Beviskontroll med Prolog

;

Grupp Lumon

Ludwig Berglind, Simon Severinsson

Beskrivning av algoritmen

Programmet börjar med att köra verify\1 som öppnar filen angiven i argumentet och läser sedan in bevisets premisser i variabeln Prems, målet i variabeln Goal och själva beviset i sig i variabeln Proof, som blir en lista av listor vars element är raderna i beviset. Dessa variabler tas sedan in som argument i validProof\3. Där kollas först att den sista raden i beviset är slutsatsen och att sista raden inte är ett antagande vilket görs genom att rekursivt traversera genom hela beviset. Sedan kallas predikatet checkSteps\4 där beviset faktiskt valideras. checkSteps\4 tar in premisserna, slutsatsen, hela beviset, samt en lista där nuvarande raden är huvudet. Här valideras varje rad i beviset rekursivt genom att kolla om den är en premiss, en ny låda, eller om den stämmer in på någon av de andra reglerna. Svansen läggs sedan in i checkSteps\4 och basfallet nås när svansen är tom, dvs. när hela beviset har kollats igenom.

För att kolla om en rad är en premiss används predikatet checkPremise\2 som tar in en rad från beviset och Prems, som ju innehåller premisserna. Sedan används det inbyggda member\2-predikatet för att se till att formeln i raden faktiskt är del av Prems.

För alla andra regler används predikatet checkRule\2 som tar in en rad från beviset och en lista över alla rader i beviset. I samtliga checkRule-klausuler kontrolleras det att ingen regel använder sig av senare rader genom checkLines-predikaten. Sedan kollas det om raderna som hänvisas till i reglerna på varje rad finns med i beviset genom member\2-predikatet. Vissa regler använder sig av boxar. Genom att kolla om den specifika raden i en lista med wildcard- svans finns i beviset kan vi konstatera att den är början av en box. Samma lista med svans kan sedan också läggas in i boxLast\3-predikatet för att kontrollera att även den sista raden i boxen existerar. boxLast\3 traverserar rekursivt till sista elementet/raden i listan och kollar att radnummer och formel stämmer överens med regeln. LEM-regeln använder inte någon member-kontroll utan kollar endast att innehållet på raden är en eller-relation mellan någon variabel och dess negation samt att lem-regeln används.

När hela beviset rekursivt har traverserats uppifrån och ner nås basfallet och programmet backtrackar hela vägen till validProof\3.

Boxhanteringen:

Boxar hanteras genom predikatet newBox\3 som kollar att första raden i boxen är ett antagande och sedan kallas predikatet checkSteps med boxens svans ([_|T]) som listan med rader som ska kontrolleras samt boxens rader appendad till Proof som Proof, så att boxens innehåll kan åtkommas tills boxen stängs. Inuti boxen kör programmet alltså på som vanligt och verifierar rader fram tills listan med boxens innehåll är slut och checkSteps når basfallet. Då backtrackar programmet till den punkt där newBox kallades från i checkSteps, och kallar som vanligt checkSteps igen på nästa rad, som är den direkt efter boxen, och med vanliga Proof så som den var innan newBox.

Predikat	Sant	Falskt
assertTransitions\1 When all transitions have been asserted as facts If the formatting of the transitions is incorrect		
assertTransitionsHelper\1	When the transitions from one node have been asserted	If the formatting of the list is incorrect
assertTransition\2	When all transitions from a node have been asserted	Då beviset inte stämmer (validProof är false), eller inläsning av misslyckas
assertStates\1	When all states have been asserted	If the formatting of the states is incorrect
assertState\1	When a state has been asserted	If the formatting of a state is incorrect
checkAllPaths\3	When all paths from the current state satisfies the formula	When one or more paths from the current state dont satsify
checkAllPathsH\3	When the current path being checked satisfies the formula	When the current path being checked does not satisfy the formula
checkExistsPath\3	When one of the paths from the current state satsify the formula	When none of the paths from the current state satisfy the formula
checkExistsPathH\3	When the current path being checked satisfies the formula	When the current path being checked does not satisfy the formula
check\3	When the current state satisfies the formula	When the current state does not satisfy the formula

