Laboration 2: Beviskontroll

Elvira Häggström
elvhag@kth.se

Tom Axberg
taxberg@kth.se

1. Problembeskrivning

Denna laboration går ut på att ett prologprogram ska skrivas. Det ska kontrollera ifall ett bevis skrivet med naturlig deduktion, enligt det sätt som beskrivs i vår kurslitteratur[1], är korrekt eller inte. Programmet ska svara med “yes” eller “true” om det givna beviset är korrekt, och annars med “no” eller “false”.

Ett bevis är korrekt om alla regler som används är applicerade på ett tillfredsställande sätt. En lista av de regler som tillåts i programmet finns i appendix.

Problemet anses löst när det skrivna programmet ges som input till det givna programmet ‘run_all_tests.pl’ och detta program då ger oss en utskrift i stdout där alla givna textfiler, med bevis i naturlig deduktion, skrivs ut med strängen “passed” efteråt. “passed” indikerar att det skrivna programmet hanterar beviset på ett korrekt sätt.

2. Angreppssätt

För att konstruera en lösning bestämdes det att beviset skulle läsas uppifrån och ned, rad för rad. För varje ny rad valideras regeln som används.

I de fall då boxar förekommer används det faktum att en box ska fungera precis likadant som beviset i stort. Om itereringen genom boxen sker som om det vore ett eget bevis är den korrekt utformad. Detta ska även fungera för de rekursiva fallen då det finns boxar i boxar.

Det behöver implementeras ett sätt att se tidigare i beviset, då majoriteten av reglerna kräver en eller flera referenser till rader i formen av radnummer. Detta görs genom att skicka med två listor som indata i iteratorpredikatet - en med alla rader som ännu inte har validerats och en med alla de rader som redan har validerats. För varje ny rad, eller box, som valideras flyttas den över från den första till den andra listan. På detta sätt finns det alltid möjlighet att titta tillbaka tidigare i beviset på de rader, eller boxar, som blivit validerade.

Boxhantering är det mest komplexa problemet att lösa för den här uppgiften.
Den planerade strukturen grundar sig i att se till att hantera boxar precis som beviset i stort - men vilka rader som ska synas för vilka kommer behöva specificeras.

[1] Michael Huth and Mark Ryan, *Logic in Computer Science*: *Modeling and reasoning about systems*, Second Edition., Cambridge Yale University Press, 2004, e-bok, kapitel 1.2.

2.1 Beviskontroll

Beviskontrollen, alltså det prologrogram denna rapport omfattar (se appendix B), är till sin struktur uppdelad i fyra delar. Predikatet check_proof/3 tillsammans med de första raderna i programmet, vilka tar in data från textfiler (se exempel i appendix C) samt kollar målet, som utgör grunden i beviskontrollen. En uppsättning predikat som alla implementerar de regler som finns med i Appendix A. Därefter predikatet check_box/3 som hanterar boxar. Mer utförlig förklaring av check_box/3, se 2.2 Boxhantering. Till sist, gruppen av hjälppredikat som hjälper de ovan nämnda delarna att fungera korrekt. Ingående förklaring av hjälppredikaten hittas i 2.3 Hjälpfunktioner.

Grundstrukturen i beviskontrollen ligger runt predikatet check_proof/3 som tar in tre argument. Det första är premisserna, det andra är beviset som en lista, givet av indata och det tredje som är en lista av rader och lådor som genomgått validering. Innan exekvering av check_proof/3 ska tilläggas, med hjälp av check_goal/2 (se 2.3 Hjälpfunktioner), att slutsatsen jämförs med sista raden i beviset. Skulle dess inte stämma överens är inte beviset korrekt skrivet. Vid exekvering av check_proof/3 skickas som argument premisserna i en lista, beviset som en lista och en tom lista. Uppifrån och ned kommer nu algoritmen ta ett element i taget i bevislistan, validera detta och sedan lägga till det i den lista över validerade listor. Check_proof/3 exekveras rekursivt med listan över premisser, de resterande elementen i beviset som behöver valideras och listan över de validerade elementen. När den lista över rader att validera är tom kommer algoritmen vara klar. Basfallet check_proof/3 avslutar rekursionen genom att ta in argumenten “(_, [ ], _)”.

Beviskontrollen är relativt enkel då den endast innehåller ett antal steg. En sammanfattad lista över de steg som redogjorts följer nedan.

