

Stellingbewijzer voor propositiologica

Thomas Markus - 0312770
Matthijs Melissen - 0423165

13 januari 2006

Keuze implementatietaal

We hebben onze stellingbewijzer in de scriptingtaal Ruby geschreven.

We hebben bewust voor een imperatieve programmeertaal en niet voor een functionele taal gekozen, omdat we de verwachting hadden dat debuggen in bijvoorbeeld Haskell moeilijker zou zijn. Bovendien hadden we weinig ervaring met deze taal. Ook het gebruik van een logische taal zoals Prolog leek ons geen goed idee, omdat Prolog een bepaalde structuur afdwingt die wat onoverzichtelijk wordt bij grotere programma's. Verder is het wat omslachtig om de control flow van je programma zelf te bepalen, waar het uiteindelijk bij het maken van deze stellingbewijzer wel om draait.

Van de vele mogelijke imperatieve talen hebben we uiteindelijk voor Ruby gekozen, omdat deze taal een aantal praktische en theoretische voordelen heeft. Allereerst is de taal volledig object-georiënteerd. Ruby is dynamisch getypeerd, waardoor we ons op de structuur van een programma konden richten en niet op het feit wat voor type iets precies is. Tot slot is Ruby een scripttaal, waardoor geen extra compilerstap nodig is.

Taakverdeling

Dankzij het gebruik van een functie waarmee meerdere gebruikers tegelijk aan dezelfde bestanden kunnen werken (SubEthaEdit op Mac OS X via Bonjour), was een stricte taakverdeling niet echt nodig. Je kunt zeggen dat we van tijd tot tijd een beetje aan Extreme Programming hebben gedaan (de één programmeert en de ander geeft commentaar).

Nadat de eerste stellingbewijzer die gebruik maakte van de standaard tableauxregels af was en goed getest, hebben we de Rules-klasse voor twee regels samen herschreven om ons ontwerp te testen. Daarna heeft Matthijs de andere regels behalve de DC snel geïmplementeerd. Thomas richtte zich toen op de Engine-klasse.

Ontwerpkeuzes

Omdat het plan was om eerst een werkende tableaugebaseerde stellingbewijzer te maken en deze, indien er nog tijd was, uit te breiden tot een SAKE-stellingbewijzer hebben we geprobeerd de regels goed te scheiden van de rest

van het programma. Dit ontwerp heeft zich zeker terugbetaald bij de implementatie van SAKE.

Ons programma bestaat uit de volgende klassen:

- (1) Engine - De klasse die de hoofdlus van het programma en het regelen van de uitvoer bevat.
- (2) Formula - De datastructuur voor formuleobjecten, inclusief parser.
- (3) Node - Een uitbreiding van de *Hash*. In deze klasse worden sequenten opgeslagen.
- (4) RuleBase - Basisklasse voor alle niet-splitsende regels.
- (5) Ook alle regels zijn een eigen klasse.

Verder is er een bestand genaamd `main.rb` welke de Engine klasse initialiseert en de opdrachtregel-opties doorgeeft hieraan. Ook wordt de bestede tijd gemeld.

Zoekstrategie

We doorzoeken de boom van sequenten met behulp van depth-first search, want bij bijvoorbeeld het gebruik van breadth-first search met takken waarin het bewijs diep ligt zou het aantal sequenten dat moet worden bijgehouden exponentieel groter zijn. Bij depth-first search is dit niet het geval.

Keuzes op implementatieniveau

Beschrijving datastructuur *Node*-objecten:

- left - De onware kant van de sequent. Dit is een array van *Nodes*.
- right - De ware kant van de sequent. Dit is een array van *Nodes*.
- blacklist - Een array van *Formulas* waar al op gesplitst is.
- step - Een uniek nummer dat de plek van de *Node* in het bewijs aangeeft.

Beschrijving datastructuur *Formula*-objecten:

- operator (optioneel) - Het hoofdconnectief van de formule.
- argument1 - Het eerste argument van de formule. Dit is ook zelf weer een *Formula*-object.
- argument2 (optioneel) - Het tweede argument van de formule. Ook dit is een *Formula*-object.

