Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Noschinski, Lars Hupel, Dr. Jasmin Blanchette

WS 2013/14
11.12.2013
Abgabe: 18.12.2013, 15:30

Einführung in die Informatik 2
9. Übung

Aufgabe G9.1 (Bi-)Implikation

Erweitern Sie den Datentyp `Form` um zwei weitere binäre Infix-Konstruktoren: die Implikation `:->:` und die Bi-Implikation `:<->:`. Für alle Wertebelegungen soll der Wert von `f1 :->: f2` gleich dem Wert von `Not f1 :|: f2` und der Wert von `f1 :<->: f2` gleich dem Wert von `(f1 :->: f2) :&: (f2 :->: f1)` sein.

Passen Sie die Funktionen `show`, `eval`, `vars`, `isSimple`, `simplify`, sowie `arbitrary` für die QuickCheck-Test-Erzeugung entsprechend an. Die Negation einer (Bi-)Implikation gilt im Sinne von `isSimple` (bzw. `simplify`) als nicht einfach (bzw. soll von `simplify` "hineingeschoben" werden).

Schreiben Sie einen QuickCheck-Eigenschaft `prop_simplify_sound`, die überprüft, ob die Werte der Formeln `p` und `simplify p` für beliebige Wertebelegungen übereinstimmen. Überlegen Sie warum es nicht zielführend wäre die gesamte Wertebelegung (vom Typ `[(Name, Bool)]`) von QuickCheck zufällig erzeugen zu lassen, und wie man das Problem umgehen kann.

Wichtig: Verwenden Sie für diese Aufgabe eine Kopie der Schablone `Form.hs` (z.B. mit dem Modul-Namen `ExtForm`), da Sie in den Hausaufgaben mit der Original-Schablone arbeiten sollen.

Aufgabe G9.2 Arithmetische Ausdrücke

Definieren Sie einen Datentyp `Arith` zur Darstellung von arithmetischen Ausdrücken, bestehend aus Addition, Multiplikation, Integer-Konstanten und Variablen.

Schreiben Sie eine Funktion `evalA :: [(Name, Integer)] -> Arith -> Integer`, um solche Ausdrücke bezüglich einer gegebenen Variablenbelegung auszuwerten.

Sollte in einer Formel eine Variable vorkommen, die in der Variablenbelegung nicht aufgeführt wird, so soll der Wert mit 0 angenommen werden.

Aufgabe G9.3 Die Stunde der Wahrheit(stabelle)

Wir wollen die Auswertung einer booleschen Formel in einer Tabelle darstellen. Für jede in der Formel vorkommende Variable gibt es eine Spalte, die mit den möglichen Werten (`True` bzw. `False`, ausgegeben als `T` bzw. `F`) so ausgefüllt ist, dass in jeder Zeile eine Wertebelegung abzulesen ist. Die Formel selbst und ihre Auswertung für die jeweilige Belegung steht in der letzten Spalte. Beispiel:

```
P  Q  | ((P  &  Q)  |  (~P))
-  -  - ----------------
F  F  |         T
```

```
F T |         T
T F |         F
T T |         T
```

Schreiben Sie eine Funktion `showTable :: Form -> String`, so dass `putStrLn . showTable` eine solche Wahrheitstabelle ausgibt. Die Ausgabe der Funktion `showTable` muss also Zeilenumbrüche an den richtigen Stellen im String enthalten. Beispiel:

```
showTable (Var "P" :&: Var "Q" :|: Not (Var "P")) ==
  "P Q | ((P & Q) | (~P))\n- - - ----------------\n\
  \F F |         T         \nF T |         T         \n\
  \T F |         F         \nT T |         T         \n"
```

Verwenden Sie die im Beispiel dargestellten Tabellenbegrenzungen und zentrieren Sie die Ausgabe der booleschen Werte (`T` bzw. `F`) in jeder Spalte (falls notwendig).

Aufgabe H9.1 Nie In Ordnung (4 Punkte)

Beweisen Sie die Gleichung

```
inorder = reverse . niorder
```

per Induktion über Bäume. Benutzen Sie das aus der Vorlesung bekannte "induction template" (Folie 124; Induktion über Bäume ist beschrieben auf Folie 240ff). Geben Sie zusätzlich die Gleichungen, die Sie als Induktionshypothesen (IH_1 und IH_2) benutzen, explizit an.

```
data Tree a = Empty | Node a (Tree a) (Tree a)

inorder Empty = []
inorder (Node a l r) = inorder l ++ [a] ++ inorder r

niorder Empty = []
niorder (Node a l r) = niorder r ++ [a] ++ niorder l

reverse [] = []
reverse (x : xs) = reverse xs ++ [x]

-- Lemmas
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)           -- append_assoc
reverse (xs ++ ys) = reverse ys ++ reverse xs -- reverse_append
```

Verwenden Sie in jedem Schritt *nur eine* dieser Gleichungen und Induktionshypothesen und vergessen Sie nicht, den Namen der angewendeten Gleichung anzugeben.

