

Tutorial: Automating Commonsense Reasoning*

Gopal Gupta¹, Elmer Salazar¹, Sarat Chandra Varanasi¹, Kinjal Basu¹, Joaquín Arias², Farhad Shakerin¹, Fang Li¹ and Huaduo Wang¹

¹University of Texas at Dallas, USA

²CETINIA, Universidad Rey Juan Carlos, Spain

Abstract

Automating commonsense reasoning, i.e., automating the human thought process, has been considered fiendishly difficult. It is widely believed that automation of commonsense reasoning is needed to build intelligent systems that can rival humans. We argue that answer set programming (ASP) along with its goal-directed implementation allows us to reach this automation goal. We discuss essential elements needed for automating the human thought process, and show how they are realized in ASP and the s(CASP) goal-directed ASP engine.

I don't see that human intelligence is something that humans can never understand.

— John McCarthy, March 1989

1. Introduction

Artificial Intelligence (AI) researchers are coming to the realization that for intelligent systems to be as good as humans, they must incorporate commonsense reasoning. While machine learning technologies have had tremendous successes, they alone are not enough. Intelligent behavior includes both learning and (commonsense) reasoning, and automating commonsense reasoning is equally important. Indeed, automating commonsense reasoning has been the goal of research in artificial intelligence for a long time. Automation of commonsense reasoning requires formalizing the human thought process, which has proven to be hard. The quest to formalize the human thought process led to the founding of the field of Logic. The study of logic as a means to formalize reasoning has been conducted over several millennia [1, 2]. In modern times this effort culminated in Boolean logic [3], first order logic [4], and various other advanced logics [5]. These logics, however, are limited in one way or the other. They cannot match the sophistication of human reasoning in the sense that it is hard to use these logics to faithfully model the human thought process in an elegant manner.

*Work partially supported by EIT Digital, MICINN project RTI2018-095390-B-C33 InEDGEMobility (MCI-U/AEI/FEDER, UE), US NSF (Grants IIS 1718945, IIS 1910131, IIP 1916206), DoD, and Amazon. We are grateful to input from a large number of people, including students in our logic programming courses, who have used the s(CASP) system and given us feedback. We thank Jan Wielemaker for including the s(cASP) system in SWISH.

2nd Workshop on Goal-directed Execution of Answer Set Programs (GDE'22), August 1, 2022

✉ gupta@utdallas.edu (G. Gupta); joaquin.arias@urjc.es (J. Arias)

ORCID 0000-0001-9727-0362 (G. Gupta); 0000-0001-8693-9307 (K. Basu); 0000-0003-4148-311X (J. Arias)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

The early problems of naive set theory found by Russell—as illustrated by Russell’s paradox [6]—led mathematicians and logicians to only focus on well-founded reasoning, i.e., inductive reasoning. Well-founded reasoning stipulates that to reason soundly one has to start from the simplest object (e.g., an empty set) and then build larger objects by embellishing the simpler objects. That is, we could obtain one element sets by adding an element to the empty set, two element sets by adding another element to one element sets, and so on. *Assumption-based reasoning* that humans frequently employ, and which requires *circular* or *coinductive reasoning* [7], was banished from mathematical discourse. Only recently work on study of non well-founded sets and coinductive reasoning has been taken up [7, 6, 8].

Because of this dogmatic focus on restricting reasoning to inductive structures since the early 20th century, systems of logic could not reason about themselves. Meta-reasoning within the logic was disallowed due to fears of unsoundness and circularity. Thus, classical logics do not have the ability to derive a conclusion based on failure of a proof in that logic itself. Tarski stipulated that given a logic L_1 , we need another logic L_2 to reason about L_1 , and yet another logic L_3 to reason about L_2 , and so on, *ad infinitum* [9, 6]. Thus, Tarski deemed it impossible for a language to have its own *truth predicate*. Only in 1975 did Kripke show that a language can consistently contain its own truth predicate [10]. Kripke’s work eventually led to work on circular reasoning and coinduction [7, 6, 11]. Coinduction, the dual of induction, allows us to perform circular or assumption-based reasoning, as we shall see later.

In the late 1970s and 1980s considerable effort started to being invested in adding negation to logic programming. The concept of *negation as failure* [12, 13] was added to logic programming along with notion of stable model semantics that admitted multiple possible worlds, and thus admitted assumption-based reasoning [14]. Negation-as-failure is frequently employed by us humans. It allows us to take an action *if proof of a goal fails*. That is, if a goal cannot be achieved because we fail to prove it, we may take an action that is predicated on the failure of the proof. Thus, negation-as-failure allows us to work with incomplete knowledge, as a proof may fail because of lack of information. Being able to work with incomplete knowledge, arguably, is the essence of commonsense reasoning. Classical logic cannot reason about proof failure. For example, if we program reachability (of one node in a directed graph from another node) in logic, then the axioms for *reachability* cannot be used to prove *unreachability*. Separate axioms have to be given to reason about unreachability. With negation-as-failure, unreachability can be simply realized by stating that if the proof of reachability from node N_1 to N_2 fails, then node N_2 is unreachable from node N_1 .

While coinduction [7] and negation as failure [13] have been around for more than 40 years, they did not lead to formalization of the human thought process. The advent of *answer set programming* (ASP) [14] based on the stable model semantics for negation-as-failure [15] made this possible. ASP allows complex human thought processes such as default reasoning, counterfactual reasoning, abductive reasoning and reasoning over multiple worlds to be formally realized in an elegant manner. Progress was still limited by the type of implementations available for ASP that only admitted propositional logic programs. While these implementations are highly sophisticated, their use for knowledge representation and reasoning is beset by many problems. Recent development of goal-directed or query-driven ASP execution engines that can also handle predicates, such as the s(ASP) [16] and s(CASP) [17] systems, overcame many of these problems. These goal-directed predicate ASP systems hold the key to achieving practical

automation of commonsense reasoning.

To understand the significance of automating commonsense reasoning, consider autonomous driving. To learn to drive, a human driver has to mostly focus on learning how to control the car (steering, accelerator, brakes) and know the rules of the road. A human doesn't have to be explicitly taught that if a pedestrian is in front of the car, they must brake and stop. In contrast, for an autonomous car, steering, braking, accelerating can be readily automated with advances in control theory, however, "knowing" how it must react to traffic on the road is a major obstacle today. For a human driver, how he/she must behave on the road is dictated by their commonsense and the rules that they learn during driver education. Learning to steer, brake and accelerate (and use the clutch, for stick-shift cars) is the harder part for humans. Thus, an autonomous vehicle that takes driving decisions based on automated commonsense reasoning may perform better than one that is based purely on machine learning technology [18]. Such a system, of course, will use machine learning technologies for scene processing, just as humans use their eyes and ears to understand the environment around them.

In the rest of the paper we make a case for how answer set programming executed using goal-directed implementations of ASP with predicates leads to automation of commonsense reasoning. We assume the reader has familiarity with logic programming and Prolog.

2. Automating Commonsense Reasoning

Aside from being able to perform deduction, the main characteristics of human commonsense reasoning are that (i) we can reason with incomplete information (e.g., default reasoning), (ii) perform assumption-based or circular reasoning, and (iii) reason with constraints. We explain these aspects in more detail.

2.1. Reasoning with Incomplete Information

Humans often reason with incomplete information. Reasoning with incomplete information amounts to drawing a conclusion from a failed proof as the proof cannot be completed due to insufficient information. Inference procedures in classical logic assume complete information. This means that if we give axioms to define a property p in classical logic, then nothing can be concluded about $\neg p$ if we are *unable* to prove p . Separate axioms are needed to draw conclusions about $\neg p$. Humans, in contrast, very often, will infer negation of p if they *fail* to prove p , given the axioms about p . A human's ability to draw such conclusions is modeled through negation-as-failure. Humans use negation-as-failure (NAF) copiously in their day-to-day reasoning process.

Negation-as-failure (*not p*) is quite distinct from negation in classical logic ($\neg p$). Negation-as-failure is traditionally represented as 'not p' and can be interpreted as *no evidence that p holds*. Negation in classical logic (termed classical negation) is denoted as $\neg p$ and can be interpreted to mean that p is *definitely* false (that is, there is definite evidence that p does not hold).

