GK: Implementing Full First Order Default Logic for Commonsense Reasoning (System Description)

Tanel Tammet[1(✉)], Dirk Draheim[2], and Priit Järv[1]

[1] Applied Artificial Intelligence Group, Tallinn University of Technology, Tallinn, Estonia
{tanel.tammet,priit.jarv1}@taltech.ee
[2] Information Systems Group, Tallinn University of Technology, Tallinn, Estonia
dirk.draheim@taltech.ee

Abstract. Our goal is to develop a logic-based component for hybrid – machine learning plus logic – commonsense question answering systems. The paper presents an implementation GK of default logic for handling rules with exceptions in unrestricted first order knowledge bases. GK is built on top of our existing automated reasoning system with confidence calculation capabilities. To overcome the problem of undecidability of checking potential exceptions, GK performs delayed recursive checks with diminishing time limits. These are combined with the taxonomy-based priorities for defaults and numerical confidences.

1 Introduction

The problem of handling uncertainty is one of the critical issues when considering the use of logic for automating commonsense reasoning. Most of the facts and rules people use in their daily lives are uncertain. There are many types of uncertainty, like fuzziness (is a person somewhat tall or very tall), confidence (how certain does some fact seem) and exceptions (birds can typically fly, but penguins, ostriches etc., can not). Some of these uncertainties, like fuzziness and confidence, can be represented numerically, while others, like rules with exceptions, are discrete. In [18] we present the design and implementation of the CONFER framework for extending existing automated reasoning systems with confidence calculation capabilities. In the current paper we present the implementation called GK for default logic [13], built by further extending the CONFER implementation. Importantly, we design a novel practical framework for implementing default logic for the full, undecidable first order logic on the basis of a conventional resolution prover.

1.1 Default Logic

Default logic was introduced in 1980 by R. Reiter [13] to model one aspect of common-sense reasoning: rules with exceptions. It has remained one of the most

J. Blanchette et al. (Eds.): IJCAR 2022, LNAI 13385, pp. 300–309, 2022.
https://doi.org/10.1007/978-3-031-10769-6_18

well-known logic-based mechanisms devoted to this goal, with the *circumscription* by J. McCarthy and the *autoepistemic logic* being the early alternatives. Several similar systems have been proposed later, like defeasible logic [11].

Default logic [13] extends classical logic with default rules of the form

$$\frac{\alpha(x) : \beta_1(x), ...\beta_n(x)}{\gamma(x)}$$

where a *precondition* α(x), *justifications* $\beta_1(x), ...\beta_n(x)$ and a *consequent* $\gamma(x)$ are first order predicate calculus formulas whose free variables are among $x = x_1, ..., x_m$. For every tuple of individuals $t = t_1, ..., t_n$, if the precondition $\alpha(t)$ is derivable and none of the *negated* justifications $\neg\beta(t)$ are derivable from a given knowledge base KB, then the consequent $\gamma(t)$ can be derived from KB. Differently from classical and most other logics, default logic is *non-monotonic*: adding new assumptions can make some previously derivable formulas non-derivable.

As investigated in [7], the interpretation of quantifiers in default rules can lead to several versions of default logic. We follow the original interpretation of Reiter in [13] which requires the use of Skolemization in a specific manner over default rules. For example, a default rule: $\exists x P(x) \vdash \exists x P(x)$ should be interpreted as : $P(c) \vdash P(c)$, where c is a Skolem constant.

Consider a typical example for default logic: birds can normally fly, but penguins cannot fly. The classical logic part

$$penguin(p)\ \&\ bird(b)\ \&\ \forall x.penguin(x) \Rightarrow bird(x)\ \&\ \forall x.penguin(x) \Rightarrow \neg fly(x).$$

is extended with the default rule $bird(x) : fly(x) \vdash fly(x)$. From here we can derive that an arbitrary bird b can fly, but a penguin p cannot. The default rule cannot be applied to p, since a contradiction is derivable from $fly(p)$. This argument cannot be easily modelled using numerical confidences: the probability of an arbitrary living bird being able to fly is relatively high, while the penguins form a specific subset of birds, for which this probability is zero.

