Albert-Jan Yzelman : 0216313 & Rico Slagmolen : 0331805

- Rechtvaardiging van keuze programmeertaal:
  Wij hebben gekozen voor Java omdat, Java een makkelijke taal is om in te programmeren. Java is niet zo efficiënt als Haskell of Prolog maar het gemak van de taal maakt dat erg goed. Met Java is het namelijk zo dat je kan programmeren wat je zegt. Zie README.txt voor aanwijzingen hoe het programma werkend te krijgen.

- Summiere taakverdeling:

  - Albert-Jan heeft het meeste programmeerwerk gedaan. Toen we voor het eerst samen kwamen had Albert-Jan bijna alle classes gemaakt. Tijdens onze eerste bijeenkomst hebben de eerste stappen gezet voor het Gentzen systeem. We eindigde die dag met een werkend systeem alleen was de output niet correct.
  - We gingen uiteen om zelfstandig door te werken aan het programma en twee weken later kwamen we weer bijeen. Het was Rico gelukt om er een mooie uitput uit te krijgen maar Albert-Jan had er een schepje boven op gedaan en had er een GUI bij gemaakt. We zijn toen met zijn versie verder gegaan. Er viel weinig aan te verbeteren; na een paar uur was die klaar.
  - Tijdens die bijeenkomst werd er besloten dat Rico de eerste versie van dit verslag zou maken en dat we het dan telkens heen en weer via de mail zouden sturen voor verbeteringen.

- Ontwerpkeuzes
  Wij hebben ervoor gekozen om logische functies (disjunctie, conjunctie, etc..) in aparte classes te zetten. Hierdoor zijn ze makkelijker op te roepen van uit andere classes. In de class *TracedGentzen* hebben we de regels van het Gentzen-systeem opgesteld. *TracedGentzen* bevat ook de zoekstrategie van ons programma;
  we vonden het niet nodig om dit in een andere class te zetten. De GUI is er later bijgekomen deze stelt de gebruiker in staat om zelf de verschillende Pelletier-problemen te kiezen. Dit is puur gedaan om het de gebruiker makkelijker te maken. De GUI stelt de gebruiker ook in staat om zelf er voor te kiezen of die de hele 'berekening' (trace) wilt zien. Ook kan er gekozen worden uit meerdere outputmodi.

- Zoekstrategie
  Het programma begint met het ontleden van de propositie op het hoogste niveau. We hebben gekozen om eerst de RHS oftewel het 'False' gedeelte te doen. Wij dachten namelijk dat als we het meest resolveren, we sneller tot een eventueel tegenmodel aan zouden lopen, mocht de propositie onwaar zijn. Achteraf bleek dit niet waar te zijn. Het ligt natuurlijk maar net aan welke propositie je de ATP geeft.
  Het programma gaat door tot dat er aan de RHS alleen nog maar atomen zijn, daarna begint die aan de LHS. Na elke actie aan de LHS kijkt het programma eerst of er nog steeds alleen maar atomen staan aan de RHS; dit kan natuurlijk zijn veranderd. Is dit niet het geval dan gaat het gewoon verder aan de LHS, en anders gaat het verder aan de RHS totdat daar weer alleen nog maar atomen zijn.
  Door sommige van de regels wordt er gesplit. Wanneer dit het geval is dan gaat het programma eerst door met de linkersplitsing. Het eerst doen van de linkerkant had geen reden, we hadden ook met de rechterkant kunnen beginnen.
  Het programma werkt eerst de ene splitsing af, met steeds als er gesplit wordt de linkerkant, voordat het met de rechterkant begint (pre-order traversal).
  Dit gaat zo door totdat er alleen nog maar atomen zijn in zowel de LHS als de RHS.

- Keuzes op implementatieniveau
  Er is ervoor gekozen om voor alle logische operatoren een aparte class te maken, die allen afgeleid zijn van dezelfde superklasse 'Proposition'. Een operator die op één andere propositie werkt (bv. negatie), heeft in zijn bijbehorende class ook alleen maar één andere propositie als kind. Een operator als Conjunctie werkt op twee proposities