Start:

1. Kolla att slutsatsen är den samma som sista raden i beviset.

Rekursion check_proof/3:

1. Validera första elementet i listan. (Se 2.3 Hjälpfunktioner)
2. Lägg till elementet i listan över hanterade element (Append/3)
3. Kör rekursionen med premisser, de resterande elementen att validera och listan över hanterade element. (check_proof/3) tills listan över element att validera är en tom lista.

För mer ingående förklaring om hur element (rader och lådor) valideras, se avsnitten 2.2 Boxhantering samt 2.3 Hjälpfunktioner.

2.2 Boxhantering

Lösningen hanterar boxar med hjälp av predikaten validate_line/3, check_proof/3, och check_box/3.

validate_line/3 fångar upp de fall då nästkommande “rad” i själva verket är en box, och kallar då på check_proof/3. På så sätt hanteras alltså boxar som om de vore egna bevis. check_proof/3 går igenom boxen rad för rad, precis som med beviset i stort.

check_box/3 används i alla de fall av validate_line/3 som implementerar de regler som baseras på att en specifik box existerar. check_box/3 undersöker om det förekommer en box tidigare i beviset med den givna första raden (variabeln FirstLine), andra raden (variabeln LastLine), samt de redan validerade raderna (variabeln Processed).

Ett specialfall av check_box/3 är när en box endast är en rad lång. Då fungerar inte det inbyggda predikatet last/2 som originalversionen av check_box/3 bygger på, vilket gör att en specialversion behövde implementeras. Denna fungerar på liknande sätt som originalversionen, förutom det faktum att den första inputen (FirstLine) och den andra inputen (LastLine) båda kan unifiera till samma variabel - i vårt fall till variabeln Line.

2.3 Hjälpfunktioner

check_proof/3 fungerar som en iterator och går igenom beviset. Som inputs tar den listan med premisser (Prems), listan med rader att gå igenom ([ToProcess|ToBeProcessed]), och listan med rader som redan har gåtts igenom (Processed). För varje rad valideras den nuvarande raden med hjälp av validate_line/3. Därefter flyttas den nuvarande raden till listan av föregående rader, och sist körs check_proof/3 igen med de uppdaterade variablerna. Detta fortsätter tills den når basfallet: listan av rader att gå igenom är tom.

check_processed/2 ska undersöka om det första argumentet (en rad i beviset) finns i listan av validerade rader (Processed). Ett problem som stöttes på här var i vilka fall en rad ska vara synlig för en annan. Exempelvis ska en rad inte kunna vara synlig för en annan om den första finns i en box som redan har stängts. Här utnyttjas det faktum att för varje nyöppnad box så hanteras den som om den vore ett eget bevis. Allt som ska vara synligt hamnar då temporärt på samma nivå i Processed, och man kommer åt rätt rader med en enkel användning av predikatet member/2.

validate_line/3 är det predikat som används för att implementera alla regler. Lösningen använder en version av validate_line/3 för varje regel som ska tillåtas, och använder i sin tur bland andra check_box/3 och check_processed/2 för att ta reda på om vardera regel har använts på korrekt sätt. En specialversion av validate_line/3 är den som hanterar nyöppnade boxar. Istället för att hantera inputen som en rad tolkar programmet att det i själva verket är en box, och kallar då på predikatet check_proof/3 för att hantera boxen som om det vore ett eget bevis.

check_goal/2 används för att tidigt i programmets körning se till att det mål som beviset har i själva verket är den sista raden. Om så inte är fallet, vet vi redan då att beviset är felaktigt.

check_box/3 fungerar som tidigare nämnt i 2.2 Boxhantering genom att den undersöker ifall det finns en stängd box i listan av redan validerade rader (Processed) som startar med den första raden (FirstLine) och slutar med den andra raden (LastLine). Det finns även en specialvariant som hanterar de fall då boxen endast består av en enda rad.

3. Delmål / Plan

Planen var att tidigt implementera en grundstruktur i programmet som läste av det givna beviset, uppifrån och ned, och som avslutade när sista raden hade validerats. I samband med implementationen skulle grundläggande delar också implementeras. Dessa var en verifikation av att slutsatsen var den sista raden i beviset samt en lista över validerade rader som skulle följa med genom hela bevisavläsningen. När detta implementerats kunde sedan de enskilda fallen valideras. Detta gjordes genom att skriva ett predikat, per regel, som identifierade vilken regel som var den aktuella. Genom att inte specificera implikationen till det enskilda predikatet kunde det lätt testas, för några enklare bevis, om grundstrukturen fungerade. När testet av mål, grundstruktur och listan över validerade rader var fungerade kunde sedan predikaten för respektive regel implementeras. I detta skede skulle predikat som premisser och enklare regler implementeras och testas för att sedan, om implementeringen är korrekt, implementera alla regler som ska ingå i programmet. När detta var klart kom fallen med lådor. Implementeringen av validering av lådor beskrivs i 2.1. Slutligen skulle alla textfiler med bevis testas för att identifiera eventuella problem. Förväntningen var här att finjustera de enskilda predikaten och att grundstrukturen skulle hålla, vilket också var fallet.

