)
        update(parent, s, U) // update parent node's state and unification results
 3.   else
        prover = proverFor(parent)
 4.     prover.propagateToProver(n, parent, s, U)
```

Figure 4: Master Algorithm that builds the full proof graph for a given query Q

4.5 Propagation

The propagate event is used to update the node-states of parent nodes when the node-state of a child node is changed. The event is a triple con-

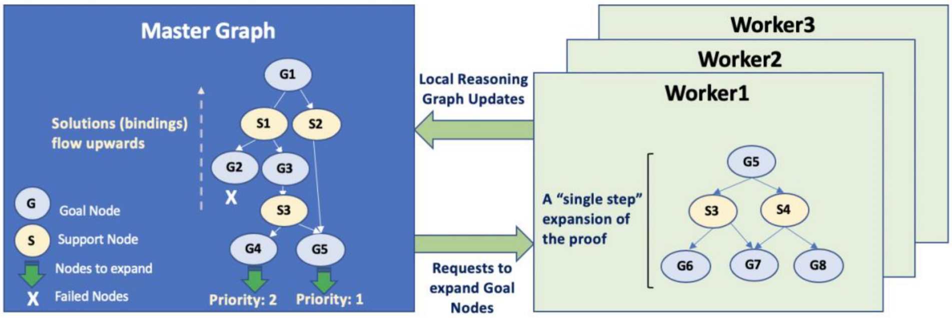

Figure 3: Distributed Proof Graph Builder

taining the triggering node, its new state, and any new unification results on it. Upon propagation, many nodes (e.g. goal nodes) will have a simple update function which creates a new unification result for the parent node using the child's results (also considering the variable-variable mapping on the edge linking the nodes). Some prover support nodes require more complex propagation logic, and the master will trigger this logic on a propagation to that support node (see an example of this in the Agentful-Action Prover in Section 5.2).

4.6 Workers

The Master graph builder is supported by a set of Workers which perform local graph updates. Each Worker is configured with a set of Provers and responds to incoming work requests from the Master by running the provers on the incoming goal. The worker algorithm is quite straightforward and is described in Figure 5.

Algorithm: Incremental Graph Update (Worker)
Input: K (facts + rules)

```
while ((sg = recvWork()) != null)
  for prover in provers
    if prover.canProve(sg) // prover says if it can handle the goal sg
      G = prover.prove(sg, K) // Prover function as defined in section 4.2
      sendGraphUpdates(G) // sends them to master
```

Figure 5: Worker Algorithm that uses provers to perform local incremental graph updates

5 Reasoning Algorithms in Braid-BC

We now describe two reasoning algorithms that have been implemented as Provers in Braid-BC: a default reasoner that extends SLD and can be used to explain any query, and a specialized reasoner for explaining why agents carry out certain actions in a given story.

5.1 Default Backchaining Prover: SLD+

The default back-chainer in Braid is based on SLD resolution described in Section 3 with a few important extensions (hence the name SLD+). The algorithm is described in Figure 6.

SLD+ Prover
(Configured with an aggregate collection of Unifier Functions AU, and a Rule-Generator RG)

```
// Given goal G and KB K, returns a partial derivation graph PD
prove(G, K)
 1. PD = {G}
    // unify goal with facts in the KB
 2. for (f in facts(K))
 3.   for (UR in AU.unify(G, f))) // use configured unifier functions
 4.     FS = new fact-support(F, UR)
 5.     PD.expand(G --> FS)
    // process rules in KB, including dynamically generated rules
 6. for (R in union(rules(K), RG.dynamicRules(G, K)))
 7.   if (AU.unify(G, consequentOf(R))
 8.      RS = new rule-support(R)
 9.      ant = antecedentOf(RS)
 10.     PD.expand(G --> RS --> ant)
         // handle existentials in consequent of rule
 11.     for (c in existentiallyQuantifiedPropsConsequentOf(r))