Negation-as-failure allows us to draw conclusions when a proof fails. As an example, consider the recursive logic programming rules (axioms) for reachability of one node from another node in a graph, where $edge/2$ represents the graph edge relation.

$$\text{reach}(X, Y) \leftarrow \text{edge}(X, Y).$$

$$\text{reach}(X, Y) \Leftarrow \text{edge}(X, Z) \wedge \text{reach}(Z, Y).$$

We can use these (classical) logic axioms to check if a given node *can* be reached from another. However, we cannot use them to check if a given node *cannot* be reached from another node: we will need separate axioms for non-reachability. In contrast, with NAF, we can use `not reach(A, B)` to check if A is unreachable from B. `not reach(A, B)` simply says that if proof for `reach(A, B)` fails, we conclude B is unreachable from A. Failure of a proof means that there is no way to make progress in the proof. We could take an action based on this failed proof (e.g., use a helicopter to go from point A to point B, where edge represents a road-connection between cities [nodes]).

It should be mentioned that fault is not of logic, rather how a set of formulas are interpreted. As humans we frequently interpret an implication (if statement) as a bi-implication (if and only if statement). To draw negative inferences automatically, we have to use the idea of *program completion* [12, 15], where the above rules for reachability are interpreted as:

$$\text{reach}(X, Y) \Leftrightarrow \text{edge}(X, Y) \vee (\text{edge}(X, Z) \wedge \text{reach}(X, Y))$$

Completing a program above amounts to adding the *dual rule*:

$$\text{not reach}(X, Y) \Leftarrow \text{not edge}(X, Y) \wedge \text{not} (\text{edge}(X, Z) \wedge \text{reach}(Z, Y)).$$

This rule can be written as a logic program by applying De Morgan's law. As humans, we automatically complete such rules in our mind. The dual rule essentially allows us to infer that `not reach(X, Y)` holds if we fail to prove that X can reach Y. By completing a program we are assuming that the rule is causal. Thus, given the rule $p \Leftarrow B.$, which states that p can be inferred if B holds, its completion allows us to conclude that p does not hold if we have no evidence that B holds.

2.2. Circular Reasoning

Humans also perform circular reasoning, or assumption based reasoning. Consider the following statement about Jack and Jill who are in love with each other:

Jack will eat dinner if Jill will eat dinner.

Jill will eat dinner if Jack will eat dinner.

To answer the question if Jack will eat dinner, we end up performing circular reasoning. If we reflect on these statements, then we can see that there are two possibilities: either both will eat dinner, or none will eat. If we interpret the above statements as logic program rules:

`jack_eats` \Leftarrow `jill_eats`.

`jill_eats` \Leftarrow `jack_eats`.

then the completion of these rules is

`jill_eats` \Leftrightarrow `jack_eats`

Viewed as a propositional formula, we can produce two models: $\{\text{jill_eats} = \text{true}, \text{jack_eats} = \text{true}\}$ and $\{\text{jill_eats} = \text{false}, \text{jack_eats} = \text{false}\}$. The first one corresponds to both eating dinner and the second one to none eating dinner. If we stay in the realm of propositions, there is no problem, despite the apparent circular dependence, we can compute the two models. The problem arises when we graduate to predicates. Consider the rule:

`eat_dinner(X)` \Leftarrow `loves(X, Y), eat_dinner(Y)`.

then giving this rule an operational semantics to perform computation becomes complicated. Coinduction and coinductive logic programming, based on greatest fixpoint semantics, can be

used to give operational semantics to such circular rules containing predicates [11, 8].

2.3. Constraints

Humans perform reasoning under constraints. For example, we know that we cannot sit and walk at the same time. So the moment we know that a person was sitting on a chair, we conclude that he/she was not walking. From the knowledge that the person was sitting, we can even conclude that the person could not have moved away from the spot that person was in. These constraints can be thought of as integrity constraints and we employ them to make our proof process efficient. Constraints can be generally stated as declaring that the conjunction of a set of propositions or predicates is false, e.g.,

$$\text{false} \Leftarrow \text{sitting}(X) \wedge \text{walking}(X).$$

As an aside, for any proposition or predicate p , we have the following implicit constraint in our mind:

$$\text{false} \Leftarrow p \wedge \neg p.$$

2.4. Modeling Human Thought

Negation-as-failure models incomplete information. Along with assumption-based reasoning and constraints, human thought processes can now be coded as rules in logic. Consider the following four statements:

1. Paul will go to Mexico if no evidence that Sally will go to Mexico.
2. Sally will go to Mexico if no evidence that Rob will go to Mexico.
3. Rob will go to Mexico if no evidence that Paul will go to Mexico.
4. Rob will go to Mexico if no evidence that Sally will go to Mexico.

The question we want to answer is: who will go to Mexico? Notice that each rule is quite easy to understand by itself, however, it is hard to understand what these rules mean when put together, i.e., it is nearly impossible to compute the model of this program in our head. These rules are also circular. Thus, as humans, these rules will make sense if we consider models of these rules that preserve the consistency. Since these rules are circular, i.e., are based on assumptions, we will make assumptions and then check that the rules are satisfied. Those assumptions that satisfy the rules are the ones that, as humans, we are interested in as the answers we want to compute.

Note also that, as humans, we give an operational semantics to logic rules such as the ones we have seen above. For instance, consider the rule:

$$\text{flies}(\text{tweety}) \Leftarrow \text{bird}(\text{tweety})$$

This rule will have 3 models: $\{\text{bird}(\text{tweety}) = \text{true}, \text{flies}(\text{tweety}) = \text{true}\}$, $\{\text{bird}(\text{tweety}) = \text{false}, \text{flies}(\text{tweety}) = \text{true}\}$, and $\{\text{bird}(\text{tweety}) = \text{false}, \text{flies}(\text{tweety}) = \text{false}\}$. We generally ignore the models where the antecedent $\text{bird}(\text{tweety})$ is false. The first model where the consequent and the antecedent are true is termed a *supported model* [15].

Moreover, when we see such rules, we automatically “complete” them in our mind (in the sense of program completion discussed earlier). That is, we interpret the above rule as:

$\text{flies}(\text{tweety}) \Leftrightarrow \text{bird}(\text{tweety})$

That is, we implicitly assume that if Tweety is not a bird, it does not fly. Wason's selection task [19] indeed shows that humans interpret an *if* as an *if and only if*¹. In Wason's selection task, people are shown four cards, where each card has a letter on one side and a number on the other side.

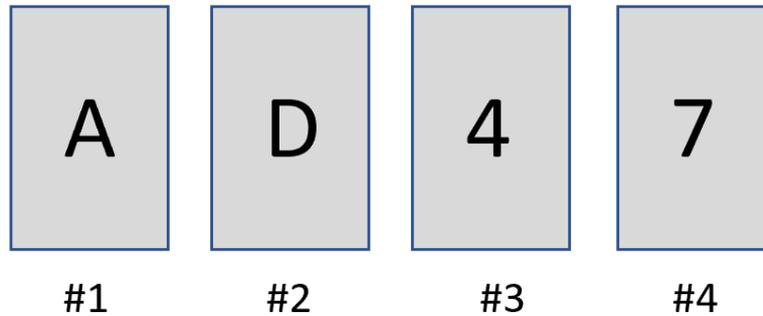


Figure 1: Wason Selection Task Example

Human subjects are asked “which cards must you turn over in order to test the truth of the statement that if a card shows an vowel on one face, then its opposite face is an even number.” 45% of the people in the original experiment selected cards 1 and 3. The reason is that the humans interpret the *if* statement as a causal statement: a card has a letter on one side *iff* it has a vowel on the other side. Wason's selection task also demonstrates that we are only interested in *supported models*, as the models arising out of $\text{false} \Leftrightarrow \text{false}$ are ignored by the subjects.

Answer set programming (ASP), that extends Prolog with negation-as-failure based on the stable model semantics, supports the features discussed above: negation-as-failure, constraints, circular reasoning, completion, and supported models. These features allow ASP to closely emulate the human thought process [15]. We briefly discuss answer set programming next.

3. Answer Set Programming

Answer set programming (ASP) [20, 15], a paradigm based on logic programming, incorporates reasoning with incomplete information, circular reasoning, as well as constraints. ASP extends logic programming with negation-as-failure based on the stable model semantics [14]. Negation-as-failure allows ASP to perform non-monotonic reasoning. Non-monotonic reasoning is essential for modeling commonsense reasoning since as our knowledge changes, i.e., as more things become known from being unknown, the conclusions we can draw may change. For example, a conclusion that could have been drawn earlier may have to be retracted. ASP is a highly expressive paradigm that can elegantly express complex reasoning methods used by humans such as default reasoning, deductive and abductive reasoning, counterfactual reasoning,

¹In some situations the dual is not inferred as in those familiar situations, the dual rule is suppressed through additional knowledge.

and constraint satisfaction [20, 15]. Answer set programs can also elegantly represent inductive generalizations.