4. Reflektion

Genom att göra detta projekt har vi fått en djupare förståelse för Prolog som programmeringsspråk. Det var svårt att vänja sig vid ett logikprogrammeringsspråk efter att man blivit van vid imperativa språk som C och Java, men efter detta projekt känner vi oss båda mer bekväma med användningen av språket.

Detta projekt var också en bra start för att få in en rutin för hur man programmerar i grupp. Exempelvis användes Github flitigt, vilket var ovant för båda parter. På grund av rådande omständigheter har vi dessutom inte kunnat ses och arbeta tillsammans i verkligheten, utan all kommunikation har skett via Discord, genom att dela skärm och diskutera lösningar. Detta har fungerat mycket bra och båda parter är nöjda med resultatet och samarbetet.

Under projektets gång har labbrapporten i sig också varit en tillgång, för att hålla koll på delmål, planering och vilka tester som passerar. Under tiden som projektet pågått så har en struktur upprätthållits för att ha en överblick över arbetets uppdelning och status. Detta har gjort att arbetet lätt har kunnat återupptas utan förvirring när var och en haft tid.

Vi är mycket nöjda med resultatet!

Appendix

A. Regler

I följande lista återfinns de regler och koncept som tillåts som indata i programmet:

- premise — Hanterar premisser
- assumption — Hanterar antaganden (och öppnar därmed också boxar)
- copy(x) — Hanterar copy
- andint(x, y) — Hanterar and-introduction
- andel1(x) — Hanterar and-elimination 1
- andel2(x) — Hanterar and-elimination 2
- orint1(x) — Hanterar or-introduction 1
- orint2(x) — Hanterar or-introduction 2
- orel(x, y, u, v, w) — Hanterar or-elimination
- impint(x, y) — Hanterar implication-introduction
- impel(x, y) — Hanterar implication-elimination
- negint(x, y) — Hanterar negation-introduction
- negel(x, y) — Hanterar negation-elimination
- contel(x) — Hanterar contradiction-elimination
- negnegint(x) — Hanterar negation-negation-introduction
- negnegel(x) — Hanterar negation-negation-elimination
- mt(x, y) — Hanterar modus tollens
- pbc(x, y) — Hanterar proof by contradiction
- lem — Hanterar law of excluded middle