Beschrijving hoofdlus:

Alle regel-objecten zitten in de array *rules* in de *Engine* klasse.

checkRules()

Eerst wordt *branchCloses()* aangeroepen of de tak wel of niet sluit. De *rules* array wordt doorlopen en van iedere regel wordt de *apply* methode aangeroepen.

Het resultaat van deze methode kan een array van nieuwe sequenten zijn (het resultaat van de toepassing van een regel) of een boolean FALSE. Het resultaat wordt dan gebruikt om te bepalen of de regel geslaagd is of niet. Als dit het geval is, dan stopt de methode *checkRules* en gaat terug naar *prove()* waar het resultaat verder wordt verwerkt. In het andere geval wordt geprobeerd de volgende regel toe te passen. Als alle regels geprobeerd zijn en ze allemaal niet toepasbaar waren dan hebben we een tegenmodel te pakken. Voor de rest van de structuur, zie de gegenereerde Ruby-documentatie in de /doc map.

Limieten van de stellingbewijzer

De volgende formule kan op de computers bij Wijsbegeerte niet binnen een minuut worden uitgerekend:

```
((((((((((((a=b)=c)=d)=e)=f)=g)=h)=i)=j)=k)=(a=(b=(c=(d=(e=(f=(g=(h=(i=(j=k))))))))))))))
```

De grootste boosdoener van de vertraging is de veelvuldige toepassing van de relatief computationeel dure DC regel. De sequent moet dan gedupliceerd worden en er moet gezocht worden naar de langste formule.

Natuurlijk is het vergroten van de beschikbare rekenkracht een mogelijk middel om de limieten van ons programma op te rekken. De computers bij wijsbegeerte zijn wat aan de trage kant (G4-800mhz in BG043). Bovenstaande formule doet er ruwweg 65 seconden over, maar op bijvoorbeeld een 2.4Ghz Solaris Opteron server duurt het sluiten van alle takken slechts 16 seconden. Natuurlijk kan ook voor deze snellere machine een formule gevonden worden waar langer dan 60 seconden voor nodig is, bijvoorbeeld door de bovenstaande formule met een paar letters uit te breiden.

Door het gebruik van depth-first search is het geheugengebruik geen beperkende factor.

Extra

Ons script kan vanaf de opdrachtregel als volgt worden aangesproken:

```
ruby main.rb formula [-s | -d]
```

Dit script geeft een bewijs van formula als dit bewijsbaar is, of een tegenmodel als formula niet bewijsbaar is. De logische connectieven die gebruikt kunnen worden zijn & (conjunctie), | (disjunctie), ~ (negatie), > (implicatie) en = (bi-implicatie).

Opties:

- **-s silent**
Toont geen tussentijdse statusinformatie.
Deze optie levert een kleine versnelling in het zoekproces op.
- **-d debug**
Toont elke stap van de stellingbewijzer.
Deze optie vertraagt het zoekproces aanzienlijk.

Wanneer geen opties worden meegegeven, wordt wel tussentijdse statusinformatie (wanneer de afleiding langer dan 2 seconden duurt) getoond, maar er wordt geen debuginformatie weergegeven.

Conclusie

De stellingbewijzer zou op de volgende punten nog verbeterd kunnen worden:

- De lengte van een formule zou gecached kunnen worden.
- De formules worden niet op lengte gesorteerd en gehouden, hierdoor is het zoeken van de langste formule (nodig bij de toepassing van de heuristiek bij de DC regel) kostbaarder dan strikt noodzakelijk is. Door bijvoorbeeld insertion sort te implementeren in een methode en dan deze methode te gebruiken ipv gewoon *push()* van de array te gebruiken is dit te verbeteren.
- Om echt snelle resultaten te krijgen, zou hetzelfde algoritme geïmplementeerd kunnen worden in een snellere programmeertaal. Met name het gebruik van een gecompileerde taal valt aan te raden.