Aufgabe H9.2 Disjunktive Normalform (8 Punkte)

> **Wichtig:** Für die Bearbeitung dieser Aufgabe legen Sie die `Form_9.hs` in das gleiche Verzeichnis wie die `Exercise_9.hs`. Bei der Abgabe hingegen brauchen Sie nur die `Exercise_9.hs` auf den Übungsserver hochzuladen.

Gegeben seien die folgenden Definitionen:

- Ein *Atom* (vom Typ `Form`) ist entweder F, T oder eine Variable x (d.h. `Var` x).
- Ein *Literal* ist ein Atom a oder seine Negation $\neg\, a$ (d.h. `Not` a).
- Eine *Coklausel* ist eine n-stellige, nach links geklammerte Konjunktion $((\cdots((l_1 \wedge l_2) \wedge l_3) \wedge \cdots) \wedge l_{n-1}) \wedge l_n$ von Literalen l_j (für $n \geq 1$).
- Eine *Formel in disjunktiver Normalform* (*DNF*) ist eine n-stellige, nach links geklammerte Disjunktion $((\cdots((c_1 \vee c_2) \vee c_3) \vee \cdots) \vee c_{n-1}) \vee c_n$ von Coklauseln c_j (für $n \geq 1$).

Beispiele:

- $\neg\, \mathsf{F}$ ist ein Literal, eine Coklausel und eine DNF-Formel.
- $\neg\, \neg\, \mathsf{F}$ ist kein Literal (wegen der doppelten Negation).
- $\mathsf{F} \wedge \mathsf{T}$ ist eine Coklausel und eine DNF-Formel.
- $\mathsf{F} \vee \mathsf{T}$ ist eine DNF-Formel.
- $\mathsf{x} \vee (\mathsf{y} \wedge \neg\, \mathsf{z})$ ist eine DNF-Formel.
- $\mathsf{x} \vee ((\mathsf{y} \wedge \neg\, \mathsf{z}) \wedge \mathsf{w})$ ist eine DNF-Formel.
- $\mathsf{x} \vee (\mathsf{y} \wedge (\neg\, \mathsf{z} \wedge \mathsf{w}))$ ist keine DNF-Formel (wegen der nicht kanonischen Klammerung).

Aufgaben:

1. Schreiben Sie eine Funktion `isDnf :: Form -> Bool`, die `True` zurückgibt, wenn die gegebene boolesche Formel in DNF ist und ansonsten `False`.

   Hinweis: Schreiben Sie Hilfsfunktionen `isAtom`, `isLiteral` und `isCoclause` und testen Sie sie gründlich, bevor Sie sie in `isDnf` kombinieren.

2. Schreiben Sie eine Funktion `pushNot :: Form -> Form`, die die gegebene Formel so umschreibt, dass alle Negationen direkt auf Atome angewandt werden. Verwenden Sie die folgenden Eigenschaften:

$$\neg\, \neg\, \varphi \equiv \varphi \qquad \neg\, (\varphi \wedge \chi) \equiv (\neg\, \varphi \vee \neg\, \chi) \qquad \neg\, (\varphi \vee \chi) \equiv (\neg\, \varphi \wedge \neg\, \chi)$$

3. Schreiben Sie eine Funktion `pushAnd :: Form -> Form`, die Konjunktionen ($\wedge$) über Disjunktionen ($\vee$) distribuiert:

$$(\varphi \vee \chi) \wedge \psi \equiv (\varphi \wedge \psi) \vee (\chi \wedge \psi) \qquad \varphi \wedge (\chi \vee \psi) \equiv (\varphi \wedge \chi) \vee (\varphi \vee \psi)$$

4. Schreiben Sie eine Funktion `balance :: Form -> Form`, die die Klammerung von Konjunktionen und Disjunktionen mit den Klammern möglichst nach links normalisiert:

$$\varphi \wedge (\chi \wedge \psi) \equiv (\varphi \wedge \chi) \wedge \psi \qquad \varphi \vee (\chi \vee \psi) \equiv (\varphi \vee \chi) \vee \psi$$

5. Schreiben Sie eine Funktion `toDnf :: Form -> Form`, die eine beliebige Formel in eine äquivalente DNF-Formel übersetzt.

   Hinweis: Die einfachste Lösung verwendet `pushNot`, `pushAnd` und `balance` sowie die in Aufgabe H5.2 eingeführte Funktion `fixpoint`. Die Grundidee besteht darin, `pushNot`, `pushAnd` und `balance` wiederholt anzuwenden, bis die Formel sich nicht mehr ändert.

