Christoph Lüth
Marcus Ermler
Christian Maeder
Martin Ring
Diedrich Wolter

11. Übungsblatt

Ausgabe: 16.01.13
Abgabe: 25.01.13

Eine aussagenlogische Formel ϕ ist *erfüllbar*, wenn es eine Belegung σ der Variablen in ϕ gibt, die ϕ wahr werden läßt. Die Frage, ob für eine gegebene Formel ϕ eine solches σ existiert, ist das *Erfüllbarkeitsproblem* (SAT-Problem), und es spielt eine wichtige Rolle im Schaltkreisentwurf, in der Logistik, und in vielen anderen Anwendungsbereichen, da sich Fragestellungen oft leicht in aussagenlogischer Form formulieren lassen.

Der bekannte Informatik-Pionier Sir Michael Philip Jagger vermutete, dass SAT-Problem sei unlösbar (‘*I can't get no satisfaction*‘), aber das ist zu kurz gegriffen: tatsächlich ist das Problem NP-vollständig. Aufgrund seiner Wichtigkeit ist es allerdings gut untersucht, und es existieren gute Heuristiken und Werkzeuge (SAT-Solver), die auch große Instanzen dieses Problems schnell lösen können. In diesem Übungsblatt wollen wir den Algorithmus implementieren, auf dem die meisten existierenden SAT-Solver basieren.

11.1 *Konjunktive Normalformen* — *4 Punkte*

Gängige SAT-Solver wie MiniSat[1] erwarten als Eingabe nicht eine beliebige aussagenlogische Formel, sondern eine in *konjunktiver Normalform* (kurz: *KNF*),[2] die wie folgt definiert sind:

- Eine *KNF* besteht aus einer Konjunktion von Klauseln,
- eine *Klausel* ist eine Disjunktion von Literalen, und
- *Literale* sind negierte oder nichtnegierte *Variablen*.

Ein Beispiel für eine KNF ist die Formel: $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge x_2$. Sie besteht aus 3 Klauseln, 6 Literalen und 4 Variablen.

1. Modellieren Sie einen Datentyp CNF für konjunktive Normalformen, einen Datentyp Binding für Variablenbelegungen und einen Datentyp Satisfiability für Erfüllbarkeit (mit Variablenbelegungen) und Unerfüllbarkeit. Damit verschiedene Arten von Variablen möglich sind (Integer, selbstdefinierte Datentypen, usw.), sollten alle drei Datentypen polymorph sein.

2. Um Ihren SAT-Solver später mit großen SAT-Instanzen testen zu können, macht es Sinn auf bereits vorliegende Instanzen in KNF zurückzugreifen. Industrielle SAT-Instanzen werden häufig im sogenannten *DIMACS*-KNF-Format[3] gespeichert.

   Das DIMACS-KNF-Format ist folgendermaßen aufgebaut: Kommentarzeilen stehen nur am Anfang der Datei. Jede Kommentarzeile beginnt mit einem c. Nach den Kommentarzeilen folgt eine Zeile in der das Format, die Anzahl an Variablen und die Anzahl an Klauseln angegeben wird, gekennzeichet wird diese Zeile mit einem p am Anfang. Danach folgen die Klauseln der KNF. Literale werden durch ganze Zahlen (ohne 0) ausgedrückt. Ein positive Zahl entspricht hierbei einem nichtnegierten Literal und eine negative Zahl einem negierten Literal. Jede Klausel schließt mit einer 0 ab. Das obige Beispiel kann dann im DIMACS-KNF-Format wie folgt formuliert werden:

   ```
   c Ein Kommentar
   p cnf 4 3
   1 2 0
   -1 3 -4 0
   2 0
   ```

[1] `http://www.minisat.se`

[2] Durch die *Tseitin-Transformation* kann allerdings eine beliebigen aussagenlogische Formel mit linearem Aufwand in eine äquivalente KNF transformiert werden.