 12.       PD.expand(c --> RS)
    // split the conjuncts of a conjunction; add a JOIN support node in between
 13. if (G is a1 ^ a2..^ an) // conjunction
 14.   JS = join-support(G)
 15.   PD.expand(G --> JS --> a1..
                               \--> an)
 16. return PD

// Method called when new bindings propagate in Master
propagateToProver(n, parent, s, U)
 17. if (isRule(parent))
 18.   update(parent, s, U) // update parent node's state and unification results
 19. elif (isJoin(parent))
       // perform standard DB join operation on bindings, but use unification as fallback
 20.   state, unifs = doJoin(children(parent))
 21.   update(parent, state, unifs)

// can handle any query/goal
canProve(G)
  return true
```

Figure 6: Default Prover based on SLD resolution with extensions

Like SLD resolution, it tries to resolve the goal by unifying it with clauses (facts/rules) in the KB. Unlike standard SLD, however, it uses custom unifier functions when performing the unification (/matching) in three separate places: when matching the goal against facts in the KB (line 3), when unifying against the consequents of rules (line 7), and in the join operation when bindings are propagated (line 18) as a fallback if the standard DB join operation fails. As noted earlier, the unifiers return Unification-Results with a matching score which is used by Braid to score and rank the solutions and proofs for the input query.

Additional differences from standard implementations of SLD resolution include: the algorithm splits conjunctive goals into individual conjuncts which can be evaluated in parallel (lines 13-15); it uses a Rule-generator to dynamically generate rules for the given goal (line 6); and it creates additional nodes for rule consequents containing existentially quantified variables (lines 11-12), in order to forward propagate skolemized inferences.

5.2 "Agentful" Action Prover

For the example in Section 3, we need to explain the action of Zoey placing her plant near the sunny window. We refer to such actions as Agentful actions (the agent being Zoey in this case).

One reasoning strategy for explaining Agentful actions is the following: first find the motivations of the agent, and then check if the action carried out by the agent leads to one of the agent's objective.

This reasoning strategy can be described by the following logical rule:

`motivates(?agent, ?action, ?goal) :- hasGoal(?agent, ?goal), leadsTo(?action, ?goal)`

The rule uses the predicate `leadsTo` which has the following operational semantics: find a proof for the goal and check if the proof contains the specified action and at least one cause-effect rule. If so, we can conclude that the action causally leads to the goal.

The default SLD+ prover does not have support for handling specialized predicates like leadsTo. Moreover, for efficiency sake, the antecedents of the rule should be evaluated in the order specified above – i.e. first, find the goals of the agent, and then check if the action leads to any one of the bound goal propositions. Specifying a particular ordering for processing rule-antecedents is also unsupported by the SLD+ prover.

As a result, we define a specialized prover to enact the rule above – the Agentful Action Prover, whose algorithm is described in Figure 7.

Agentful Action (AA) Prover

```
prove(G, K)
 1. agent = agentOf(G)
 2. action = actionOf(G)
 3. expand(G --> AAP-support(action) --> hasGoal(agent, ?goal))

propagateToProver(node, AAP-support, state, UR)
 4. if (isAgentGoal(node)) // Node created by line 3. Try to prove action leads to bound goal
 5.   expand(AAP-support --> leadsTo(action, goal))
 6. elif (isLeadsTo(node)) // Node created by line 5. Try to confirm that proof is causal
 7.   for (proof in proofsOf(leadsTo))
 8.     if (isCausal(proof))
 9.       update(parent, SUCCESS, {})

canProve(G)
 return isActionMotivation(G)
```

Figure 7: Prover to explain the rationale for Agentful action

The prove step begins the search for the agent's goals (lines 1-3), and when bound agent-goal solutions are propagated back to the prover, we try to prove whether the action in the original query can lead to the bound goal (line 5). Eventually, when solutions for the latter are propagated back to the prover at a later point, we check if the proof contains a cause-effect rule (lines 7-9), so that we can conclude that the action causally leads to the goal and complete the entire proof.

6 Dynamic Rule Generator (DRG)

As noted in the introduction, the idea behind dynamic rule-generation is to provide missing rule knowledge on the fly to the reasoning engine, where the rules can come from an external function (the rule generator). The generic DRG interface has one core method, which given a target goal and KB, returns rules relevant to proving the goal.

We now describe two DRG implementations that we have developed.