Another well-known example – Nixon's triangle – introduces the problem of multiple extensions and *sceptical* vs *credulous* entailment: the classical facts $republican(nixon)$ & $quaker(nixon)$ extended with two mutually excluding default rules $republican(x) : \neg pacifist(x) \vdash \neg pacifist(x)$ and $quaker(x) : pacifist(x) \vdash pacifist(x)$. The credulous entailment allows giving different priorities to the default rules and accepts different sets (*extensions*) of consequences, if there is a way to assign priorities so that all the consequences in an extension can be derived. The sceptical entailment requires that a consequence is present in all extensions. GK follows the latter interpretation, but allows explicit priorities to be assigned to the default rules.

The concept of *priorities* for default rules has been well investigated, with several mechanisms proposed. G. Brewka argues in [4] that "for realistic applications involving default reasoning it is necessary to reason about the priorities of defaults" and introduces an ordering of defaults based on specificity: default rules for a more specific class of objects should take priority over rules for more general classes. For example, since birds (who typically do fly) are physical objects

and physical objects typically do not fly, we have contradictory default rules describing the flying capability of arbitrary birds. Since birds are a subset of physical objects, the flying rule of birds should have a higher priority than the non-flying rule of physical objects.

1.2 Undecidability, Grounding and Implementations

Perhaps the most significant problem standing in the way of automating default logic is undecidability of the applicability of rules. Indeed, in order to apply a default rule, we must prove that the justifications do not lead to a contradiction with the rest of the knowledge base KB. For full first order logic this is undecidable. Hence, the standard approach for handling default logic has been creating a large ground instance KB_g of the KB, and then performing decidable propositional reasoning on the KB_g.

Almost all the existing implementations of default logic like DeReS [5], DLV2 [1] or CLINGO [8], with the noteworthy exception of s(CASP) [2], follow the same principle. More generally, the field of *Answer Set Programming* (ASP), see [10], is devoted to this approach. As an exception, the s(CASP) system [2] solves queries without the grounding step and is thus better suited for large domains. It is noteworthy that the s(CASP) system has been used in [9] for automating common sense reasoning for autonomous driving with the help of default rules. However, s(CASP) is a logic programming system, not a universal automated reasoner. For example, when we add a rule `bird(father(X)) :- bird(X)` to the formulation of the above birds example in s(CASP), the search does not terminate, apparently due to the infinitely growing nesting of terms.

While ASP systems are very well suited for specific kinds of problems over a small finite domain, grounding becomes infeasible for large first order knowledge bases (*KB* in the following), in particular when the domain is infinite and nested terms can be derived from the KB. The approach described in this paper accepts the lack of logical omniscience and performs delayed recursive checking of exceptions with diminishing time limits directly on non-grounded clauses, combined with the taxonomy-based priorities for defaults and numerical confidences.

2 Algorithms

Our approach of implementing default rules in GK for first order logic is to delay justification checking until a first-order proof is found and then perform recursively deepening checks with diminishing time limits. Thus, our system first produces a potentially large number of different candidate proofs and then enters a recursive checking phase. The idea of delaying justification checking is already present in the original paper of R. Reiter [13], where he uses linear resolution and delayed checks as the main machinery of his proofs. The results produced by GK thus depend on the time limits and are not stable. Showing specific fixpoint properties of the algorithm is not in the scope of our paper.

A practical question for implementation is the actual representation of default rules and making the rules fit the first-order proof search machinery. To this end we introduce *blocker atoms* which are similar to the justification indexes of Reiter.

In the following we will assume that the underlying first order reasoner uses the resolution method, see [3] for details. The rest of the paper assumes familiarity with the basic concepts, terminology and algorithms of the resolution method.

2.1 Background: Queries and Answers

We assume our system is presented with a question in one of two forms: *(1)* Is the statement Q true? *(2)* Find values V for existentially bound variables in Q so that Q is true. For simplicity's sake we will assume that the statement Q is in the prefix form, i.e., no quantifiers occur in the scope of other logical connectives.

In the second case, it could be that several different value vectors can be assigned to the variables, essentially giving different answers. We also note that an answer could be a disjunction, giving possible options instead of a single definite answer.

A widely used machinery in resolution-based theorem provers for extracting values of existentially bound variables in Q is to use a special *answer predicate*, converting a question statement Q to a formula $\exists X(Q(X)\&\neg answer(X))$ for a tuple of existentially quantified variables X in Q [6]. Whenever a clause is derived which consists of only answer predicates, it is treated as a contradiction (essentially, answer) and the arguments of the answer predicate are returned as the values looked for. A common convention is to call such clauses *answer clauses*. We will require that the proof search does not stop whenever an answer clause is found, but will continue to look for new answer clauses until a predetermined time limit is reached. See [16] for a framework of extracting multiple answers.