ASP supports better semantics for negation (*negation as failure* or *NAF*) than does standard logic programming and Prolog. An ASP program consists of rules that look like Prolog rules. However, the semantics of an ASP program P is given in terms of the *answer sets* (models) of the program $\text{ground}(P)$, where $\text{ground}(P)$ is the program obtained from the substitution of elements of the *Herbrand universe*² for variables in P [20]. An answer set program consists of rules of the form:

$$p :- q_1, \dots, q_m, \text{not } r_1, \dots, \text{not } r_n.$$

where $m \geq 0$ and $n \geq 0$. Each of p and q_i ($\forall i \leq m$) is a literal, and each $\text{not } r_j$ ($\forall j \leq n$) is a NAF-literal. The declarative semantics of an Answer Set Program P is given via the Gelfond-Lifschitz transform [20, 15] in terms of the answer sets of the program $\text{ground}(P)$, as discussed earlier. ASP also supports classical negation. A classically negated predicate is denoted as $\neg p$ and means that p is provably false. Its definition is no different from a positive predicate, in that explicit rules have to be given to establish $\neg p$. More details about ASP can be found elsewhere [20, 21, 15].

The goal in ASP is to compute an *answer set* given an answer set program, i.e., compute the set that contains all propositions that if set to true will serve as a model of the program (those propositions that are not in the set are assumed to be false). Intuitively, the rule above says that p is in the answer set if q_1, \dots, q_m are in the answer set and r_1, \dots, r_n are not in the answer set.

As mentioned earlier, default negation or negation as failure is represented as $\text{not } p$ and can be interpreted as *no evidence that p holds*. Classical negation is denoted as $\neg p$ and can be interpreted to mean that p is *definitely* false. To further understand the distinction, consider a case of bank b being allegedly robbed by an individual i . If we *fail* to prove that x robbed the bank, then $\text{not } \text{robbed}(i, b)$ holds true (there is no evidence that i robbed bank b). In contrast, if we able to definitively show that i did not rob bank b by demonstrating, for example, that the bank was located in Dallas, but i was seen at the Boston airport at the time the robbery happened, then $\neg \text{robbed}(i, b)$ is true. In the latter case we have an explicit proof to demonstrate that $\text{robbed}(i, b)$ is false.

The two types of negation allow us to realize various “shades” of truth, namely, *definitely true*, *maybe true*, *unknown*, *maybe false*, and *definitely false*. They are crucial in allowing ASP to simulate commonsense reasoning. ASP only considers supported models. It also assumes that the programs are completed with the caveat that cyclical programs that do not involve negation will be assigned only a single model in which the propositions involved in the cycle are assigned false. Thus, given the rules above about Jack and Jill eating dinner, the only model ASP will admit is the one where they both don’t eat. This assumption is made to stay consistent with Prolog, from which ASP originates. Next, we discuss how commonsense reasoning can be automated with ASP.

²The Herbrand universe of a logic program is the set of terms that can be constructed from its constants and function symbols.

4. Commonsense Reasoning

4.1. Default Reasoning with exceptions and preferences

Humans use *default reasoning* [22, 15] to jump to conclusions. These conclusions may be revised later in light of new knowledge. For example, if we are told that Tweety is a bird, and then asked whether Tweety flies, we will immediately answer, yes, it does. However, later if we are told that Tweety is a penguin, we will withdraw the conclusion about Tweety's flying ability, labeling Tweety as an *exception*. Thus, human reasoning is non-monotonic in nature, meaning that conclusions may be withdrawn as new knowledge becomes available. Humans use this sort of *default reasoning* to jump to conclusions all the time, and if they find the assumptions made to jump to this conclusion to be incorrect, they revise their conclusion.

Multiple default conclusions can be drawn in some situations, and humans will use additional reasoning to *prefer* one default over another. Thus, *default rules with exceptions and preferences* capture a bulk of human reasoning. (More details on default reasoning can be found in [15]). It should be noted that expert knowledge is nothing but a set of default rules about a specialized topic [15].

Classical logic is unable to model default reasoning and non-monotonicity in an elegant way. We need a formalism that is non-monotonic and can support default reasoning to model the human thought process. ASP is such a formalism. ASP supports both negation-as-failure (`not p`) as well as classical negation (`-p`), where `p` is a proposition or a predicate. Combining these two forms of negations results in nuanced reasoning close to how humans reason:

1. `p`: denotes that `p` is *definitely* true.
2. `not -p`: denotes that `p` *maybe* true (i.e., no evidence that `p` is false).
3. `not p ^ not -p`: denotes that `p` is unknown (i.e., no evidence of either `p` or `-p` being true).
4. `not p`: denotes that `p` *may* be false (no evidence that `p` is true).
5. `-p`: denotes that `p` is *definitely* false.

The above insight can be used, for example, to model the exceptions to Tweety's ability to fly in two possible ways. Consider the rules:

```
flies(X):-bird(X), not abnormal_bird(X).    % default
abnormal_bird(X):-penguin(X).              % exception
```

which state that if we know nothing about a bird, `X`, we conclude that it flies. This is in contrast to the rules:

```
flies(X):-bird(X), not abnormal_bird(X).    % default
abnormal_bird(X):-not -penguin(X).          % exception
```

which states that a bird flies only if we can *explicitly* rule out that it is a penguin. So in the latter case, if we know nothing about a bird, we will conclude that it *does not* fly. Which of the two rules one will use depends on how conservative or aggressive one wants to be in jumping to the default conclusion. Note that exceptions can have exceptions, which in turn can have their own exceptions, and so on. For example, animals normally don't fly, unless they are birds. Thus, birds are exception to the default of not flying. Birds, in turn, normally fly, unless they are

penguins. Thus, a penguin is an exception to the exception for not flying. Defaults, exceptions, exceptions to exceptions, and so on, allow us humans to perform reasoning elegantly and in an *elaboration tolerant* manner [20, 15].

The abnormal predicate represents exception to the default conclusion. Such an exception is called a *weak* exception. A strong exception can be directly stated by specifying objects that definitely cannot fly. For example, if we know that ostriches cannot fly, we represent it as:

```
-flies(x):-ostrich(x).
```

which implicitly states that an ostrich x can never satisfy `flies(x)`. However, we will have to extend our default rule above to avoid a contradiction if we have the knowledge that ostriches are birds

```
bird(x):-ostrich(x).
```

The extended rule is as follows:

```
flies(x):-bird(x), not abnormal_bird(x), not -flies(x). % default
```

The extension prevents the original rule from concluding that x flies if there is definitive evidence that it does not, thereby preventing a contradiction. Note that ostriches are not declared to be abnormal birds.

Defaults, strong/weak exceptions, nested exceptions, and aggressive/conservative reasoning, allow us to construct a set (e.g., birds that can fly) in a nuanced, elaboration tolerant manner as we gradually learn more information about which birds can fly and which cannot. Elements are included or excluded from the set as we acquire more knowledge and the set is gradually refined as the decision boundary is adjusted. Default rules with nested exceptions are an excellent vehicle for representing the rules underlying a machine learning model [23].

Note that these 5 shades of truth discussed above are effective, because that's what humans use. Humans *do not* use probabilities in their day to day life. When told that Tweety is a bird, we do not say that Tweety's chance of flying is 95% (or whatever probability one can come up with for a bird to fly). We will jump to the conclusion that Tweety flies, and if contrary information becomes available later, we revise it. This has been borne out by earlier research. Quoting from the AI text book by Ginsberg [24] (page 231), we learn that moving from probabilities to four shades of truth did not degrade performance of the MYCIN system.

“An experiment was done in which MYCIN's set of truth values (the real numbers between -1 and 1) was replaced with a just a set of four values: -1 (certainly false), -0.3 (evidence against), +0.3 (evidence for), and +1 (certainly true). The declarative sentences labeled with certainty factors that fell between these values were simply given the closest label possible. *The interesting thing is that MYCIN's performance did not degrade after this change was made.*”

Note that the value *unknown* (zero) is missed. Perhaps modeling it also would have brought MYCIN closer to a human expert.