6.1 Fine-tuning Pre-trained Transformers on Curated Rule Knowledge

In prior work, (Mostafazadeh et al., 2020), we had crowd-sourced a dataset of common-sense explanatory knowledge called GLUCOSE. The GLUCOSE dataset consists of both general and specific semi-structured inference rules that apply to short children's stories. The rules are collected along ten dimensions of causal explanations, focusing on events, states, motivations, emotions, and naive psychology. We had shown that by fine-tuning pre-trained transformer models like T5 (Raffel et al., 2019) on the GLUCOSE data, the resultant neural generative model was able to produce contextual common sense inference rules on unseen stories with surprisingly high accuracy.

We use the GLUCOSE trained model as one

Figure 8: GLUCOSE suggestion for a story about soccer.

of our DRG implementations to dynamically produce unstructured textual rules. Since Braid-BC is a symbolic reasoning engine, we need to convert the unstructured rules into a structured logical form, and for this we use our state-of-the-art semantic parsing system known as Spindle (Kalyanpur et al., 2020). Also, the crowd-sourced rules were collected in a semi-structured form (using subject-verb-object templates), and hence the rule expression text is regularized, which makes the parsing task easier. Example inferences made by the Glucose rule generator on a children's story are shown in Figure 8.

6.2 Template-based Approach

Our second DRG implementation generates grounded rules from highly general rule patterns/templates that have been successful in the past.

Consider the two rule-templates in Figure 9. The first template captures the notion that someone who possesses an object wants that object to be in a particular state. For example, a child who possesses a toy wants it to be functional. We may have learned this specific rule in some prior story-understanding task, and the system (over) generalizes the rule, in order to create the template, by replacing all the constants in the rule (e.g. the specific name of the child or toy) with variables to create the template. However, we store the type information for the variables – Child, Toy, Functional are all types in our type system (derived from WordNet) – along with the template in our Rule-Template-KB. When the template over-fires and produces an incorrect rule application in a different context, we also store the negative typed bindings in the rule KB as well.

General Rule Template	Prior + Rule Application	Specific rule for Plants story
hasGoal(?agent, hasState(?object, ?state)) :- hasPossession(?agent, ?object)	hasGoal(?Child, hasState(?Toy, Functional) :- hasPossession(?Child, ?Toy)	hasGoal(Zoey, hasState(plant, Healthy)) :- hasPossession(Zoey, plant) [0.8]
hasState(?object, ?state) :- contact(?object, ?theme, ?proximity, ?degree)	hasState(?Battery, Damaged) :- contact(?Battery, ?Heat, Ambient, High)	hasState(plant, Healthy) :- contact(plant, light, Ambient, High) [0.75]

Figure 9: Generic Rule Templates and their specific applications

This Rule-template KB containing general rule templates with positive and negative rule bindings (captured as types) is given to our Template-based-DRG implementation along with the current goal (as the query) and KB (as the context). DRG's task is to produce KB-specific rules for the given goal, by specializing the general templates against the KB. The last column of Figure 9 shows story-specific rules for the example story in Section 3.

For example, the specific rule in the last column of row 1 is generated by grounding the variables in the corresponding rule-template (i.e. ?agent, ?object, ?state) using constants in the KB (e.g. Zoey, plant, Healthy) and then scoring the rule (in the example, 0.8) using variety of heuristics that check syntactic and semantic well-formedness, similarity to prior solutions, and plausibility from a background corpus. The DRG would generate this specific rule when given a goal/query such as hasGoal(Zoey, ?goal), as the rule helps satisfy this goal.

7 Proof Finding using Integer Linear Programming (ILP)

Once the Braid-BC Master has built a proof-graph for an input query, the final step is to extract proof-trees (explanations) for the query answers, provided the status of the original goal (query) node is SUCCESS. Note that there could be multiple solutions for a given query, and each of those solutions could have multiple proof-trees.

Since rules used in Braid are associated with confidences, and some facts may be brought in via a fuzzy unification (with an associated confidence), we need to take these confidences into account when scoring proofs and find a proof that maximizes the overall confidence of the rules/facts used in it.