We also assume that queries take a general form $(KB\&A) \Rightarrow Q$ where KB is a commonsense knowledge base, A is an optional set of precondition statements for this particular question and Q is a question statement. The whole general query form is negated and converted to clauses, i.e., disjunctions of literals (positive or negative atoms). We will call the clauses stemming from the question statement *question clauses*.

2.2 Blocker Atoms and Justification Checking

Without loss of generality we assume that the precondition and consequent formulas α and γ in default rules are clauses and justifications $\beta_1, ..., \beta_n$ are literals, i.e. positive or negative atoms: $\alpha : \beta_1, ...\beta_n \vdash \gamma$. Complex formulas can be encoded with a new predicate over the free variables of the formula and an equivalence of the new atom with the formula. Recall that Reiter assumes that the default rules are Skolemized.

We encode a default rule as a clause by concatenating into one clause the precondition and consequent clauses $\alpha(x)$ and $\gamma(x)$ and blocker atoms $block(\neg\beta_1)$,

..., $block(\neg\beta_n)$ where each justification β_i is either a positive or a negative atom. The negation $\neg$ is used since we prefer to speak about *blockers* and not *justificatons*. For example, the "birds can fly" default rule is represented as a clause

$$\neg\texttt{bird(X)} \vee \texttt{fly(X)} \vee \texttt{block(0,neg(fly(X)))}$$

where `X` is a variable and `neg(fly(X))` encodes the negated justification. The first argument of the blocker (0 above) encodes priority information covered in the next section.

A proof of a question clause is a clause containing only answer atoms and blocker atoms. In the justification checking phase the system attempts to prove each decoded second blocker argument $\neg\beta_i$ in turn: the proof is considered invalid if some of $\neg\beta_i$ can be proved and this checking-proof itself is valid. If we pose a question $\texttt{fly(X)} \Rightarrow \texttt{answer(X)}$ to the system to be proved (see the earlier example), we get two different answers: $\texttt{answer(p)} \vee \texttt{block(neg(fly(p))}$ and $\texttt{answer(b)} \vee \texttt{block(neg(fly(b))}$. Checking the first of these means trying to prove $\neg\texttt{fly(p)}$ which succeeds, hence the first answer is invalid. Checking the second answer we try to prove $\neg\texttt{fly(b)}$ which fails, hence the answer is valid.

Notice that the contents $\neg\beta_i$ of blockers, just like answer clauses, have a role of collecting substitutions during the proof search: this enables us to disregard the order in which the clauses are used, i.e. both top-down, bottom-up and mixed proof search strategies can be used.

Importantly, blockers are used during the subsumption checks similarly to ordinary literals. A clause C_1 with fewer or more general literals than C_2 is hence always preferred to C_2, given that (a) the literals of C_1 subsume C_2, disregarding the priority arguments of blockers, and (b) the priority arguments of corresponding blocker literals in C_1 are equal or stronger than these of C_2. When combined with the uncertainty and inconsistency handling mechanisms of CONFER, the subsumption restrictions of the latter also apply. There are also other differences to ordinary literals. First, we prohibit the application of equality (demodulation or paramodulation) to the contents of blocker atoms during proof search. Second, we discard clauses containing mutually contradictory blockers (assuming the decoding of the second argument) like we would discard ordinary tautologies.

2.3 Priorities, Recursion and Infinite Branches

Default rule priorities are critical for the practical encoding of commonsense knowledge. The usage of priorities in proof search is simple: when checking a blocker with a given priority, it is not allowed to use default rules with a lower priority. We encode priority information as a first argument of the blocker literal, offering several ways to determine priority: either as an integer, a taxonomy class number, a string in a taxonomy or a combination of these with an integer.

For automatically using specificity we employ taxonomy classes: a class has a higher prirority than those above it on the taxonomy branch. We have built a topologically sorted acyclic graph of English words using the WordNet taxonomy

along with an efficient algorithm for quick priority checks during proof search. Taxonomy classes are indicated with a special term like $(`61598`). Alternatively one can use an actual English word like $("`bird`") which is automatically recognized to be more specific than, say, $("`object`"). To enable more fine-grained priorities, an integer can be added to the term like $("`bird`", `2`) generating a lexicographic order.