en heeft dan ook twee kind-proposities in zijn class; de zogenaamde linker- en rechterkind. Een speciaal soort Propositie is de atoom; dit is niet een logische operator, maar een booleaanse variabele waaruit elke propositieformule is opgebouwd. Voordeel van deze datastructuur is dat het in principe een boom representeert, waarin de root de propositie is die we willen bewijzen. Deze structuur is makkelijk om mee te werken en bovendien ook efficiënt.
Een propositie in ons programma wordt naast bovenstaande dingen ook geacht een beschrijving te bevatten (in String-formaat, een atoom kan bijvoorbeeld als beschrijving de String “P” hebben). Ook wordt opgeslagen wat voor een type propositie het betreft, zodat een programmeur makkelijk kan testen op type door middel van bijvoorbeeld “proposition.isAtom” in een if-statement te zetten. Het derde en laatste verplichte ding van een propositie is een validatie-methode, welk brute-force validatie van een propositie mogelijk maakt. Dit gaat er wel vanuit dat alle atomen in de propositie zijn geïnitialiseerd op ofwel waar of onwaar. Een foutmelding wordt gegeven wanneer een atoom niet geïnitialiseerd is. Deze brute-force validatie wordt uiteraard *niet* gebruikt in het Gentzen systeem, maar is wel nodig voor ondersteuning van de waarheidstabellen-methode, die ook is geïmplementeerd en veelvuldig is gebruikt tijdens het debuggen.
Onze proposities worden in lijsten gezet (specifieker: linked lists), omdat je makkelijk een lijst kan doorlopen met een iterator; met behulp van zo’n iterator kan je de hele lijst doorlopen op zoek naar niet-atomen. Verder heeft een linked list als voordeel dat we ons geen zorgen hoeven te maken over gefixeerde groottes zoals bij een array.
Het beginnen met scannen aan de RHS had voor ons uiteindelijk geen specifieke betekenis. In principe is het een ‘kop of munt’-keuze. Zoals eerder genoemd wordt eerst de hele RHS gescand totdat er alleen nog maar atomen in zitten. Bij elke splitsing wordt eerst de linker splitsing helemaal afgewerkt voordat er aan de rechter begonnen wordt. Ook deze keuze was een ‘kop of munt’ keuze, dus zonder bijbedoelingen.
We hebben het programma zo opgebouwd dat eerst de rechtse Gentzen regels behandeld worden en dan de linkse (we begonnen immers aan de RHS). Er is niet vastgelegd met wat voor een soort propositie (conjunctie, disjunctie, implicatie, negatie en gelijkheid) we eerst beginnen; de Solver pakt simpelweg de eerste niet-atomische propositie en past daar de toepasselijke regel erop toe.
Terwijl het programma op deze manier telkens regels toepast, zien we een duidelijke boomstructuur ontstaan dankzij de recursieve aanroepen die het programma maakt. Tegelijkertijd is ons programma gewoon sequentieel, dus alle bewijsstappen (welke te beschouwen zijn als *nodes* in de bewijsboom) worden na elkaar bezocht. Omdat de methodes recursief zijn zal, wanneer een tak van de boom gesloten wordt, er door het programma heel snel worden teruggelopen tot het punt van de laatste split, of helemaal terug naar de root (in welk geval het programma klaar is). Dit teruglopen (post-order traversal) kunnen we gebruiken om een lineair, bottom-up bewijs op te stellen. In effect is het opstellen van het bewijs iets moeilijker dan hier is beschreven; dit vanwege het feit dat we alle bewijsstappen moesten nummeren en aangeven welke bewijsstappen welke andere bewijsstappen gebruiken. Voor details verwijzen we naar de volgende sectie en naar (commentaar in) de broncode zelf.
Een ander (meer cosmetisch) punt is dat het bewijs in zijn ‘pure’ vorm niet goed leesbaar was vanwege het feit dat de bewijsregels geen gelijke lengte hadden, noch dezelfde kolomgrootte. Hiervoor is de class FormatString toegevoegd; deze zorgt voor een goede uitlijning, ook qua kolommen. Het uitleggen hoe dit is geïmplementeerd gaat voorbij de scope van dit verslag, en laten we dus ook achterwege.

- Implementatie Oplosmethodes

  Hier leggen we in groter detail uit hoe TracedGentzen in elkaar steekt. De code is niet geheel doorzichtig, vandaar een aparte subsectie hierover. Onze Automated Theorem Prover ondersteunt drie verschillende methodes om propositionele functies te bewijzen; het Gentzen systeem is de 1e methode. De 2e methode is de truthtables-methode, welk alle atomen op alle mogelijke manieren instantieert, en telkens valideert met die verschillende instantiaties. Wordt er geen tegenmodel gevonden, dan is de

propositie waar. De 3e methode is linear KE (KE exclusief DC); het plan was volledige KE te implementeren, maar het programmeren van de DC-regel ging helaas niet probleemloos. Met meer tijd hadden we wellicht de problemen op kunnen lossen, maar nu kwamen we dus niet verder dan de incomplete linear KE (TracedKE.java). Deze werkt overigens ongeveer hetzelfde als TracedGentzen, alleen dan met een andere set regels. Maar laten we terug te keren naar de beschrijving van onze Gentzen-implementatie:

de initiële propositie (de te bewijzen propositie) wordt toegevoegd aan een lijst van proposities, de RHS. Dit omdat we op zoek gaan naar een tegenmodel, zoals beschreven in het dictaat. De LHS wordt ingesteld op een lege propositielijst. Vervolgens wordt de main-loop gestart door de methode iterate aan te roepen.