We frame proof-extraction as an Integer Linear Programming (ILP) problem as follows: we create binary variables that represent the inclusion/exclusion of each node in the graph. As mentioned earlier, there are two types of nodes in the graph – goal nodes (e.g. original query, any inference from a rule, or fact in the KB) and support nodes linked to provers (e.g. rule).

The ILP program has the following constraints

- The query/solution goal node must be included
- If a goal node is included, exactly one of its support nodes must be included
- If a support node is included, all its antecedent (goal) nodes must be included

Along with the following objective function: maximize the product of confidences of all included nodes.

Each solution to the ILP problem constitutes a single proof-tree to the input query, and the top solution has the highest overall confidence. Note that the objective function can be modified to consider other ways (besides product, which assumes independence) to aggregate the confidences of the individual rules/facts.

8 Putting it all together

Braid-BC solves the motivating example in Section 3 by generating the proof tree shown in Figure 10.

It starts by using the Agentful Action Prover (AAP) to backchain on the rule R1 described in Section 5.2, which first looks for Zoey's goals and then checks if the "place" action (e3) performed by Zoey can lead to one of her goals (line 1). The subproofs use rules to infer that Zoey wants her plant to be healthy (R2), and that the plant being in contact with light causes it to become healthy (R4). R2 and R4 are generated by both the GLUCOSE-based and template-based DRG (the reasoner chooses the higher of the two rule confidences). Braid-BC also uses a dynamically generated rule (from the template-based DRG) about ambient properties being transferred between nearby objects (R5). Note that rules R3, R6 and R7 are not dynamically generated but come from the semantics of "buy", "put" and "sunny" respectively, as defined in the Hector ontology. Finally, the reasoner uses a fuzzy unification function to realize that put and place are similar actions in this context (as described in Section 4.2).

Our larger QA system built on top of Braid-BC has a Natural Language Generation (NLG) module which generates a more consumable explanation for end-users from the logical proof tree (also shown in the figure).

9 Conclusions and Next Steps

It is our belief that a reasoner for the NLU problem needs to overcome the brittle matching and knowledge-gap problem prevalent in traditional KR&R engines. It would also be remiss to not leverage statistical models, especially given the breakthroughs in deep-learning approaches to NLP. With this in mind, we have designed Braid-BC, a novel FOL-based reasoner that combines symbolic reasoning (based on SLD resolution) with statistical functions (e.g. for fuzzy unification, dynamic rule generation).

Our next step is to evaluate the reasoner on a large set of NLU problems and demonstrate its effectiveness in question answering and explanation generation. For a detailed example, see the **Appendix: Using Braid-BC for the Story Cloze Test.**

9.1 Going beyond the backward-chainer

The larger Braid reasoning framework combines Braid-BC with forward chaining and constraint solving.

In a nutshell, this works as follows[1]: Braid builds a "working memory" structure, which initially contains a set of facts as well as a set of possible assumptions that are not necessarily true.

[1] More details are in a forthcoming publication.

Logical Proof

1. motivates(Zoey, e3, hasState(plant, Healthy)) :- hasGoal(Zoey, hasState(plant, Healthy)) ^ leadsTo(e3, hasState(plant, Healthy)) **[R1]**
2. |_hasGoal(Zoey, hasState(plant, Healthy)) :- hasPossession(Zoey, plant) [0.8] **[R2]**
3. |_hasPossession(Zoey, plant) :- buy(e1) ^ buyer(e1, Zoey) ^ theme(e1, plant) [1.0] **[R3]**
4. |_ buy(e1) ^ buyer(e1, Zoey) ^ theme(e1, plant) **[FACTS]**
5. |_hasState(plant, Healthy) :- contact(plant, light, ambient, high) [0.65] **[R4]**
6. |_contact(plant, light, ambient, high) :- contact(window, light, ambient, high) ^ near(plant, window) [0.7]**[R5]**
7. |_ contact(window, light, ambient, high) **[FACT]**
8. |_ near(plant, window)) :- put(e2) ^ theme(e2, plant) ^ destination(e2, window, near)[1.0] **[R6]**
9. |_ put(e2) ^ theme(e2, plant) ^ destination(e2, window, near) **[FACTS]**
10. |_contact(window, light, ambient, high) :- hasProperty(window, sunny) [1.0] **[R7]**
11. |_put(e2) ~ place(e3) [0.9] **[FUZZY UNIFICATION]**

NLG for explanation

Zoey possesses a plant as a result of buying it. Because Zoey possesses a plant, she wants the plant to be healthy. This motivates her to move the plant to the window.