[3] Siehe auch `www.cfdvs.iitb.ac.in/download/Docs/verification/papers/BMC/JT.ps`

Implementieren Sie eine Funktion

```
dimacs :: FilePath → IO (CNF Int)
```

die bei Eingabe eines Dateipfades die zugehörige DIMACS-Datei öffnet und die KNF einliest.

Auf der Webseite steht eine Archivdatei `uebung11.zip` bereit, welche verschiedene Dateien im DIMACS-Format enthält, mit denen Sie ihre Funktion testen können.

11.2 *SAT-Solving via DPLL-Algorithmus* *12 Punkte*

Im folgenden soll nun der sogenannte DPLL-Algorithmus implementiert werden, mit dem Formeln in KNF in vielen Fällen schnell gelöst werden können. Er ist benannt nach Martin Davis, Hilary Putnam, George Logemann und Donald W. Loveland, die ihn schrittweise in den 60er Jahren des letzten Jahrhunderts entwickelten [2, 1].

1. Wir beginnen mit einer Funktion

   ```
   evaluate :: CNF a → Binding a → Bool.
   ```

   die eine KNF mit einer gegebenen Variablenbelegung auswertet. Diese Funktion können Sie nutzen, um die Ergebnisse Ihres SAT-Solvers zu überprüfen: wenn dieser eine Variablenbelegung zurückliefert, dann muss diese die Eingabe-KNF zu True auswerten.

2. Für den eigentlichen Solver implementieren Sie zuerst eine Funktion[4]

   ```
   simplify :: CNF a → Binding a → Maybe (CNF a)
   ```

   die eine KNF gemäß einer Variablenzuweisung vereinfacht. Hilfreich sind die folgenden Vereinfachungsregeln, wobei f eine beliebige Formel ist: $\mathit{True} \wedge f = f$, $\mathit{True} \vee f = \mathit{True}$, $\mathit{False} \wedge f = \mathit{False}$, $\mathit{False} \vee f = f$, $f \wedge f = f$ und $f \vee f = f$. Die Rückgabe ist ein Maybe, weil sich die KNF auch als nicht erfüllbar herausstellen kann.

3. Damit eine KNF erfüllbar wird, muss jede Klausel zu True auswerten. Besteht eine Klausel nur aus einem Literal, muss dieses folglich so belegt werden, dass die Klausel True ergibt. Solche Klauseln werden auch *Unit-Clauses* genannt.

   Implementieren Sie eine Funktion

   ```
   unitPropagation :: CNF a → (Maybe (CNF a), Binding a)
   ```

   die für eine KNF prüft, ob sie eine Unit-Clause enthält. Ist dies der Fall wird die Formel entsprechend der erhaltenen Variablenzuweisung mit simplify vereinfacht und die daraus resultierende Formel auf weitere Unit-Clauses überprüft. Ansonten lässt sich nichts weiter vereinfachen und die KNF wird zurückgeben. Beachten Sie, dass auch hier Klauseln unerfüllbar sein können. Desweiteren müssen die vorgenommenen Variablenbelegungen abschließend zurückgegeben werden.

4. Um die Formel auf Erfüllbarkeit zu überprüfen werden den Variablen schrittweise Belegungen zugewiesen. Sowohl die Auswahl der Variablen als auch die Zuweisung eines geeigneten Wertes ist eine wichtiger Baustein zum schnellen Lösen von Formeln. Implementieren sie hierfür eine Funktion

   ```
   decide :: CNF a→ Binding a
   ```

   die aus den in der KNF vorkommenden Variablen eine auswählt, für diese eine Zuweisung bestimmt und diese dann zurückgibt (das Ergebnis enthält also nur die Belegung einer einzelnen Variablen). Entscheiden Sie sich für eine Strategie, zum Beispiel: zufälliges Bestimmen einer Variablen und ihrer Belegung (wie im original DPLL-Algorithmus) oder die unbelegte Variablen verwenden, die im Vergleich zu allen anderen unbelegten Variablen am häufigsten in Klauseln auftritt; hierbei wird sie mit True belegt, wenn die Variable öfter in ihrer nichtnegierten als in ihrer negierten Form auftritt, und mit False im anderen Fall.