4.2. Possible Worlds

Humans can represent *multiple possible worlds* in parallel in their minds and reason over each. For example, in the real world, fish do not talk like humans, while in a cartoon world, fish

(cartoon characters) can talk. Humans can maintain the distinction between various worlds in their minds and reason within each one of them. These multiple worlds may have aspects that are common (fish can swim in both the real and cartoon worlds) and aspects that are disjoint (fish can talk only in the cartoon world). Thus, we need a logic that can support multiple possible worlds. Classical logic, due to various historical choices made (allowing only inductive structures and inductive proofs [6]), can only deal with a single world. ASP supports possible world semantics as well. Given the rules

```
teaches(john, db):-not teaches(mary, db).
teaches(mary, db):-not teaches(john, db).
```

an ASP engine will produce two worlds: one in which John teaches databases and Mary does not

```
{teaches(john, db), not teaches(mary,db)}
```

and the other one where Mary teaches and John does not

```
{teaches(john, db), not teaches(mary,db)}
```

For clarity, we explicitly list both positive and negative literals in the answer set here. ASP will assign models to such *non-well founded* programs. If we interpret the rules operationally as in Prolog, then a call to `teaches(john, db)` will loop forever. Such a cyclical set of rules where the recursive call is in the scope of even number of negations is called an *even loop over negation*.

Consider the example that comes from Peter Norvig in which he gives a challenge problem that is hard to model in classical logic:

1. People can talk.
2. Non-human animals are not able to talk.
3. Human-like cartoon characters can talk.
4. Fish can swim.
5. A fish is a non-human animal.
6. Nemo is a human-like cartoon character.
7. Nemo is a fish.

The question is “can Nemo talk?” and, “can Nemo swim?”. The answer to “Can Nemo talk?” leads to a contradiction under classical logic. This happens because two worlds (real and cartoon) are intertwined here. Humans easily recognize this, but it is harder for machines. However, this is easily resolved in ASP, where we can have two worlds: real world (rw) and cartoon world (cw). We just have to know which statement is true in which world(s). Note that some information is mutually exclusive between two worlds, while some may be common, and some may be known in one world but not (yet) in the other. The encoding in ASP for each of the statements is shown below.

```
1 talk(X):- people(X).           % people talk in both worlds
2 -talk(X):- non_human_animal(X), rw. % non human animals do not talk in rw
3 talk(X):- human_like_cc(X), cw.  % human-like cartoon char can talk in cw
4 swim(X):- fish(X).             % fish swim in both worlds
5 non_human_animal(X):- fish(X), rw. % fish is a non-human-animal in rw
6 human_like_cc(nemo):- cw.       % Nemo is a human-like cartoon char in cw
7 fish(nemo).                    % Nemo is a fish in both worlds
```

8

```
9 cw:- not rw.                                % cw and rw are two separate worlds
10 rw:- not cw.
```

An ASP engine will produce two models: one corresponding to the cartoon world (cw) and the other to the real world (rw). Only cw will contain `talk(nemo)` while both worlds will contain `swim(nemo)`. Possible worlds can also be used to simulate abductive reasoning [25], as we shall see later.

4.3. Constraints

ASP can also model constraints elegantly. A constraint is a rule of the form:

```
false:-p1, p2, ..., pn.
```

which states that the conjunction of p_1, p_2, \dots, p_n is false (the keyword `false` is often omitted). Constraints elegantly model global invariants or restrictions that our knowledge must satisfy, e.g., p and $\neg p$ cannot be true at the same time, denoted

```
false:-p, -p.
```

Humans indeed use constraints in their everyday reasoning: as restrictions (two humans cannot occupy the same spot) and invariants (a human must breathe to stay alive).

Earlier we saw even loops over negation. *Odd loops over negation* also exist, for example, programs such as:

```
p:-q, not p.
```

An odd loop over negation represents a constraint. If we complete the rule above, we obtain $p \Leftrightarrow (q \wedge \text{not } p)$. A little thought will reveal that the only feasible model is $p = \text{false}$, $q = \text{false}$. In other words, this rule is stating that q *must* be false. This can be alternatively stated via a constraint.

```
:-q.
```

Note that if there is an alternative proof of p via other rules, then the above odd loop rule will be rendered inapplicable due to p being true through other means. If we wish to reason in terms of possible worlds, then even loops are used to generate multiple worlds and odd loops or constraints are used to “kill” worlds.

Next we show that ASP is a good vehicle for realizing deductive and abductive reasoning—also performed by humans—as well as a good way of representing inductive generalizations.

5. Deduction, Abduction, and Induction

There are three major modes of reasoning that humans use in their day to day life: deduction, abduction, and induction. Consider the proposition p , q , and the formula $p \Rightarrow q$.

Deduction: Given premises p and $p \Rightarrow q$, we *deduce* q . Suppose we are given the premises that Tweety is a bird ($bird(tweety)$), and the formula $\forall X bird(X) \Rightarrow flies(X)$. From these two premises, we can deduce that $flies(tweety)$ holds, i.e., Tweety can fly. Deductive reasoning is easily expressed using answer set programming. If we consider Rule 1, then given $q_1, \dots, q_m, \text{not } r_1, \dots, \text{not } r_n$, we can deduce p . Essentially, the antecedent can become more complex and can include negation-as-failure (NAF) literals.

Abduction: Given the observation q and the premise $p \Rightarrow q$, we *abduce* p . Suppose we observe Tweety flying ($flies(tweety)$), and we know that $\forall X bird(X) \Rightarrow flies(X)$. From these, we can *abduce* that $bird(tweety)$ holds, i.e., we assume (or advance the most likely explanation) that Tweety is a bird. Note that there may be other explanations, as Tweety may be an airplane. Generally, the set of abduced literals is fixed in advance. Abductive reasoning in ASP is elegantly modeled via possible worlds. If we make an assumption p , i.e., declare p to be abducible, then we can assert an even loop over negation:

```
p:- not _notp.  
_notp:-not p.
```

where $_notp$ is a dummy proposition. This even loop will result in two possible worlds: one in which p is true and one in which p is false. The assumption that will produce a consistent model (either p true or p false) is the correct assumption that we will abduce, if we add this even loop to a program where we want to find a likely explanation.

Induction: Given instances of p and corresponding instances of q that may be related, we may induce $p \Rightarrow q$. Thus, given the observations that Tweety is a bird and Tweety can fly, Sam is a bird and Sam can fly, Polly is a bird and Polly can fly, and so on, we may *induce* $bird(X) \Rightarrow flies(X)$. Induction, of course, relates to learning associations between data. The results of rules induced can be elegantly captured as an answer set program. This is because ASP can be used to represent defaults with exceptions, which allows us to elegantly represent inductive generalizations. Consider the following rule:

```
flies(x):-bird(x), not abnormal_bird(x).  
abnormal_bird(x):-penguin(x).
```

The above default rule with exception, namely, *normally birds fly unless they are penguins*, elegantly captures the rule that a human may form in their mind after observing birds and their ability to fly. The list of exceptions can grow (ostrich, wounded bird, baby bird, ...). Similarly, the list of defaults rules can grow. For instance, we may have a separate rule for planes being able to fly. Explainable machine learning tools that induce default theories as answer set programs have been developed [23]. These tools are comparable in accuracy to traditionally popular tools such as XGBoost [26] and Multi-level Perceptrons [27].

Given that deduction, abduction, and induction fit into the framework of ASP well, it gives us confidence that ASP can be a good means of representing commonsense knowledge.

6. Implementations of ASP

ASP is a powerful, expressive paradigm for representing knowledge. It represents knowledge in a manner very similar to how humans represent it and use it. However, it is a challenging task to implement ASP. Most implementations of answer set programming work by first grounding the program with elements of the program's Herbrand universe, then suitably transforming the program in a way such that the models of the transformed program computed with a SAT solver are identical to the original program's answer sets [28, 29]. While these grounding and SAT solver-based implementation approaches are extremely efficient, and have been used for very sophisticated applications, they have several drawbacks:

1. SAT solver-based methods only work for programs that have constants and variables as

arguments of predicates. Programs with general terms such as lists and trees will produce grounded programs of infinite size. Even if there are no terms in the program, the size of the grounded program can be exponential in size of the original program.

2. A program that involves reasoning over real numbers will also lead to an infinite-sized grounded program.
3. These methods generate the *whole* model for the program which can be quite large. In most cases we simply want to know if a goal is true or not for a given program.
4. Justification for a given element of the answer set, i.e., why it is true and thus belongs to the answer set, is hard to produce in these methods.

Humans, obviously, do not compute the entire model given the knowledge of the world, as it will be too inefficient. Humans also reason in a goal-directed manner, i.e., they establish a target goal and then attempt to establish that it holds. This can be simulated by goal-directed execution of ASP. The advantage of goal-directed execution of ASP is that only relevant part of the knowledge is used in proving a goal, very much like in Prolog. In goal-directed execution, programs containing predicates can be directly executed without requiring grounding.