Moving the plant to the window leads to it being healthy for the following reason(s): After she moves the plant, it is near to the window. The window is in contact with sunlight, which is ambient. As a result, the plant is in contact with sunlight, so it becomes healthy.

Figure 10: **Explanation for: Why does Zoey place her plant near the window?** Rule R1 is the generic pattern captured in the Agentful-Action prover; R2, R4, R5 are dynamically generated; while R3, R6 and R7 fall out from the semantics of the underyling lexical ontology Hector. Note that FACTS in the proof, which are generated by the NL interpretation components, are also associated with confidences (not shown above) that are factored into the final explanation score.

For example, in an NLU application, a sentence might have multiple possible interpretations, only one of which is correct. Forward-direction rules apply to these to derive new inferences, which may also have uncertain truth values. Braid-BC uses the working memory as its knowledge base, and applies backward-direction rules in response to user queries. Completed backward proofs are then added to the working memory. `Contradiction` rules can also be added; these indicate that a set of inferences cannot be true together in a consistent interpretation.

A constraint solver operates on the working memory. If the user asserts that some assumptions or inferences are true, constraint propagation can determine the truth or falsity of other parts of the working memory. It is also possible to define a cost function and then to find the lowest-cost truth assignment to the possible assumptions. This approach of applying rules to build a "ground program" and then applying a constraint solver is influenced by Answer Set Programming (ASP) (Lifschitz and Lifschitz, 2008).

Further features of Braid are that it can help the user diagnose inconsistent KB states, and that it is incremental, so that it can quickly respond to the addition/removal of facts or rules.

9.2 Learning with Braid

Braid as described so far does not have a learning mechanism or a training routine. This is because we do not have ground truth for complete explanations to questions. However, there is a natural extension to the system to collect such training data in an online manner and continuously learn and improve Braid.

The larger vision involves using Braid as part of a QA system which interacts with end-users, showing answers/explanations for questions. The end-user provides direct feedback on the correctness of explanations, and moreover, when an explanation is wrong, can pinpoint which rules or facts (including the fuzzy matched ones) are incorrect.

When an explanation is incorrect, there are only 3 possible reasons

- One or more of the rules used in the proof is incorrect – this is training data to improve the dynamic rule generators (Section 6)
- One or more of the fuzzy unification results is incorrect – this is training data to improve the

custom unifier functions (Section 4.2)

- One or more of the facts (interpretations) is incorrect – this is training data to improve the components in NLP stack which produce the semantic interpretations (from text) as the input KB to Braid. While this is beyond the scope of Braid, it is still valuable information to the NLU system overall.

When an explanation is correct, we get positive training data on all the rules and fuzzy unification results used in the proof.

Our plan is to deploy Braid in this interactive QA system and check how its performance accuracy changes over time as we collect more user-data and train the system.

10 Acknowledgments

We would like to thank David Buchanan for his work on the template-based dynamic rule generator, and Marcello Balduccini and Jesse Dunietz for their valuable feedback and comments.

References

Paul Fodor Benjamin Grosof, Michael Kifer and Janine Bloomfield. 2018. Rulelog: Highly expressive, yet scalable, semantic rules. In *16th International Conference on Principles of Knowledge Representation and Reasoning (KR-2018) - Conference tutorial*.

Forgy Charles. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence*, pages 17–37.

Jiaao Chen, Jianshu Chen, and Zhou Yu. 2019. Incorporating structured commonsense knowledge in story completion. Technical report, AAAI.

Weidong Chen, Michael Kifer, and David S. Warren. 1993. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230.