Predikat	Sant	Falskt
stateContains\2	When the current state satisfies the formula (formula will always be an atom)	When the current state does not satisfy the formula
abolishAll\0	Always	Never

Appendix A - Kod

```
verify(InputFileName) :-
    abolishAll(),
    see(InputFileName),
    read(Adjacencies), read(States), read(State), read(Formula),
    seen,
    assertTransitions(Adjacencies),
    assertStates(States),
    check(State, [], Formula), !,
    abolishAll().

assertTransitions([]).

assertTransitions([Head | Tail]) :-
    assertTransitionsHelper(Head),
    assertTransitions(Tail).

assertTransitionsHelper([Node | [To | _]]) :-
    assertTransition(Node, To).

assertTransition(_, []).
assertTransition(Node, [Head | Tail]) :-
    assertz(transition(Node, Head)),
    assertTransition(Node, Tail).

assertStates([]).
assertStates([Head | Tail]) :-
    assertState(Head),
    assertStates(Tail).

assertState([Node | [State | _]]) :-
    assertz(state(Node, State)).

checkAllPathsH([], _, _).

checkAllPaths(State, U, Formula) :-
    findall(Next, transition(State, Next), L),
    checkAllPathsH(L, U, Formula).

checkAllPathsH([Head | Tail], [], Formula) :-
    check(Head, [], Formula),
    checkAllPathsH(Tail, [], Formula).

checkAllPathsH([Head | Tail], U, Formula) :-
    check(Head, U, Formula),
```

```
    checkAllPathsH(Tail, [Head| U], Formula).

checkExistsPath(State, [], Formula) :-
    findall(Next, transition(State, Next), L),
    checkExistsPathH(L, [], Formula).

checkExistsPath(State, U, Formula) :-
    findall(Next, transition(State, Next), L),
    checkExistsPathH(L, U, Formula).


checkExistsPathH([], _, _) :- fail.

checkExistsPathH([Head | Tail], U, Formula) :-
    check(Head, U, Formula) ;
    checkExistsPathH(Tail, [Head | U], Formula).

% Atom
check(State, _, Formula) :-
    stateContains(State, Formula).

% Neg
check(State, _, neg(Formula)) :-
    \+ stateContains(State, Formula).

% And
check(State, _, and(F, G)) :-
    check(State, [], F), check(State, [], G).

% Or
check(State, _, or(F, G)) :-
    check(State, [], F) ; check(State, [], G).

% AX
check(State, [], ax(Formula)) :-
    checkAllPaths(State, [], Formula).

% EX
check(State, _, ex(Formula)) :-
    checkExistsPath(State, [], Formula).

% AG
check(State, U, ag(_)) :-
    member(State, U).

check(State, U, ag(Formula)) :-
```

```
    \+ member(State, U),
    check(State, [], Formula),
    checkAllPaths(State, [State | U], ag(Formula)).

% EG
check(State, U, eg(_)) :-
    member(State, U).
check(State, U, eg(Formula)) :-
    \+ member(State, U),
    check(State, [], Formula),
    checkExistsPath(State, [State | U], eg(Formula)).

% EF
check(State, U, ef(Formula)) :-
    \+ member(State, U),
    check(State, [], Formula).

check(State, U, ef(Formula)) :-
    \+ member(State, U),
    checkExistsPath(State, [State | U], ef(Formula)).

% AF
check(State, U, af(Formula)) :-
    \+ member(State, U),
    check(State, [], Formula).
check(State, U, af(Formula)) :-
    \+ member(State, U),
    checkAllPaths(State, [State | U], af(Formula)).

stateContains(Node, Formula) :-
    state(Node, L),
    member(Formula, L).

abolishAll() :- abolish(transition/2), abolish(state/2).
```