B. Kod

```
verify(InputFileName) :- see(InputFileName),
read(Prems), read(Goal), read(Proof),
seen,
valid_proof(Prems, Goal, Proof).

% Check if last element in Proof is Goal and iterates through the proof and checks for validity.
valid_proof(Prems, Goal, Proof) :-
check_goal(Proof, Goal),
check_proof(Prems, Proof, []),
!.

% If the list of rows to validate is the empty list, we are done.
check_proof(_, [], _).

% Check if the given proof is valid
check_proof(Prems, [ToProcess|ToBeProcessed], Processed) :-
validate_line(Prems, ToProcess, Processed),
append(Processed, [ToProcess], Concat),
check_proof(Prems, ToBeProcessed, Concat).

% Chech if row is in Processed.
check_processed(Row, Processed) :-
member(Row, Processed), !.

% Checks box
check_box(FirstLine, LastLine, Processed) :-
member([FirstLine|T], Processed),
last(T, LastLine).

check_box(Line, Line, Processed) :-
member([Line], Processed).
```

```
% Handles premise
validate_line(Prems, [_, Sats, premise], _) :-
member(Sats, Prems).

% Handles assumption. Case when assumption gives box.
validate_line(Prems, [[_, _, assumption]|BoxTail], Processed) :-
append(Processed, [[_, _, assumption]], Concat),
check_proof(Prems, BoxTail, Concat).
```

```
% Handles copy.
validate_line(_, [_, A, copy(X)], Processed) :-
check_processed([X, A, _], Processed).

% Handles andint.
validate_line(_, [_, and(A, B), andint(X,Y)], Processed) :-
check_processed([X, A, _], Processed),
check_processed([Y, B, _], Processed).

% Handles andel1(X).
validate_line(_, [_, A, andel1(X)], Processed) :-
        check_processed([X, and(A, _), _], Processed).

% Handles andel2(X).
validate_line(_, [_, B, andel2(X)], Processed) :-
        check_processed([X, and(_, B), _], Processed).

% Handles orint1(X).
validate_line(_, [_, or(A, _), orint1(X)], Processed) :-
        check_processed([X, A, _], Processed).

% Handles orint2(X).
validate_line(_, [_, or(_, B), orint2(X)], Processed) :-
        check_processed([X, B, _], Processed).

% Handles orel(X, Y, U, V, W).
validate_line(_, [_, A, orel(X, Y, U, V, W)], Processed) :-
        check_processed([X, or(B, D), _], Processed),
        check_box([Y, B, assumption], [U, A, _], Processed),
        check_box([V, D, assumption], [W, A, _], Processed).

% Handles impel(X, Y).
validate_line(_, [_, B, impel(X, Y)], Processed) :-
        check_processed([X, A, _], Processed),
        check_processed([Y, imp(A, B), _], Processed).

% Handles negint(X, Y).
validate_line(_, [_, neg(Z), negint(X, Y)], Processed) :-
        check_box([X, Z, assumption], [Y, cont, _], Processed).
```

```
% Handles negel(X, Y).
validate_line(_, [_, cont, negel(X, Y)], Processed) :-
        check_processed([X, A, _], Processed),
        check_processed([Y, neg(A), _], Processed).

% Handles contel(X).
validate_line(_, [_, _, contel(X)], Processed) :-
        check_processed([X, cont, _], Processed).

% Handles negnegint(X).
validate_line(_, [_, neg(neg(Sats)), negnegint(X)], Processed) :-
        check_processed([X, Sats, _], Processed).

% Handles negnegel(X).
validate_line(_, [_, Sats, negnegel(X)], Processed) :-
        check_processed([X, neg(neg(Sats)), _], Processed).

% Handles mt
validate_line(_, [_, _, mt(X, Y)], Processed) :-
        check_processed([X, imp(_, B), _], Processed),
        check_processed([Y, neg(B), _], Processed).

% Handles pbc(X, Y).
validate_line(_, [_, A, pbc(X, Y)], Processed) :-
        check_box([X, neg(A), assumption], [Y, cont, _], Processed).

% Handles lem.
validate_line(_, [_, or(A, neg(A)), lem], _).
```

```
% Handles impint.
validate_line(_, [_, imp(R,Q), impint(X,Y)], Processed) :-
        check_box([X, R, assumption], [Y, Q, _], Processed).

% Exploits check_goal such that the last element cannot be an assumption
check_goal(Proof, Goal) :-
        last(Proof, [_, Goal, assumption]), !, fail.

% Checks if goal is the last line.
check_goal(Proof, Goal) :-
        last(Proof, [_, Goal, _]).
```

C. Exempelbevis

Inkorrekt bevis.

```
[imp(p, imp(q, r))].

imp(and(q, neg(r)), neg(p)).

[
   [1, imp(p, imp(q, r)), premise],
   [
      [2, and(q, neg(r)), assumption],
      [3,     q         , andel1(2) ],
      [
         [4, p,          assumption],
         [5, imp(q, r), impel(4, 1)],
         [6, r,         impel(3, 5)],
         [7, neg(r),     andel2(2) ],
         [8, cont,       negel(6,7)]
      ],
      [9, neg(and(q, neg(r))), negint(4,8)],
      [10, cont,               negel(2,9)],
      [11, neg(p), contel(10)]
   ],
   [12, imp(and(q, neg(r)), neg(p)), impint(2, 11)]
].
```

Korrekt bevis.

```
[imp(p, imp(q, r))].

imp(and(q, neg(r)), neg(p)).

[
   [1, imp(p, imp(q, r)), premise],
   [
      [2, and(q, neg(r)), assumption],
      [
         [3, p, assumption],
         [4, imp(q, r), impel(3, 1)],
         [5, q, andel1(2)],
         [6, r, impel(5, 4)],
         [7, neg(r), andel2(2)],
         [8, cont, negel(6, 7)]
      ],
      [9, neg(p), negint(3, 8)]
   ],
   [10, imp(and(q, neg(r)), neg(p)), impint(2, 9)]
].
```