Jesse Dunietz, Greg Burnham, Akash Bharadwaj, Owen Rambow, Jennifer Chu-Carroll, and Dave Ferrucci. 2020. To test machine comprehension, start by defining comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7839–7859, Online. Association for Computational Linguistics.

Aditya Kalyanpur, Or Biran, Tom Breloff, Jennifer Chu-Carroll, Ariel Diertani, Owen Rambow, and Mark Sammons. 2020. Open-domain frame semantic parsing using transformers.

Laura Kovács and Andrei Voronkov. 2013. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification (CAV 2013)*.

Robert Kowalski and Donald Kuehner. 1971. Linear resolution with selection function. *Artificial Intelligence*, 2:227–60.

Prakash M. Shepherd M. Lenat, D. 1985. Cyc: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *AI Magazine*, 6(4):65–85.

Zhongyang Li, Xiao Ding, and Ting Liu. 2019. Story ending prediction by transferable bert.

Vladimir Lifschitz and Vladimir Lifschitz. 2008. What is answer set programming? In *Proceedings of the 23rd national conference on Artificial intelligence*.

Marvin Minsky. 1974. A framework for representing knowledge. *MIT Technical Report*.

Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. 2016. A corpus and cloze evaluation for deeper understanding of commonsense stories. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 839–849, San Diego, California. Association for Computational Linguistics.

Nasrin Mostafazadeh, Aditya Kalyanpur, Lori Moon, Lauren Berkowitz David Buchanan, Or Biran, and Jennifer Chu-Carroll. 2020. Glucose: Generalized and contextualized story explanations. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. Technical report, OpenAI.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*.

Matthew Richardson and Pedro Domingos. 2006. Markov logic networks. In *Machine Learning*.

Josef Ruppenhofer, Michael Ellsworth, Miriam R.L. Petruck, Christopher R. Johnson, and Jan Scheffczyk. 2006. *FrameNet II: Extended Theory and Practice*. International Computer Science Institute, Berkeley, California. Distributed with the FrameNet data.

Roger C Schank and Robert P Abelson. 1975. Scripts, plans, and knowledge. *Proceedings of the 4th international joint conference on Artificial intelligence*, 1:151–157.

Luciano Serafini and Artur S. d'Avila Garcez. 2016. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *In Proceedings of AI*IA*, page 334–348.

Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. 2007. Pellet: A practical owl-dl reasoner. *Web Semantics*, 5(2):51–53.

Angus Stevenson and Christine A. Lindberg, editors. 2010. *New Oxford American Dictionary (3 ed.)*. Oxford University Press. Electronic Version NOAD3A+2019_G1.

Martin Suda, Geoff Sutcliffe, Patrick Wischnewski, Manuel Lamotte-Schubert, and Gerard de Melo. 2009. External sources of axioms in automated theorem proving. In *KI 2009: Advances in Artificial Intelligence*, pages 281–288, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Leon Weber, Pasquale Minervini, Jannes Münchmeyer, Ulf Leser, and Tim Rocktäschel. 2019. NLProlog: Reasoning with weak unification for question answering in natural language. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6151–6161, Florence, Italy. Association for Computational Linguistics.

Mantong Zhou, Minlie Huang, and Xiaoyan Zhu. 2019. Story ending selection by finding hints from pairwise candidate endings. In *TASLP*.

11 Appendix: Using Braid-BC for the Story Cloze Test

To demonstrate the effectiveness of using explicit semantic representations and dynamic rule generation for an NLP reasoning task, we used Braid to tackle the ROC Story Cloze Test (Mostafazadeh et al., 2016).

The Story Cloze task is as follows: given a 4 sentence story, and two possible endings, pick the more plausible story ending (one that fits better in the context of the story). Examples are shown in Table 1. For our experiments, we focus on the Spring 2016 ROC dataset, which has a validation set and a test set of 1871 examples each.

This task relies on commonsense knowledge, which is typically hard to acquire and represent explicitly. Nevertheless, recent advances in transformer-based language models, such as GPT2 and BERT, have made it possible to build E2E neural solutions that do very well (accuracy in the high 80s) on this task.