Output Pelletier's problem 9

Formula to be proven: $((p|q)\&(\sim p|q)\&(p|\sim q))\>(\sim(\sim p|\sim q))$

Formula after making the brackets explicit: $((p|q)\&((\sim p|q)\&(p|\sim q))\>((\sim(\sim p)|(\sim q))))$

ImplicationRight

$((p|q)\&((\sim p|q)\&(p|\sim q))) \Rightarrow \sim(\sim p|\sim q)$

NegationRight

$((p|q)\&((\sim p|q)\&(p|\sim q))), (\sim p|\sim q) \Rightarrow$

ConjunctionLeft

$(\sim p|\sim q), (p|q), ((\sim p|q)\&(p|\sim q)) \Rightarrow$

ConjunctionLeft

$(\sim p|\sim q), (p|q), (\sim p|q), (p|\sim q) \Rightarrow$

DC

$(\sim p|\sim q), (p|q), (\sim p|q), (p|\sim q), \sim q \Rightarrow$

$(\sim p|\sim q), (p|q), (\sim p|q), (p|\sim q) \Rightarrow \sim q$

NegationLeft

$(\sim p|\sim q), (p|q), (\sim p|q), (p|\sim q) \Rightarrow q$

DisjunctionLeft2

$(\sim p|\sim q), (\sim p|q), (p|\sim q), p \Rightarrow q$

DisjunctionLeft2

$(\sim p|\sim q), (p|\sim q), p, \sim p \Rightarrow q$

NegationLeft

$(\sim p|\sim q), (p|\sim q), p \Rightarrow q, p$

Branch closes...

DisjunctionLeft2

$(p|q), (\sim p|q), (p|\sim q), \sim p \Rightarrow \sim q$

DisjunctionLeft2

$(p|q), (\sim p|q), \sim p, p \Rightarrow \sim q$

NegationLeft

$(p|q), (\sim p|q), p \Rightarrow \sim q, p$

Branch closes...

Preparing output...

0: $(p|q), (\sim p|q), p \Rightarrow \sim q, p$

Axiom

1: $(p|q), (\sim p|q), \sim p, p \Rightarrow \sim q$

NegationLeft(0)

2: $(p|q), (\sim p|q), (p|\sim q), \sim p \Rightarrow \sim q$

DisjunctionLeft2(1)

3: $(\sim p|\sim q), (p|\sim q), p \Rightarrow q, p$

Axiom

4: $(\sim p|\sim q), (p|\sim q), p, \sim p \Rightarrow q$

NegationLeft(3)

5: $(\sim p|\sim q), (\sim p|q), (p|\sim q), p \Rightarrow q$

DisjunctionLeft2(4)

6: $(\sim p|\sim q), (p|q), (\sim p|q), (p|\sim q) \Rightarrow q$

DisjunctionLeft2(5)

7: $(\sim p|\sim q), (p|q), (\sim p|q), (p|\sim q) \Rightarrow \sim q$

DisjunctionLeft2(2)

8: $(\sim p|\sim q), (p|q), (\sim p|q), (p|\sim q), \sim q \Rightarrow$

NegationLeft(6)

9: $(\sim p|\sim q), (p|q), (\sim p|q), (p|\sim q) \Rightarrow$

DC(8,7)

10: $(\sim p|\sim q), (p|q), ((\sim p|q)\&(p|\sim q)) \Rightarrow$

ConjunctionLeft(9)

11: $((p|q)\&((\sim p|q)\&(p|\sim q))), (\sim p|\sim q) \Rightarrow$

ConjunctionLeft(10)

12: $((p|q)\&((\sim p|q)\&(p|\sim q))) \Rightarrow \sim(\sim p|\sim q)$

NegationRight(11)

13: $\Rightarrow ((p|q)\&((\sim p|q)\&(p|\sim q)))\>\sim(\sim p|\sim q)$

ImplicationRight(12)

Total time spent: 0.026 seconds, of which 0.024 on proving
and 0.001 on generating output.