[4] Die für diese Aufgabe vorgebenen Signaturen können für Ihre Lösung angepasst werden; insbesondere werden Sie noch Klasseneinschränkungen (Eq a, Ord a) auf den Typvariablen benötigen.

5. Der DPLL-Algorithmus ist ein auf Backtracking basierender Suchalgorithmus. Die darin enthaltenen Verzweigungen sollen nun in einer Funktion

   ```
   branch :: CNF a → Binding a→ Satisfiability a
   ```

   realisiert werden und zwar folgendermaßen: (1) Es wird eine (bislang unbelegte) Variable mit decide ermittelt und die Formel demgemäß vereinfacht; (2) Liefert dpll (siehe unten) für die vereinfachte Formel eine erfüllende Belegung, dann wird diese als Ergebnis zurückgegeben; (3) hat (2) keine erfüllende Belegung geliefert, wird der Wert der gewählten Variablenbelegung negiert und dpll mit diesem aufgerufen; (4) Liefert auch (3) keine erfüllende Belegung, ist die Formel nicht erfüllbar. Beachten Sie, dass die gewählte Variable in späteren Schritten nicht mehr ausgewählt werden kann.

6. Nun können Sie den DPLL-Algorithmus in der Funktion dpll implementieren:

   ```
   dpll :: CNF a → Satisfiability a
   ```

   Auf die eingegebene KNF wird unitPropagation angewandt, was drei verschiedene Ergebnisse liefert: (1) Die KNF ist leer, dann ist die Formel erfüllbar und die Variablenbelegung wird zurückgegeben; (2) Es wird Nothing ermittelt, dann ist die Formel unerfüllbar; (3) Die KNF besteht noch aus mindestens einer Klausel, dann werden mit branch weitere Variablenbelegungen ermittelt.

7. Implementieren Sie abschließend eine Funktion

   ```
   satsolver :: CNF a → IO ()
   ```

   die dpll aufruft und im Falle der Erfüllbarkeit die Variablenbelegungen, die Anzahl der Klauseln und Literale und die Berechnugszeit in der Kommandozeile ausgibt. Beachten Sie, dass es im Falle der Erfüllbarkeit möglich ist, dass Variablen keine Belegung haben (bei $x_1 \vee x_2$ würde nur einer der beiden Variablen einen Wert zugewiesen bekommen), wählen Sie dann eine Belegung.

11.3 *Untersuchung des Laufzeitverhaltens* *4 Punkte*

In dieser Aufgabe untersuchen wir das Laufzeitverhalten Ihrer Implementierung des DPLL-Algorithmus für verschiedene SAT-Instanzen. Insbesondere große SAT-Instanzen sind hier von Interesse, weil Sie so Ihren SAT-Solver an seine Grenzen bringen können. Große Instanzen manuell zu erzeugen ist jedoch sehr aufwändig und wenig sinnvoll, daher verwenden Sie die in der bereitgestellten Archivdatei `uebung11.zip` enthaltenen Normalformen, um diese auf Erfüllbarkeit zu untersuchen.

Listen Sie die Ergebnisse der einzelnen Instanzen in einer Tabelle der folgenden Form auf:

SAT-Instanz	SAT?	#Klauseln	#Literale	Berechnungszeit (in sec)
Name	JA	100	1000	10
...				

? *Verständnisfragen*

1. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?
2. Ist eine in allen Argumenten als strikt erkannte und übersetzte Funktion immer schneller in der Ausführung als dieselbe als nicht-strikt übersetzte Funktion?
3. Warum kann ich die Funktion seq nicht in Haskell definieren?

Literatur

[1] M. Davis, G. Logemann, D. W. Loveland: *A machine program for theorem-proving*. Commun. ACM **5**(7): 394-397 (1962)

[2] M. Davis, H. Putnam: *A Computing Procedure for Quantification Theory*. J. ACM **7**(3): 201-215 (1960)