Story	**Right Ending**	**Wrong Ending**
Rick grew up in a troubled household. He never found good support in family, and turned to gangs. It wasn't long before Rick got shot in a robbery. The incident caused him to turn a new leaf.	He is happy now	He joined a gang
Ignacio wants to play a sport while he is in college. Since he was a good swimmer, he decides to try out for swim the team. Ignacio makes it onto the team easily. At the first swim meet, Ignacio wins second place!	Ignacio gave up swimming.	Ignacio won a silver medal.
Nya had been asked on a paintball trip with friends. She was nervous about going. But she went anyways, hoping to have fun. She shot paintballs at her friends and laughed the whole time.	She loved it so much she planned a trip for the next week.	She was shot and vowed to never go there again.

Table 1: ROC Story Cloze Test Examples

A common downside with these approaches is the lack of explicability. Instead of a pure neural approach, we developed a hybrid neuro-symbolic solution to this problem using Braid, one that is capable of giving an explanation for choosing a particular story ending.

Our approach is based on the notion of Minsky's frames (Minsky, 1974) or Schank's scripts (Schank and Abelson, 1975), and assumes that each of the stories involves one or more frames or situations, and that information in the story is consistent with the frames. For example, the first row in Table 1 might correspond to the frame *lesson-learned* since Rick realizes the folly of his ways; the second to the frame *made-plan-executed* as Ignacio has a goal in mind and works hard to achieve it; while the third row might correspond to the frame *change-belief-happy* where Nya was initially nervous about a trip that she eventually enjoyed.

Our hypothesis is that if we can correctly detect the applicable frame from the first 4 sentences of the story, we should be able to predict the right ending that is consistent with the detected frame. Furthermore, the frame provides an explanation for the chosen ending.

11.1 Frame Inference

We decompose the Story Cloze problem into two steps:

1. Detect a frame given the first four sentences of the story, i.e. estimate $Pr(Frame|Story)$
2. For each possible ending, compute the probability of it being the ending of the story considering the detected frame, i.e. $Pr(Ending|Story, Frame)$

Our first challenge was to define a collection of relevant frames for the ROC stories. We analyzed a random sample of 350 story examples from the validation set (approximately 1/5th of the entire set) and manually labeled each story with a frame. Each frame is a generic description of the situation unfolding in the story. The process of frame creation was iterative but admittedly subjective - we generalized concepts that seemed too specific, or specialized highly generic concepts to capture sentiment/emotion. In all, we created a set of 19 frames. Examples include: *met expectations (happy/sad)*, *surprise (pleasant/unpleasant)*, *resolving problem*, *persistence (worked/failed)*, *ran into accident* etc.

We used this annotated set to train a classifier using a pre-trained encoder-decoder model (T5-base), which given the first four sentences of the story as input, predicts the corresponding frame. We used a 90/10 split for train/dev and found that the classifier accuracy to be 87.5% on the dev set. The high accuracy with relatively little training data is not surprising given the power of pre-trained transformers and the nature of this task.

We then applied the trained frame detection classifier to the entire validation set in order to label the frames for all the 1871 stories in it. This gave us a "bronze" dataset of frame labels for each story.

The bronze frame labels were used to create larger training sets (from the entire validation set) for the two steps described above, i.e., classifying a frame given a story, and classifying whether a given ending to a story is consistent with the frame. Moreover, we created two different train sets - one using raw text, and the other using semantic parses, obtained by running our SOTA PropBank/Hector parser (Kalyanpur et al., 2020) on each of the story and ending sentences. The training data format for the second step was *input*: `story SEP ending SEP frame`, *output*: `1/0` for right/wrong story ending (SEP stands for a separator token). Finally, we built neural models (again, fine-tuning the T5-base encoder decoder on the train set) using the above training data.

The models from the two steps are used in neural dynamic rule generators (DRGs): the first which produces rules of the form: `frame :- story`, while the second generates `ending :- story, frame` rules. The rules are associated with confidences coming from the corresponding classifier.

11.2 Experiments

We ran the following three experiments:

1. **Baseline**: Our baseline system is an E2E neural model. We train a binary classifier (fine-tuning T5-base) on the validation set to predict whether a given ending to a story is correct. At test time, we apply the classifier to both ending choices and select the ending with the higher classification score.
