Proof Checker

Nelly Vouzoukidou, Yannis Theoharis, Yannis Makrydakis, Antons Krithinakis, Nikos Mouchtaris, Giorgos Hatzivassilis, Vasilis Efthymiou

Abstract

The goal of a proof checker is to decide whether the provided proof, given a theory, is valid or not. In the case the proof is not valid, an appropriate error message is returned, depending on the nature of the problem. Our approach was to modify the prolog code of the *metaprogam* that is used to turn simple Prolog into DR-Prolog. Every decision is made in O(1) time, since recursion is avoided, as described below. Finally, a set of 11 examples were created to exhibit as many error cases/messages as possible, as well as correct proofs that return no error messages.

Assumptions

1. We assume that the theory is the same as the one given to the proof generator. Moreover, any given theory is accepted as valid without any checks.
2. No checks are performed recursively. Thus, we require any information in depth more than one a priori.
3. Any knowledge given in the theory is considered to be definitely provable. For instance all facts are added in definite knowledge regardless of the presence of a statement in the proof supporting it.
4. We require the minimal information that will contribute to the proof checking process.

Structures

We distinguish between two groups of collections.

Theory structures: We have four knowledge bases holding the strict rules, defeasible rules, facts and rules hierarchy. These help as retrieve any information used by the proof.

```
%Theory structures
:- dynamic strictkb/1.
:- dynamic defeasiblekb/1.
:- dynamic factkb/1.
:- dynamic supkb/1.
```

Proof deduction structures: We have four knowledge bases holding any deducted information already stated by the proof and confirmed by the checker.

```
%Proof deduction structures
:- dynamic definitelykb/1.
```

:- dynamic defeasiblykb/1.
:- dynamic minusdefinitelykb/1.
:- dynamic minusdefeasiblykb/1.

For example a rule that adds knowledge to the defintlykb is the following:
definitelyCheck(X, Print):- Print=printOn, factkb(F), memberchk(X, F), addDefinitely(X).

Facts

All facts given in the theory, are also added in the definite knowledge base. This means that any stated fact is by default considered by the proof checker, e.g. we accept all the following three statements, provided that a is a fact in the theory:

1. fact(a).
   definitely(a).

2. definitely(a).

3. fact(a).

Rules

Since theory is by default loaded, the proof does not have to state rules explicitly, although it is allowed to do so. Explicit statements are ignored.

Deductions

Two separate predicates are used for strict deductions, namely +Δ (definitely) and -Δ (not_definitely). Similarly, for defeasible deductions we define predicates defeasibly for +θ and not_defeasibly for -θ.

In general for positive deductions (+Δ and +θ), when the conclusion is proven by a rule, the name of the rule is required by the proof checker. Elsewise, i.e. when the conclusion is a fact or it is already given or deducted at a previous step, it is not required. This approach is followed for the sake of efficiency.

On the other hand, for negative deductions, giving the name of the rule would be redundant. That is because, even if a negative result is concluded by one rule, the checker still has to retrieve all existing relevant rules regardless of whether the proof states them or not.

For example the rules that check if a stated literal, claimed to be a fact or already proven, is definetely provable or not, are the following:

definitelyCheck(X, Print):- Print=printOn, factkb(F), memberchk(X, F), addDefinitely(X).
definitelyCheck(X, Print):- Print=printOff, factkb(F), memberchk(X, F).
definitelyCheck(X, _):- definitelykb(K), memberchk(X, K).
definitelyCheck(X, Print):- logError(Print, [X, ' is neither a fact nor has yet been proven.']).

Print is used to state if errors are to be printed or not and the values it takes are *printOn* and *printOff*. The first two rules check if X is really a fact (it is a member of the fact knowledge base). The third rule checks if X is already a member of the definitely knowledge base (it has already been proven that X is definitely provalbe). If these three rules fail, it means that X is not really a fact or a literal that has been proven definitely, so an error message is printed, following the fourth rule, stating that *"X is neither a fact nor has yet been proven."*

Examples

Below we explain the evaluation steps for a complex example where team defeat occurs.
r1: a => e
r2: b => e
r3: c => ~e
r4: d => ~e
r1 > r3
r2 > r4
a. b. c. d.

A valid and correct proof is the following:
defeasibly(a), defeasibly(b), defeasibly(c), defeasibly(d), defeasibly(e, r2).

The first four statements are obviously deducted, since a, b, c, d are facts. The last, i.e. defeasibly(e, r2) runs the following checks:
1. Is there any rule "r2" in the theory, with head equal to e?
    Yes.
2. Is there any attacking rule?
    Yes, r3, r4
    2.1 Is r2 of higher priority than r3?
        No.
        2.1.1 Are the conditions of r3 (i.e. c) defeasibly provable?
            Yes.
        2.1.2 Is there any attacking rule of r3 (different from r2), which defeats r3?
            Yes, r1, because its conditions are met and r1 > r3.
    2.2 Is r2 of higher priority than r4?
        Yes.

Similarly, defeasibly(e, r1) would also be correct.

Parsing the proof

The proof checker after receiving the validation request parses the received proof and constructs a prolog query based on the claimed proof. The received theory is also constructed and loaded on the prolog engine in support to the proof checker. The xml formatted proof, contains a lot of redundant or unnecessary information that is omitted from the validation query. Elements declaring provability of a predicate along with the predicate and often the rule name

are of major concern, while elements describing the body of a rule or superiority claims are ignored as mentioned in the above deductions section. Due to the lack of an XML shema or a DTD describing the proof construct the parser does not yet provide any xml validation.