Recently, goal-directed implementations of ASP have been realized that can execute answer set programs containing predicates in a query-driven manner [16, 17]. These systems can be thought of as implementation of Prolog extended with negation-as-failure based on stable model semantics. Incorporation of negation-as-failure based on stable model semantics opens the door for Prolog to support default reasoning with exceptions and preferences, reasoning over possible worlds, constraints, and assumption-based reasoning. In short, this extended Prolog can be used for emulating the human thought process.

These goal-directed ASP systems support predicates that can take arbitrary terms as arguments. Implementation of such systems became possible because of development of coinductive logic programming [11, 8] that gives an operational semantics to cyclical programs. Thus, an *even loop over negation* and an *odd loop over negation* can be given operational semantics to obtain Prolog-like query-driven implementation of ASP. Coinductive logic programming has to be coupled with constructive negation and various other advances to build an effective query-driven predicate ASP system. The first such system that implemented predicate ASP was the s(ASP) system [16]. This was followed by the s(CASP) system [17] which, in addition, provides support for constraint solving over reals. The s(CASP) system is now available as a library under the widely-used SWI-Prolog system (<https://swish.swi-prolog.org/>).

7. The s(CASP) Goal-directed ASP System

The s(CASP) system can be thought of as Prolog extended with negation-as-failure based on the stable model semantics. The s(CASP) system also supports constraints over reals. The s(CASP) system provides support for full Prolog, however, in addition, it also supports coinductive reasoning, constructive negation, dual rules, and support for universally quantified variables. These are explained briefly next. More details can be found elsewhere [16, 17].

7.1. Coinductive Reasoning

As discussed earlier, answer set programs may contain circular rules, for example:

```
p :- not q.  
q :- not p.
```

If we ask the query $?-p$ in Prolog, given the rules above, the execution will loop forever. This is because p reduces to $\text{not } q$, which, in turn, reduces to $\text{not not } p$ which equals p . If we appeal to *greatest fixpoint semantics*, then the query p should succeed. Essentially, what we are saying is that p succeeds if we assume p to hold. This yields the answer set in which p is true and q false. Note that all the literals encountered along the way (e.g., $\text{not } q$) will be in the (partial) answer set.

If we issue the query $?-\text{not } p$, then we will find the other answer set where q is true and p is false. Thus, if we ask a query in $s(\text{CASP})$, it will give a partial answer set that is a subset of the world in which the query is true. If there are multiple such worlds in which the query holds, we will compute them one by one through backtracking (by typing a semi-colon, as in a Prolog system).

While circular reasoning of the type above seems counter-intuitive, as humans, we perform such assumption based (circular) reasoning all the time, as illustrated in the examples discussed earlier. Consider a murder mystery in which our goal is to find the murderer. We make various assumptions about who the murderer could be, and then perform reasoning with this assumption to find a consistent world in which this assumption holds. Assumption based reasoning is, essentially, abductive reasoning discussed earlier that ASP supports by introducing an even loop for each abducible. Abductive reasoning is thus elegantly supported in $s(\text{CASP})$.

7.2. Constructive Negation

Since $s(\text{CASP})$ allows general predicates that could be negated, support for constructive negation becomes essential. Consider a program consisting of the simple fact

```
p(a).
```

If we pose the query $?-\text{not } p(X)$, it should succeed with answer $X \neq a$. Intuitively, $X \neq a$ means that X can be any term not unifiable with a . To support constructive negation, the implementation has to keep track of values that a variable cannot take. The Unification algorithm has to be extended, therefore, to account for such disequality-constrained values.

In the $s(\text{CASP})$ system, a variable can only be disequality-constrained against ground terms, and the disequality of two compound terms may require backtracking to check all the cases: $p(1, Y) \neq p(X, 2)$ first succeeds with $X \neq 1$ and then, upon backtracking, with $Y \neq 2$. We omit further details here, as they can be found elsewhere [17, 16]. The point to underscore is that while realizing constructive negation is challenging, it's an essential element in modeling of human-style commonsense reasoning.

7.3. Dual Rules

As we discussed earlier, given an answer set program, we work with its completion. To complete the program, the $s(\text{CASP})$ system will add the dual rules [30] to the program. The procedure to add the dual rules is relatively simple: Given rules of the form below, say, for a predicate p :

$p(t_{11}, \dots, t_{n1}) :- B_1.$

...

$p(t_{1m}, \dots, t_{nm}) :- B_m.$

they are coalesced into a single rule containing disjunction of bodies:

$p(x_1, \dots, x_n) :-$

$(x_1 = t_{11}, \dots, x_n = t_{n1}, B_1) \vee$

...

$\vee (x_1 = t_{1m}, \dots, x_n = t_{nm}, B_m).$

Next, the dual is represented as:

$not_p :- \neg B$

where B represents the body of the coalesced clause. We will use De Morgan's law to simplify the dual rule. Consider the following example:

$redcar(mycar).$

$redcar(X) :- red(X), car(X).$

This rule is rewritten as:

$redcar(X) :- (X = mycar); (red(X), car(X)).$

where ';' denotes disjunction. The dual rule will, obviously, be:

$not_redcar(X) :- X \neq mycar.$

$not_redcar(X) :- not red(X).$

$not_redcar(X) :- not car(X).$

To make the execution more efficient, s(CASP) will rewrite the third clause as:

$not_redcar(X) :- red(X), not car(X).$

Essentially, the dual rule states that a non-red car is anything other than `mycar` as well as anything that is not red, and among red things those that are not cars.

An additional complication in computing the dual rules is the need to handle existential variables. Consider the following very simple rule:

$p(X) :- not q(X, Y).$

This rule corresponds to the Horn clause:

$\forall X (p(X) \Leftarrow \exists Y not q(X, Y))$

Its dual will be:

$\forall X (not_p(X) \Leftarrow \forall Y q(X, Y))$

which, in s(CASP), will be represented as:

$not_p(X) :- forall(Y, q1(X, Y)).$

$q1(X, Y) :- q(X, Y).$

Universal quantification in the body of the dual rule is needed because, for example, for the goal `not p(a)` to succeed, we must prove that `q(a, Y)` holds for every possible value of `Y`. The s(CASP) system handles all these issues and produces dual rules for arbitrary programs. The execution of the *forall*, however, is non-trivial, as often times the *foralls* are nested.

7.4. Constraints

Constraints in ASP can be expressed directly, e.g.,

$:- p(X), q(X).$

or as odd loops over negation such as:

```
r(x):-q(x), not r(x).
```

When a query is executed against a s(CASP) program, a partial answer set is computed that represents the answer. If this partial answer set does not satisfy the constraint, it is rejected, and backtracking ensues. This is achieved by appending the query with appropriate goals that correspond to executing these constraints. Thus, the s(CASP) system extends the query appropriately to capture the effect of a constraint no matter in which one of the two ways above it is expressed. An appropriate sub-goal is appended for each constraint in the program. As an example, consider the following simple program:

```
p(x):-q(x), r(x).
:- t(a).
```

If we have the query:

```
?- p(x).
```

then s(CASP) will augment this query and turn it into

```
?- p(x), not t(a).
```

since $t(a)$ must be false in every answer set. If the constraint was instead:

```
:- t(x).
```

which states that $t/1$ must be false for *every* value of x in every answer set, then the extended query will be:

```
?- p(x), forall(y, s(x)).
```

where $s(x)$ is defined as:

```
s(x):-not t(x).
```

Note that these extra sub-goals arising from constraints and odd loops over negation that are added to the query should be executed as soon as a variable that occurs in the sub-goal is bound during the execution. This will result in speedier execution, and is indeed supported by s(CASP). In some cases the additional subgoals that the s(CASP) system augments to the query will over-approximate the actual constraint. However, this over approximation only leads to extra execution overhead, it does not impact the correctness of the execution [16].

A goal-driven implementation of ASP can generate a proof for the query, thus, justification for each element of an answer set can be generated [31]. Finding these justification is a major challenge for SAT-solver based ASP systems. In fact, s(CASP) facilitates the generation of explanations in natural language. The s(CASP) system is a sophisticated system that has been used for many advanced applications that rely on modeling commonsense reasoning that are discussed later. More details about the s(CASP) system can be found elsewhere [16, 17]. While a goal-directed implementation makes the ASP approach scalable, its disadvantage is that the search may be non-terminating or may take too long. To eliminate some of these problems, the implementation could be extended with constraint logic programming over finite domains. This is a topic of ongoing research.

8. Examples

Next, we show examples of modeling commonsense knowledge in ASP and executing them under the s(CASP) system.