The recursive check for the non-provability of blockers can go arbitrarily deep, except for the time limits. Our algorithm allocates N seconds for the whole proof search and spends half of N for looking for different proofs and answers for the query, with the other half split evenly for each answer. Again, the time allocated for checking an answer is split evenly between the blockers in the answer. Each such time snippet is again split between a search for the proof of the blocker, and if found, for recursively checking the validity of this proof. Once the allocated time is below a given threshold (currently one millisecond) the proof is assumed to be not found.

Answers given by the system depend on the amount of time given, the search strategy chosen etc. For example, consider the Nixon triangle presented earlier, with two contradictory default rules. In case the priorities of these rules are equal and we allow defaults with the same priority to be used for checking an answer containing a blocker, the recursive check terminates only because of a time limit, which is unpredictable. Hence, we may sometimes get one answer and sometimes another. In order to increase both stability and efficiency, GK checks the blockers in the search nodes above, and terminates with failure in cases nonterminating loops are detected. Therefore GK always gives a sceptical result to the Nixon triangle: neither $pacifist(nixon)$ nor $\neg pacifist(nixon)$ is proven.

3 Confidences and Inconsistencies

GK integrates the exception-handling algorithms described in the previous chapter with the algorithms designed for handling inconsistent KB-s and numeric confidences assigned to clauses, previously presented as a CONFER framework in [18]. The framework is built on the resolution method. It calculates the estimates for the confidences of derived clauses, using both (a) the decreasing confidence of a conjunction of clauses as performed by the resolution and paramodulation rule, and (b) the increasing confidence of a disjunction of clauses for cumulating evidence. CONFER handles inconsistent KB-s by requiring the proofs of answers to contain the clauses stemming from the question posed. It performs searches both for the question and its negation and returns the resulting confidence calculated as a difference of the confidences found by these two searches.

The integrated algorithm is more complex than the one we previously described. Whenever the algorithms of the previous chapter speak about "proving", the system actually performs two independent searches – one for the positive and one for the negated goal – with the confidences calculated for both of these. A blocker is considered to be proved in case the resulting confidence is over a pre-determined configurable threshold, by default 0.5. Blocker proofs

must also contain the clause built from the blocker. Thus, the whole search tree for a query consists of two types of interleaved layers: positive/negative confidence searches and blocker checking searches, the latter type potentially making the tree arbitrarily deep up to the minimal time limit threshold.

4 Implementation and Experiments

The described algorithms are implemented by the first author as a software system GK available at https://logictools.org/gk/. GK is written in C on top of our implementation of the CONFER framework [18] which is built on top of a high-performance resolution prover GKC [17] (see https://github.com/tammet/gkc) for conventional first order logic. Thus GK inherits most of the capabilities and algorithms of GKC.

A tutorial and a set of default logic example problems along with proofs from GK are also available at http://logictools.org/gk. GK is able to quickly solve nontrivial problems built by extending classic default logic examples. It is also able to solve classification problems combining exception and cumulative evidence and problems with dynamic situations using fluents, including planning problems. We have built a very large integrated knowledge base from the Quasimodo [14] and ConceptNet [15] knowledge bases, converting these to default logic plus confidences. GK is able to solve simple problems using this large knowledge base along with the Wordnet taxonomy for specificity: see the referenced web page for examples.

The following small example illustrates the fundamental difference of GK from the existing ASP systems for default logic. The standard penguins and birds example presented above in the ASP syntax is

```
bird(b1).
penguin(p1).
bird(X) :- penguin(X).
flies(X) :- bird(X), not -flies(X).
-flies(X) :- penguin(X).
```

Both GK and the ASP systems clingo 5.4.0, dlv 2.1.1 and s(CASP) 0.21.10.09 give an expected answer to the queries `flies(b1)` and `flies(p1)`. However, when we add the rules

```
bird(father(X)) :- bird(X).
penguin(father(X)) :- penguin(X).
```

none of these ASP systems terminate for these queries, while GK does solve the queries as expected. Notably, as pointed out by the author of s(CASP), this system does terminate for the reformulation of the same problem with the two replacement rules

```
flies(X) :- bird(X), not abs(X).
abs(X) :- penguin(X).
```

while clingo and dlv do not terminate. When we instead add the facts and rules

```
father(b1,b2).
father(p1,p2).
...
father(bN-1,bN).
father(pN-1,pN).

ancestor(X,Y):- father(X,Y).
ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).
```

for a large N, s(CASP) does not terminate and clingo and dlv become slow for `flies(b1)`: ca 8 s for $N = 500$ and ca 1 min for $N = 1000$ on a laptop with a 10-th generation i7 processor. GK solves the same question with $N = 1000$ under half a second and with $N = 100000$ under three seconds: the latter problem size is clearly out of scope of the capabilities of existing ASP systems.