Deze zoekt eerst in de RHS naar een niet-atomische propositie. Als er daar geen te vinden is, wordt er gezocht in de LHS. De gevonden niet-atomische propositie wordt vervolgens uit de LHS of RHS verwijderd, en zowel de propositie als de twee lijsten worden vervolgens gebruikt om de geïmplementeerde Gentzen-regels (links-/\, rechts-=>, enz) op toe te passen. Uit deze regels worden een nieuwe LHS en RHS afgeleid, waarop gelijk weer de methode iterate wordt aangeroepen. Ook is het mogelijk dat er een splitsing volgt uit zo'n regel; in dat geval worden de 2 verschillende LHS/RHS paren na elkaar geresolveerd.

Dit gaat door totdat alle proposities in de LHS en RHS atomen zijn. Is dit het geval, dan wordt er gecontroleerd of wat er staat een axioma is. Als dit zo is dan wordt de tak gesloten, en als het geen axioma is dan hebben we een tegenmodel gevonden. Als alle takken (van alle splitsingen) gesloten raken, kunnen we een bewijs opstellen.

Het opstellen van het bewijs gebeurt niet achteraf, maar wordt on-the-fly gedaan terwijl het programma door de verschillende Gentzen-regels en de iterate-methode heen gaat. Dit is gedaan omdat de trace al bijna een goed bewijs was, maar dan met heel veel extra informatie. Door sommige dingen die de trace als output geeft ook te laten opslaan als een bewijsregel, is een bewijs makkelijk on-the-fly geconstrueerd. Zie de broncode voor details. Moeilijkheid was wel om elk deelbewijs (elke toepassing van een regel) een apart nummer te geven; dit is nu klaargespeeld door een globale teller bij te houden die bij elke nieuwe bewijsstap geïncrementeerd wordt.

NB: al het schrijven naar de output gebeurt niet direct met System.out, maar door gebruik van de speciale klasse outputcontroller. Deze klasse zorgt ervoor dat een gebruiker kan kiezen hoe en waar hij zijn trace en bewijs wilt hebben. (Er is zelfs een aparte outputcontroller voor de trace, en een andere voor het bewijs).

- Output voor probleem 9 uit Pelletier's lijst.
  Zie bijlage ‘bijlageAR’.

- Listing van een complete trace voor probleem 9 uit Pelletier's lijst.1
  Zie bijlage ‘bijlageAR’

- Formule die je stellingenbewijzer niet meer aankan
  (niet binnen 1 minuut kan beslissen).
  Het is ons niet gelukt om een correcte formule te bedenken die er langer dan een minuut over duurt. We zijn zelfs zo ver gegaan dat we een automatische formulegenerator hebben gemaakt (Generator.java). Maar deze produceert met grote waarschijnlijkheid ‘foute’ formules. Het programma heeft dat snel door (vindt tegenmodellen) maar als we hem de rest van de boom laten doorzoeken, doet hij er wel langer dan een minuut over.
  Verder hebben we gekeken naar die gegenereerde formules en ontdekte we dat deze formules gauw 3 regels lang zijn en vooral bestaan uit equivalenties en implicaties.

Alsof dat niet genoeg was gebruikten we ook acht verschillende atomen binnen de betreffende gegenereerde propositie.
Het programma lost de door ons zelf bedachte formules op binnen de aangegeven minuut. Overigens zal ons programma zelf een formule genereren in plaats van een pelletier-probleem gebruiken zodra er als op te lossen (pelletier) probleemnr. een 0 wordt opgegeven (de normale range is 1 t/m 17). Deze optie is alleen beschikbaar via de commandline en niet de GUI. Typ 'java ATP' voor een korte handleiding over hoe ons programma op de command line te laten werken.

- Conclusie

  We zijn van mening dat ons programma goed werkt.
  Alhoewel sommige keuzes ad-hoc gemaakt waren, leken ze toch goede keuzes te zijn geweest.
  Een positief punt was ook dat we geen kloppende formule hebben kunnen bedenken die er langer dan een minuut over zal doen; deze zal er echter natuurlijk wel bestaan, al is het alleen zeer lange aaneenschakeling van iets dat waar is (een lange conjunctie van 'p of niet-p' bijvoorbeeld).
  Er zijn echter nog wel een paar dingen die we er in hadden wil zien maar door tijd gebrek is dat ons helaas niet gelukt.
  Zo hadden we ons programma willen verbeteren met KE en willekeur of heuristiek inbouwen wat betreft de selectie van de volgende propositie waar op moet worden gewerkt. Misschien dat we er in onze vrije tijd nog aanzetten, en uiteindelijk zelfs $SAKE_K$ geïmplementeerd krijgen!