Consider first a very simple example, where we want to model the college admissions process taken from [20]. Students with a high GPA are normally eligible for admission. Students with fair GPA who have a special skill are also eligible. Students who do not have a high GPA and have no special skills are ineligible. Students whose eligibility or ineligibility cannot be established (eligibility unknown) will be interviewed. This scenario can be modeled as:

```
1 eligible(X):- highGPA(X).
2 eligible(X):- fairGPA(X), specialskill(X).
3 -eligible(X):- -highGPA(X), -specialskill(X).
4 interview(X):- not eligible(X), not -eligible(X).
```

If a student John has a fair GPA, but not a high GPA, the rules above will conclude that John must be interviewed. Consider another example, this one from Levesque's Prolog course on *Thinking As Computation* [32]. Consider the following statements in natural language that we want to represent formally and reason about them.

1. George is a bachelor.
2. George was born in Boston, collects stamps.
3. A son of someone is a child who is male.
4. George is the only son of Mary and Fred.
5. A man is an adult male person.
6. A bachelor is a man who has never been married.
7. A (traditional) marriage is a contract between a man and a woman, enacted by a wedding and dissolved by a divorce.
8. While the contract is in effect, the man (called the husband) and the woman (called the wife) are said to be married.
9. A wedding is a ceremony where . . . bride . . . groom . . . bouquet . . .

From this knowledge, we would like to draw the following conclusions: (i) George has never been the groom at a wedding. (ii) Mary has an unmarried son born in Boston. (iii) No woman is the wife of any of Fred's children. We can model the knowledge in ASP, translate the conclusions we want to draw into queries, and execute them on s(CASP). The following rules represent the knowledge mentioned above and is easily understood:

```
1 % 1
2 bachelor(george).
3 % 2
4 birth_city(george, boston).
5 hobby(george, stamp_collecting).
6 % 3
7 son(X,Y):- child(X,Y), male(X).
8 child(X,Y):- son(X,Y).
9 male(X):- son(X,Y).
10 child(george,mary).
11 child(george,fred).
12 male(george).
```

```

13 % 4.
14 :- son(X, fred), son(Another_son, fred), Another_son #<> X.
15 :- son(X, mary), son(Another_son, mary), Another_son #<> X.
16 % 5.
17 man(X):- adult(X), male(X).
18 % 6.
19 bachelor(X):- man(X), not married(X, Y, T).
20 % 7, 8, 9
21 married(X,Y, T):- groom(X), bride(Y), wedded(X,Y,T1), T #> T1,
22                    not divorced(X,Y,T).
23 -married(X):- bachelor(X).
24 divorced(X,Y, T):- husband(X), wife(Y), dissolved(X,Y,T1), T #> T1.
25
26 groom(X):- wedded(X,Y,T1), male(X).
27 bride(Y):- wedded(X,Y,T1), female(Y).
28 husband(X):- groom(X).
29 wife(X):- bride(X).
30
31 % Wedding precedes divorce
32 :- wedded(X,Y,T1), divorced(X,Y,T2), groom(X), bride(Y), T1 #> T2.

```

We've made some assumptions in coding the knowledge, e.g., a groom and a husband are synonymous. We can make these notions more precise. For example, we could code that a man is no longer called a groom after the wedding is over. This program can be loaded on s(CASP) and queries executed against it. To verify that George has never been the groom at a wedding, we will pose the query:

```
?- not groom(george).
```

To check that Mary has an unmarried son born in Boston, we model the concept of an unmarried son born in a certain city:

```
umsbb(M,X,C):-son(X,M), birth_city(X,C), not married(X,Y,T).
```

then pose the query:

```
?- umsbb(mary,X,boston).
```

Likewise, to check that no woman is the wife of any of Fred's children, we define the rule:

```
fred_child_wife(Y):-child(X,fred), married(X,Y,T), female(Y).
```

then ask the query:

```
?- not fred_child_wife(X).
```

All of the above queries succeed. This example illustrates how close the ASP rules are to the commonsense knowledge that is to be represented.

9. Applications

Goal-directed ASP systems s(ASP) and s(CASP) have been used for many innovative applications. Some of these are described below. Many more such applications are possible.

Undergraduate Degree Audit: Determining if a student can graduate with an undergraduate degree at a US University is a hard problem. Typically, the criteria for graduation are laid down as a set of rules in the University’s undergraduate catalog. These rules can change every year. An example rule for the mathematics major is as follows:

Nine semester credit hours of upper-division MATH, STAT or ACTS courses, at least six of which must be MATH courses at the 4000-level. These courses cannot include those for which the catalog entry states: May not be used to satisfy mathematics requirements by students in Mathematics.

Rules may involve negation and may be non-deterministic. A mistake made in interpreting or applying a rule can delay a student’s graduation. An ASP-based degree audit system has been developed by us for CS, Math and Statistics majors. If a student can’t graduate, relational nature of ASP will allow the computation—without any additional effort—of list of potential courses that the student must take that will allow him/her to graduate. A major advantage of using ASP is that as the catalog is changed, importing these changes into the system is relatively easy.

Modeling Expert Knowledge: Human expert knowledge can be modeled in ASP. In collaboration with cardiologists, we have developed the CHEF system for automated treatment of congestive heart failure (CHF) disease. The system has been developed with the s(ASP) predicate ASP system. [33]. The treatment of CHF is guideline-driven. For patients in USA, the American College of Cardiology has developed these guidelines that consist of about 100 odd rules stated in English in an 87 page document [34]. An example rule is as follows: *In all patients with a recent or remote history of MI or ACS and reduced EF, ACE inhibitors should be used to prevent symptomatic HF and reduce mortality. In patients intolerant of ACE inhibitors, ARBs are appropriate unless contraindicated.* Note that contraindication of a drug is elegantly modeled via NAF. In the studies we have done, our system is able to perform as well as a trained physician treating CHF. In some cases, it can outperform a physician who, for example, may forget to look up the latest lab-test values or may misremember it.

Legal Reasoning: The formal representation of a legal text to automatize reasoning about them is well known in literature, and is recently gaining much attention thanks to the interest in Rules as Code [35], a public administration methodology that ideally requires rules to be drafted in natural language and represented in executable languages to facilitate autonomous decisions by public administrations [36, 37, 38]. For deterministic rules there are several proposals, often based on logic-based programming languages. However, none of the existing proposals are able to represent the ambiguity and/or administrative discretion present in contracts and/or applicable legislation, e.g., *force majeure*.³ We have developed a framework based on s(CASP), called s(LAW) [39], that allows for modeling legal rules involving ambiguity, and supports reasoning and inferring conclusions based on them. Additionally, Blawx [40], a web-based platform available at <https://github.com/lexpedite/blawx>, also based on s(CASP), has been developed by the needs of the Rules as Code Programme at the Canada School of Public Service, which is supporting departments across the Government of Canada interested in exploring the

³Force majeure is a law term that must be understood as referring to abnormal and unforeseeable circumstances.

Rules as Code methodology. Thanks to the goal-directed execution of s(CASP) both proposals provide justification [31] of the resulting conclusions (in natural language).

Modeling Cyberphysical Systems: Goal-directed ASP systems such as s(CASP) can also be used to elegantly model cyber-physical systems' behaviour using the event calculus [41]. Event calculus models systems that are evolving and involve *events* and *fluents*. System behavior is captured by predicates such as *happens(E,T)* which state that the event *E* happens at time *T* and *holdsAt(F,T)* which states that fluent *F* holds at time *T*, where a fluent describes the state of an evolving system. In the context of cyber-physical systems, fluents correspond to state of the sensors or components while the events represent actuator actions. With the event calculus realized in s(CASP) [41], which provides support for reasoning over real time, requirements specifications for cyber physical systems can be directly represented in s(CASP). As a result, they are executable, and their safety and liveness properties can be studied [42].

(Visual) Question Answering: There has been significant amount of work in developing automated question-answering systems given an image. Current approaches are based on machine learning that train on a collection of images together with the question-answer pairs for each image. Once the model has been learned, a new image along with a question is given, and the expectation is that a correct answer will be generated. Given that a generated answer cannot be justified, or may not be accurate, an alternative approach is to use machine learning for translating an image into a set of predicates that capture the objects in the image, their characteristics, and their spatial relationships. Commonsense knowledge about the domain that the image is about can be coded in ASP. A question can be translated into an ASP query as well. This query is then executed using s(CASP) against the knowledge captured from the image as predicates and commonsense knowledge that we coded about the domain of the image. 100% accuracy in answering questions is achieved in some of the machine learning datasets [43]. A similar approach has been applied to answering questions about a text passage [44]. Thus, s(CASP) allows for approaches to AI that use both learning and reasoning, very similar to how humans operate.