We have previously shown that the confidence handling mechanisms in CONFER may slow down proof search for certain types of problems, but do not have a strong negative effect on very large commonsense CYC [12] problems in the TPTP problem collection. Differently from CONFER, the algorithms for default logic described above do not substantially modify the resolution method implementation of pure first order logic search, thus the performance of these parts of GK are mostly the same as of GKC. The ability to give a correct answer to a query during a given time limit depends on the performance of these components, and not on the overall recursively branching algorithm.

5 Summary and Future Work

We have presented algorithms and an implementation of an automated reasoning system for default logic on the basis of unrestricted first order logic and a resolution method. While there are several systems able to solve default logic or similar nonmonotonic logic problems, these are built on the basis of answer set programming and are normally based on grounding. We are not aware of other full first order logic reasoning systems for default logic, and neither of systems integrating confidences and inconsistency-handling with rules with exceptions.

Future work is planned on three directions: adding features to the solver, proving several useful properties of the algorithms and incorporating the solver into a commonsense reasoning system able to handle nontrivial tasks posed in natural language. The work on incorporating similarity-based reasoning into GK and building a suitable semantic parser for natural language is currently ongoing. We are particularly interested in exploring practical ways to integrate GK with the machine learning techniques for natural language.

References

1. Alviano, M., et al.: The ASP system DLV2. In: Balduccini, M., Janhunen, T. (eds.) LPNMR 2017. LNCS (LNAI), vol. 10377, pp. 215–221. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61660-5_19

2. Arias, J., Carro, M., Salazar, E., Marple, K., Gupta, G.: Constraint answer set programming without grounding. Theor. Pract. Logic Program. **18**(3–4), 337–354 (2018)
3. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, ch. 2, pp. 19–99. Elsevier, Amsterdam (2001)
4. Brewka, G.: Adding priorities and specificity to default logic. In: MacNish, C., Pearce, D., Pereira, L.M. (eds.) JELIA 1994. LNCS, vol. 838, pp. 247–260. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0021977
5. Cholewinski, P., Marek, V.W., Truszczynski, M.: Default reasoning system deres. KR **96**, 518–528 (1996)
6. Green, C.: Theorem proving as a basis for question-answering systems. Mach. Intell. **4**, 183–205 (1969)
7. Kaminski, M.: A comparative study of open default theories. Artif. Intell. **77**(2), 285–319 (1995)
8. Kaminski, R., Schaub, T., Wanko, P.: A tutorial on hybrid answer set solving with *clingo*. In: Ianni, G. (ed.) Reasoning Web 2017. LNCS, vol. 10370, pp. 167–203. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61033-7_6
9. Kothawade, S., Khandelwal, V., Basu, K., Wang, H., Gupta, G.: Auto-discern: Autonomous driving using common sense reasoning (2021). arXiv preprint. arXiv:2110.13606
10. Lifschitz, V.: Answer Set Programming. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-24658-7
11. Nute, D.: Defeasible Logic, vol. 3. Oxford University Press, Oxford (1994)
12. Ramachandran, D., Reagan, P., Goolsbey, K.: First-orderized researchcyc: expressivity and efficiency in a common-sense ontology. In: AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications, pp. 33–40 (2005)
13. Reiter, R.: A logic for default reasoning. Artif. Intell. **13**(1–2), 81–132 (1980)
14. Romero, J., Razniewski, S., Pal, K., Pan, J.Z., Sakhadeo, A., Weikum, G.: Commonsense properties from query logs and question answering forums. In: Zhu, W. (eds.) Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM'19, pp. 1411–1420. ACM (2019)
15. Speer, R., Chin, J., Havasi, C.: ConceptNet 5.5: An open multilingual graph of general knowledge. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of the 31st AAAI Conference on Artificial Intelligence, pp. 4444–4451. AAAI (2017)
16. Sutcliffe, G., Yerikalapudi, A., Trac, S.: Multiple answer extraction for question answering with automated theorem proving systems. In: Lane, H.C., Guesgen, H.W. (eds.) Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference, FLAIRS'22. AAAI (2009)
17. Tammet, T.: GKC: a reasoning system for large knowledge bases. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 538–549. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_32
18. Tammet, T., Draheim, D., Järv, P.: Confidences for Commonsense Reasoning. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 507–524. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_29