2. **Frame-Inference-via-Text**: Use the textual versions of the two neural DRGs described in the previous section
3. **Frame-Inference-via-Semantic-Parsing**: Use the semantic-parse versions of the two neural DRGs described in the previous section

Experiments 2 and 3 use Braid-BC to solve the problem. We initialize a knowledge-base consisting of the 4 sentence story, represented either as textual propositions (Expt. 2), or semantic parse propositions (Expt. 3).

Model	Accuracy
E2E Neural Baseline	86.15%
Braid-BC: Frame Inf (Text)	87.17%
Braid-BC: Frame Inf (Sem Parse)	**87.76%**
HintNet (Zhou et al., 2019)	79.2%
GPT2 (Radford et al., 2019)	86.5%
ISCK (Chen et al., 2019)	87.6%
BERT-base + MNLI (Li et al., 2019)	90.6%

Table 2: ROC Story Cloze (Spring 2016) Test Results

We use a two stage prover (similar to the Agentful-Action Prover described in Section 5.2) to solve the story cloze task: the prover first issues an open variable frame query (`frame(?X)`) to the frame-detection-DRG (which uses the story KB as context) to infer potential frames. These frame inferences are added to the KB via the generated

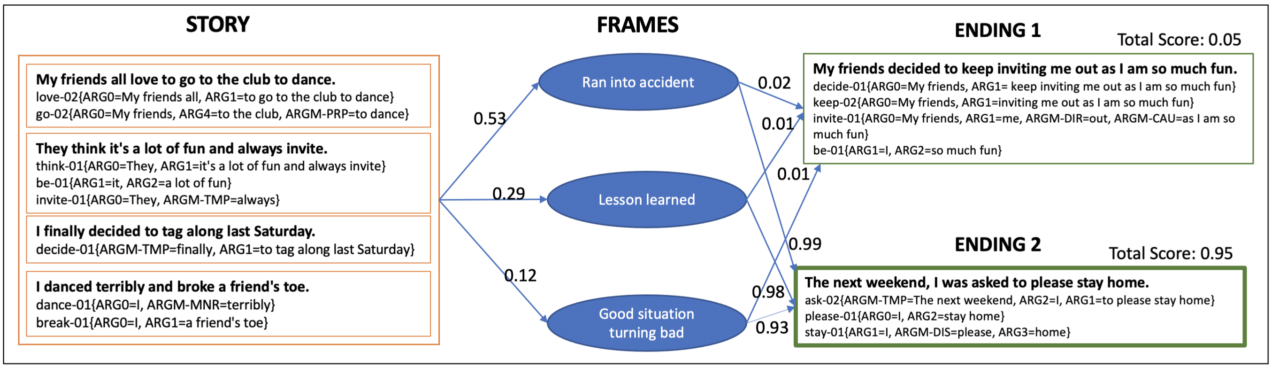

Figure 11: **Example of Frame-Based Explanation for a ROC Story Ending.** The figure shows the inference paths found by Braid-BC using the neural classifiers (for detecting frames given story, and ending given frame+story), along with the corresponding rule scores. The semantic interpretations are shown below each sentence. Only the top-3 frames predicted for the story are shown in the figure (Note: Final scores consider the entire frame distribution).

rules. Then, the prover issues a query for each ending choice to the second DRG, which uses the story and frames as context to produce rules deriving the ending. Lastly, Braid-BC searches for all proofs for each of the endings (in this case, each proof is a linear chain back from the ending to the frame and then to the story), and then uses an aggregate proof scoring function which simply marginalizes across all the intermediate frame propositions.

Results of the experiments are shown in Table 2. The table also includes the performance scores from other SOTA systems on this task, as a point of comparison. We can see that Braid-BC is highly competitive with the SOTA systems while providing frame-based explanations (see Figure 11 for an example). Also, note that the frame inference experiments were done with *bronze* frame labeled data, by manually annotating a small subset of the validation set, so there is clear headroom for improvement.

While the above solution does not use deep or complex logical reasoning, it shows how Braid can be customized for a simple rule-chaining task considering statistical rule generators and achieve close to SOTA results.