Software Certification: Software certification refers to assuring that a software system's risk is acceptable. The certification process is carried by a human expert using frameworks such as claim-arguments-evidence framework [45]. The CAE framework maps directly to ASP, where a claim is a query, an argument a rule, and evidence a fact. The CAE framework has a notion of a *defeater* that corresponds to a NAF goal. Such a mapping has been done and added to the ASCE tool by Adelard PLC. Once a certifier develops an assurance case on ASCE, they can automatically map it to ASP code. The query corresponding to the claim can be executed on s(CASP) to check for the consistency of the assurance case. If the assurance case fails, negation of the query can be executed to determine the cause of the inconsistency. The ASP encoding essentially models the mind of the software certifier.

Logic-based Learning: A goal-directed implementation of ASP also leads to a logic-based framework for reinforcement learning. Suppose we learn a theory from observations and examples, e.g., if *X* is a bird then it flies. Then we encounter Tweety the penguin. Our conclusion that *flies(tweety)* fails, when we were expecting it to succeed. At this point we can learn that penguins are exceptional birds that do not fly. This is done by examining the failed proof

and repairing the program so that the repaired program allows us to draw the correct conclusion. This technique has been applied to learning and updating playing strategies in games [46].

10. Related Work

Significant amount of work has been done in answer set programming. Modeling commonsense knowledge and reasoning in ASP has been extensively covered in the literature [20, 15]. In particular, the book by Kahl and Gelfond has extensive coverage of default rules, exceptions, preferences, constraints, possible worlds, and other related topics for knowledge representation and reasoning. However, most of the work assumes SAT-solver based implementations of ASP which limits its applications. With a goal-directed implementation such as s(CASP), many of these limitations are removed. A short tutorial on Answer Set Programming can be found elsewhere [21], so we do not cover the details here. With respect to goal-directed implementation of answer set programming, there are other efforts, but none as advanced as the s(CASP) system. A description of these systems is not included here due to lack of space, but can be found elsewhere [17]. Goal-directed implementation of ASP was also pursued under the thread of abductive logic programming, however, they are restricted in one way or the other, e.g., only propositional programs are allowed, or programs should be *range restricted* and free of odd loops over negation rules, the query must be ground, etc. [47, 48, 49]. The s(ASP) and s(CASP) systems overcome these problems by employing constructive negation and coinduction.

There has been significant amount of work on developing expert systems since the 80s. Expert systems reason through knowledge that is represented declaratively as if-then rules rather than in a procedural manner. There are two problems with this if-then rule formulation wrt modeling human expertise:

1. The goal of expert systems technology was to model expert knowledge, however, as discussed earlier, simple if-then rules are inadequate for modeling human expertise. Humans also use co-inductive reasoning that is not easily modeled in traditional languages that limit themselves to inductive reasoning (cf. the “Who will go to Mexico?” example above).
2. If-then rules are only used for deductive reasoning, and as discussed earlier, humans use abductive reasoning as well.

Answer set programming allows human-style commonsense reasoning to be modeled more faithfully, something that the formalisms employed by earlier expert systems did not or could not. Our experience with the CHeF system shows that ASP can model the human thought process quite accurately. Thus, ASP can be thought of as a better formalism for building expert systems. In addition, the s(CASP) proof tree for a failed query can be examined and failure(s) cured by employing additional knowledge, giving rise to logic-based learning. Explanation based learning, where we generalize from a single example, and then refine the rule(s) as we encounter more information can also be modeled with ASP and s(CASP). The process is very similar to what a human may follow for reasoning and learning in his/her day-to-day life.

Cyc [50] is one of the oldest AI project that attempts to model commonsense reasoning. In Cyc, knowledge is presented in the form of a vast collection of ontologies that consist of implicit

knowledge and rules about the world that represents commonsense knowledge. The ontology of Cyc as of 2017 contains around 1,500,000 terms. This includes around 500,000 collections, 50,000+ predicates and around a million well known entities. Cyc uses a *community of agents* consisting of multiple reasoning agents that rely on more than 1000 heuristic modules to solve inference problems. A potential problem with Cyc is knowing which agent to apply, and which heuristic to use. For a commonsense reasoning system to be successful, *it has to be modeled in a very simple way and very simple inference processes must be used*. Otherwise, it will be hard to determine which decision procedure to apply. In the ASP approach, we rely only on defaults and exceptions, constraints and multiple possible worlds to emulate the human thought process. If we use complex reasoning processes then we may possess the individual pieces of knowledge to answer a given question, but may not be able to compose these pieces together appropriately to arrive at an answer. Thus, simplicity of inference procedures is extremely important. Most commonsense reasoning tasks involves just making a small number of inferences.

11. Conclusion

The main contribution of this paper is to show a path towards emulating human intelligence, one of the most profound challenges in science. The paradigm of logic programming/ASP and the s(CASP) system are a step in the direction of developing the machinery to construct human-level intelligence. The critical pieces needed are ability to handle incomplete information, circular or assumption-based reasoning, and representing multiple possible worlds. We argued that the dogma of permitting only inductive semantics from the time of Russell has been a major hindrance to automating commonsense reasoning. We showed how both deductive *and* abductive reasoning can be elegantly modeled, and how default rules are an elegant way of representing inductive generalizations in the sense of machine learning. Development of coinductive semantics [7, 11, 8], the paradigm of answer set programming, and the s(CASP) system pave the way for faithfully modeling the human thought process, we believe. A significant amount of knowledge representation and reasoning work that we report on has been done by the ASP community. We recast this work through the lens of coinductive reasoning and goal-directed, Prolog-like execution.

While ASP declaratively represents knowledge in the form that humans seem to represent, the s(CASP) system can be thought of as providing an operational semantics to the human thought process. This opens up interesting possibilities. For instance, a more efficient procedure can be derived by *partial evaluation* of the s(CASP) program that represents knowledge for a specific task. Thus, it should be possible to declaratively represent knowledge about concurrency, data structures and mutual exclusion, and then automatically derive efficient concurrent algorithms for various data structures by partial evaluation or similar suitable program transformation [51]. Given that ASP and s(CASP) can represent the human thought process, it should be possible to automate any task, as long as a human can perform it and the knowledge involved can be represented as axioms in ASP.

However, many challenges still remain:

- Humans acquire commonsense knowledge over a long period of time. The first challenge is how do we create this commonsense knowledgebase? It will be ideal if it is created

automatically. At present, to develop a commonsense reasoning application using s(CASP), knowledge has to be explicitly represented and the rules have to be explicitly coded in (predicate) ASP. Inferences are then made (i.e., queries executed) using the s(CASP) system. As long as the domain is narrow, it is possible to accomplish this. Scaling it up to larger domains and large amount of commonsense knowledge is the next challenge.

- Humans can hear or read natural language sentences or look at a picture and store the knowledge depicted in the sentence or the picture in their minds. Then, they use this information to reason in conjunction with their commonsense knowledge. The second challenge, thus, is how do we extract the knowledge in text, dialogs, and pictures and represent it as predicates? Once this knowledge is represented as predicates, then we can use automated commonsense reasoning to draw inferences just like humans do. Perhaps machine learning techniques can be used to extract knowledge represented as ASP predicates, much in the same manner as language translators from one natural language to another are built, except that the target language here will be that of ASP predicates.
- The s(CASP) system performs backward chaining. When attempting to prove a goal, humans follow a similar process. However, as various concepts are utilized in the goal-driven proof tree generation, humans may recall other concepts related to the concepts used in the proof. These concepts may be used, for example, to advance a conversation. For example, if I am responding to a question from someone about my favorite actor in the movie ‘Titanic’, then the conversation may shift towards the movie’s director. This is because the concept of a movie’s director and actor are closely related, even though the movie’s director is not directly involved in the answer to the question. We believe that such local forward chaining—using knowledge that connects related concepts—is needed to build, for example, conversational socialbots based on commonsense reasoning [52].

Once the above challenges are overcome, really advanced applications such as chatbots that can understand and answer like a human, autonomous driving systems [18], etc., can become possible.

