Künstliche Intelligenz

Hausaufgabe 2

N. Lehmann, A. Zubarev

05.05.2015

1 TPTP

1.1 Teilaufgabe a)

```
% If you work hard, then you get lucky.
fof(1, axiom, ![X]: wh(X) => l(X)).
% Either you get lucky or you work hard, or both.
fof(2, axiom, ![X]: wh(X) | l(X)).
% If you get lucky then, either you are not a rogue or you work hard (but not both).
fof(3, axiom, ![X]: l(X) => (~r(X) | wh(X)) & (~(~r(X) & wh(X)))).
% You are a rogue.
fof(4, axiom, r(u)).
% You work hard.
fof(5, conjuncture, wh(u)).
```

1.2 Teilaufgabe b)

```
% Garfield is a cat. Odie is a dog.
fof(6, axiom, isCat(garfield)).
fof(7, axiom, isDog(odie)).
% Cats and dogs are animals.
fof(8, axiom, ![X]: isCat(X) | isDog(X) => isAnimal(X)).
% Jon is a human.
fof(9, axiom, isHuman(jon)).
% Every animal has a human owner.
fof(10, axiom, ![A]?[E]: isOwner(A,O) & isAnimal(A) & isHuman(E)).
% Jon is the owner of Garfield and Odie.
fof(11, axiom, isOwner(garfield,jon)).
fof(12, axiom, isOwner(odie, jon)).
% Garfield and Odie are the only animals that Jon owns.
```

```
fof(13, axiom, ![A]: (isCat(A) & ~(A=garfield)) | (isDog(A) & ~(A=odie)) =>
                      ~(isOwner(A,jon))).
% If a cat is chased by a dog, then the owner of the cat hates the owner of the dog.
fof(14, axiom, ![C,D,O,U]: chases(D,C) => hates(O,U) & isOwner(C,O),isOwner(D,U))).
% Odie has chased Garfield.
fof(15, axiom, chases(odie,garfield)).
% Jon hates himself.
fof(16, conjuncture, hates(jon, jon)).
```

1.3 Teilaufgabe c)

```
% Wolves, foxes, birds, caterpillars, and snails are animals, and there are some
% of each of them. Also there are some grains, and grains are plants.
fof(17, axiom, ![X]: isW(X) => isA(X)).
fof(17, axiom, ![X]: isF(X) => isA(X)).
fof(17, axiom, ![X]: isB(X) => isA(X)).
fof(17, axiom, ![X]: isC(X) => isA(X)).
fof(17, axiom, ![X]: isS(X) => isA(X)).
fof(18, axiom, ?[X]: isG(X) => isP(X)).

% Every animal either likes to eat all plants or all animals much smaller than itself
% that like to eat some plants.
fof(19, axiom, ![A,B]?[C]: ((eats(A,B) & isAnimal(A) & isPlant(B)) |
                            (eats(A,B) & isAnimal(A) & isAnimal(B) & isSmaller(B,A)
                            & eats(B,C) & isPlant(C))) & ~((eats(A,B) & isAnimal(A)
                            & isPlant(B)) & eats(A,B) & isAnimal(A) & isAnimal(B)
                            & isSmaller(B,A) & eats(B,C) & isPlant(C))).

% Caterpillars and snails are much smaller than birds, which are much smaller than
% foxes, which in turn are much smaller than wolves.
fof(20, axiom, ![A,B]: isSmaller(A,B) & isC(A) & isB(B)).
fof(21, axiom, ![A,B]: isSmaller(A,B) & isS(A) & isB(B)).
fof(22, axiom, ![A,B]: isSmaller(A,B) & isB(A) & isF(B)).
fof(23, axiom, ![A,B]: isSmaller(A,B) & isF(A) & isW(B)).

% Wolves do not like to eat foxes or grains, while birds like to eat caterpillars
% but not snails.
fof(24, axiom, ![A,B]: isW(A) & isF(B) & ~(eats(A,B))).
fof(25, axiom, ![A,B]: isW(A) & isP(B) & ~(eats(A,B))).
fof(25, axiom, ![A,B]: isB(A) & isC(B) & eats(A,B)).
fof(25, axiom, ![A,B]: isB(A) & isS(B) & ~(eats(A,B))).

% Caterpillars and snails like to eat some plants.
fof(26, axiom, ![A]?[B]: isC(A) & isPlant(B) & eats(A,B).
fof(27, axiom, ![A]?[B]: isS(A) & isPlant(B) & eats(A,B).
```

```
% There is an animal that likes to eat a grain eating animal.
fof(28, conjuncture, ?[A,B,C]: isAnimal(A) & isAnimal(B) & eats(A,B)
                              & eats(B,C) & isGrain(C)).
```

2 Listen in Prolog

2.1 Teilaufgabe a)

```
myLast([Element], Element).
myLast([_|Restliste], Element) :- myLast(Restliste, Element).
```

2.2 Teilaufgabe b)

```
/*
   select(Elem, Liste1, Liste2) produziert in 'Liste2' immer eine neue Liste
   aus 'Liste1' nur ohne 'Elem' (alle Permutationen wo jeweils einmal 'Elem'
   in 'Liste1' geloescht wird)
   member(Elem, Liste) ueberprueft ob das 'Elem' in der 'Liste' vorkommt oder
   nicht
   \+( ) wird wahr wenn die ineere Definition sich als falsch erweist
   Das heißt wir produziern alle Liste ohne das maximale Element und ueberpruefen
   ob darin kein groesseres Element vorkommt */
   Das Ermitteln des groessten Elements ist einfach, denn es werden einfach alle
   Kombinationen durchgelaufen bis die Definition wahr wird
 */
myMax(Liste, MaxElement) :- select(MaxElement, Liste, ListeOhneMaxElement),
                            \+( (member(AnderesElement, ListeOhneMaxElement),
                            AnderesElement > MaxElement) ).
```

2.3 Teilaufgabe c)

```
/*
   Wenn die Liste leer ist, ist die Summe gleich 0
   Wenn die Liste aus einem Element besteht, dann ist es die Summe
   Wenn die Liste mehrere Elemente hat dann addiere die ersten zwei Elemente,
   baue eine neue Liste aus der Summe der beiden Elemente und der Resliste und
   pruefe rekursiv
 */
mySum([], 0).
mySum([Element], Summe) :- Summe is Element.
mySum([Element1|[Element2|Restliste]], Summe) :- mySum([Element1 + Element2|Restliste],
                                                 Summe).
```

2.4 Teilaufgabe d)

```
/*
   Eine leere Liste ist als sortiert anzunehmen.
   Eine Liste aus einem Element ist als sortiert anzunehmen. Zugleich die
   Abbruchbedingung fuer eine rekursive Ueberpruefung ob eine Liste aufsteigend
   sortiert ist oder nicht.
   Wir nehmen an wir koennen nur pruefen ob die Liste aufsteigend sortiert ist,
   dann muss jedes Element <= dem Nachfolger in der Liste sein (Loesung rekursiv)
 */
myOrdered([]) :- true.
myOrdered([_]) :- true.
myOrdered([Element1|[Element2|Restliste]]) :- Element1 =< Element2 ->
                                              myOrdered([Element2|Restliste]).
```