References

- [1] B. Gillon, Logic in Classical Indian Philosophy, in: The Stanford Encycl. of Philosophy, fall 2016 ed., Metaphysics Rsrch Lab, Stanford Univ., 2016.
- [2] S. Bobzien, Ancient Logic, in: E. N. Zalta (Ed.), The Stanford Encyclopedia of Philosophy, Metaphysics Rsrch Lab, Stanford Univ., 2020.
- [3] G. Boole, An Investigation of the Laws of Thought, Walton & Maberly, 1854.
- [4] G. Frege, Grundlagen der Arithmetik, Wilhelm Koebner, 1884.
- [5] Handbook of logic and language, ????
- [6] J. Barwise, L. A. Moss, Vicious Circles, Cambridge University Press, 1996.
- [7] P. Aczel, Non-well-founded sets, volume 14 of *CSLI lecture notes series*, CSLI, 1988.
- [8] G. Gupta, A. Bansal, R. Min, L. Simon, A. Mallya, Coinductive logic programming and its applications, in: Proc. 23rd ICLP 2007, volume 4670 of *LNCS*, Springer, 2007, pp. 27–44.

- [9] A. Tarski, On undecidable statements in enlarged systems of logic and the concept of truth, *J. Symb. Log.* 4 (1939) 105–112.
- [10] S. Kripke, An outline of a theory of truth, *The Journal of Philosophy* Col. LXXII, No. 19, Nov. 6, 1975, pp. 690-716 (????).
- [11] L. Simon, A. Mallya, A. Bansal, G. Gupta, Coinductive logic programming, in: *Proc. ICLP'06*, volume 4079 of *LNCS*, Springer, 2006, pp. 330–345.
- [12] J. W. Lloyd, *Foundations of Logic Programming*, 2nd Edition, Springer, 1987.
- [13] K. R. Apt, R. N. Bol, Logic programming and negation: A survey, *J. Log. Program.* 19/20 (1994) 9–71.
- [14] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming., in: *ICLP/SLP*, volume 88, 1988, pp. 1070–1080.
- [15] M. Gelfond, Y. Kahl, *Knowledge representation, reasoning, and the design of intelligent agents: Answer Set Programming approach*, Cambridge Univ. Press, 2014.
- [16] K. Marple, E. Salazar, G. Gupta, Computing stable models of normal logic programs without grounding, *arXiv preprint arXiv:1709.00501* (2017).
- [17] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint answer set programming without grounding, *TPLP* 18 (2018) 337–354. doi:10.1017/S1471068418000285.
- [18] S. Kothawade, V. Khandelwal, K. Basu, H. Wang, G. Gupta, AUTO-DISCERN: autonomous driving using common sense reasoning, in: *Proc. ICLP Workshops: GDE'21*, volume 2970 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021.
- [19] P. Johnson-Laird, *How We Reason*, Oxford University Press, 2006.
- [20] C. Baral, *Knowledge representation, reasoning and declarative problem solving*, Cambridge University Press, 2003.
- [21] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, *Commun. ACM* 54 (2011) 92–103.
- [22] R. Reiter, A logic for default reasoning, *Artif. Intell.* 13 (1980) 81–132.
- [23] H. Wang, G. Gupta, FOLD-R++: A scalable toolset for automated inductive learning of default theories from mixed data, in: *Proc. FLOPS'22*, volume 13215 of *LNCS*, Springer, 2022, pp. 224–242.
- [24] M. L. Ginsberg, *Essentials of Artificial Intelligence*, Morgan Kaufmann, 1993.
- [25] I. Douven, Abduction, in: E. N. Zalta (Ed.), *The Stanford Encycl. of Philosophy*, 2011.
- [26] T. Chen, C. Guestrin, XGBoost: A scalable tree boosting system, in: *Proc. 22nd ACM SIGKDD, KDD '16*, 2016, pp. 785–794.
- [27] C. C. Aggarwal, *Neural Networks and Deep Learning - A Textbook*, Springer, 2018.
- [28] F. Lin, Y. Zhao, Assat: Computing answer sets of a logic program by sat solvers, *Artificial Intelligence* 157 (2004) 115–137.
- [29] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, M. Schneider, Potassco: The potsdam answer set solving collection, *Ai Communications* 24 (2011) 107–124.
- [30] J. J. Alferes, L. M. Pereira, T. Swift, Abduction in well-founded semantics and generalized stable models via tabled dual programs, *Theory Pract. Log. Program.* 4 (2004) 383–428.
- [31] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for Goal-Directed Constraint Answer Set Programming, in: *Proceedings 36th ICLP (Technical Communications)*, volume 325 of *EPTCS*, 2020, pp. 59–72. doi:10.4204/EPTCS.325.12.
- [32] H. J. Levesque, *Thinking as Computation: A First Course* by, MIT Press, Cambridge, MA.,

2012.

- [33] Z. Chen, K. Marple, E. Salazar, G. Gupta, L. Tamil, A physician advisory system for chronic heart failure management based on knowledge patterns, *Theory Pract. Log. Program.* 16 (2016) 604–618.
- [34] C. W. Yancey, M. Jessup, et al., Accf/aha guideline for the management of heart failure, *Circulation* 28(16) (2013) e240–e327.
- [35] J. Mohun, A. Roberts, *Cracking the code: Rulemaking for humans and machines* (2020).
- [36] A. Cerrillo i Martínez, El derecho para una inteligencia artificial centrada en el ser humano y al servicio de las instituciones: Presentación del monográfico., *IDP: Revista de Internet, Derecho y Política* (2019).
- [37] J. Cobbe, Administrative law and the machines of government: judicial review of automated public-sector decision-making, *Legal Studies* 39 (2019) 636–655.
- [38] J. P. Solé, Inteligencia artificial, derecho administrativo y reserva de humanidad: algoritmos y procedimiento administrativo debido tecnológico, *Revista general de Derecho administrativo* 50 (2019).
- [39] J. Arias, M. Moreno-Rebato, J. A. Rodríguez-García, S. Ossowski, Modeling Administrative Discretion Using Goal-Directed Answer Set Programming, in: *Advances in Artificial Intelligence, CAEPIA 20/21*, Springer International Publishing, Cham, 2021, pp. 258–267. doi:10.1007/978-3-030-85713-4_25.
- [40] J. Morris, Constraint answer set programming as a tool to improve legislative drafting: a rules as code experiment, in: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Law, 2021*, pp. 262–263.
- [41] J. Arias, M. Carro, Z. Chen, G. Gupta, Modeling and reasoning in event calculus using goal-directed constraint answer set programming, *Theory and Practice of Logic Programming* 22 (2022) 51–80. doi:10.1017/S1471068421000156.
- [42] S. C. Varanasi, J. Arias, E. Salazar, F. Li, K. Basu, G. Gupta, Modeling and verification of real-time systems with the event calculus and s(casp), in: *Proc. PADL'22*, volume 13165 of *LNCS*, Springer, 2022, pp. 181–190.
- [43] K. Basu, F. Shakerin, G. Gupta, AQuA: ASP-Based Visual Question Answering, in: *Proc. PADL, Springer, Cham, 2020*, pp. 57–72.
- [44] K. Basu, S. C. Varanasi, F. Shakerin, J. Arias, G. Gupta, Knowledge-driven natural language understanding of english text and its applications, in: *Proc. AAAI'21*, AAAI Press, 2021, pp. 12554–12563.
- [45] R. Bloomfield, J. Rushby, *Assurance 2.0*, CoRR abs/2004.10474 (2020).
- [46] K. Basu, K. Murugesan, M. Atzeni, P. Kapanipathi, K. Talamadupula, T. Klinger, M. Campbell, M. Sachan, G. Gupta, A hybrid neuro-symbolic approach for text-based games using inductive logic programming, *AAAI Workshop on Combining Logic and Reasoning (CLeaR)* (2022).
- [47] A. C. Kakas, R. A. Kowalski, F. Toni, Abductive logic programming, *J. Log. Comput.* 2 (1992) 719–770.
- [48] A. Saptawijaya, L. M. Pereira, Tabled abduction in logic programs, *Theory Pract. Log. Program.* 13 (2013).
- [49] K. Satoh, N. Iwayama, A query evaluation method for abductive logic programming, in: *Proc. Joint International Conference and Symposium on Logic Programming, JICSLP 1992*,

MIT Press, 1992, pp. 671–685.

- [50] D. B. Lenat, Cyc: A large-scale investment in knowledge infrastructure, *Communications of the ACM* 38 (1995) 33–38.
- [51] S. C. Varanasi, N. Mittal, G. Gupta, Generating concurrent programs from sequential data structure knowledge using answer set programming, in: *Proc. ICLP'21 (Tech. Comm.)*, volume 345 of *EPTCS*, 2021, pp. 219–233.
- [52] F. Li, H. Wang, K. Basu, E. Salazar, G. Gupta, Discasp: A graph-based ASP system for finding relevant consistent concepts with applications to conversational socialbots, in: *Proc 37th ICLP (Tech. Comm)*, volume 345 of *EPTCS*, 2021, pp. 205–218.