6. Geben Sie QuickCheck-Tests für `toDnf` an, die die wichtigsten Eigenschaften der Funktion abdecken.

Achtung: Damit der letzte Punkt zur Aufgabe wird, sind die offiziellen Tests absichtlich unvollständig. Verlassen Sie sich also nicht blind auf diese.

Aufgabe H9.3 Baumorigami (3 Punkte)

1. Definieren Sie folgende Funktionen:

```
treeSum :: Tree Int -> Int
treeMin :: Ord a => a -> Tree a -> a
evens :: Tree Int -> [Int]
```

   Die Funktion `treeSum` soll die Summe aller Elemente eines Baumes berechnen. `treeMin` x soll das Minimum von allen Elementen und x berechnen. Zuletzt soll `evens` eine Liste aller geraden Elemente zurückgeben (die Reihenfolge ist egal).

2. Für volle Punktzahl definieren Sie die obigen Funktionen *nicht-rekursiv* mit Hilfe einer allgemeineren Funktion `foldTree`. Diese rekursive Funktion soll unter anderem einen Parameter vom Typ `a -> b -> b -> b` haben und diesen auch verwenden. Lassen Sie sich von den `fold`-Funktionen auf Listen inspirieren.

Aufgabe H9.4 Die Quasi-Mehrheit (Wettbewerbsaufgabe, 5 Punkte)

> *Imagine a convention center filled with delegates (i.e., voters) each carrying a placard proclaiming the name of his candidate. Suppose a floor fight ensues and delegates of different persuasions begin to knock one another down with their placards. Suppose that each delegate who knocks down a member of the opposition is simultaneously knocked down by his opponent. Clearly, should any candidate field more delegates than all the others combined, that candidate would win the floor fight and, when the chaos subsided, the only delegates left standing would be from the majority block.*
>
> — Robert S. Boyer and J Strother Moore (1981)

Ein Wert x ist ein *Mehrheitselement* einer Liste, wenn über die Hälfte der Listeneinträge gleich x sind. Mehrheitselemente sind in der Politik unter der Bezeichnung „50 %+1“ bzw. „50 %+X“ bekannt. Zum Beispiel ist `CSU` ein Mehrheitselement für [`CSU`, `SPD`, `CSU`, `SPD`, `CSU`, `CSU`, `CSU`, `FDP`, `CSU`], da $6/9 > 1/2$. Eine Liste kann maximal ein Mehrheitselement enthalten.

Der Master of Competition ist immer auf der Suche nach neuen Begriffen. Sonst könnten die Wettbewerber allzu einfach auf die Schultern von Riesen klettern (das heißt, Lösungen im Internet finden). Die Aufgabe der siebten Woche hat den Ausdruck „quasi-identisch“ eingeführt. Diesmal sind *Quasi-Mehrheitselemente* an der Reihe. Das sind Elemente x, für die die Hälfte oder mehr der Listeneinträge gleich x sind. Bei Quasi-Mehrheitselementen reichen 50 % aus. Listen wie [`Yea`, `Nay`] und [`Yea`, `Yea`, `Nay`, `Yea`, `Nay`, `Nay`] enthalten zwei Quasi-Mehrheitselemente.

Gesucht wird eine Funktion

```
quasiMajorityElems :: Eq a => [a] -> [a]
```

die eine Liste als Argument nimmt und eine Liste Quasi-Mehrheitselemente zurückgibt. Die Ausgabenliste darf keine Duplikate enthalten. Bei mehreren Quasi-Mehrheitselementen ist die Reihenfolge beliebig.

Beispiele:

```
                  quasiMajorityElems []  == []
                 quasiMajorityElems [0]  == [0]
              quasiMajorityElems [0, 0]  == [0]
       sort (quasiMajorityElems [0, 1]) == [0, 1]
          quasiMajorityElems [0, 0, 1]  == [0]
          quasiMajorityElems [0, 1, 0]  == [0]
          quasiMajorityElems [0, 0, 1]  == [0]
sort (quasiMajorityElems [0, 0, 1, 1]) == [0, 1]
   quasiMajorityElems [0, 0, 1, 1, 2]  == []
      quasiMajorityElems [0, 1, 2, 2]  == [2]
```

Für den Wettbewerb zählt die Effizienz, vor allem die Skalierbarkeit. Ihre Implementierung sollte mit langen Listen möglichst effizient vorgehen. Lösungen, die sich effizienztechnisch ähnlich verhalten, werden bezüglich ihrer Lesbarkeit verglichen, wobei Formatierung und Namensgebung besonders gewichtet werden.

Die Lösung muss innerhalb der Kommentare {`-WETT-`} und {`-TTEW-`} abgegeben werden, um als Wettbewerbsbeitrag zu zählen.

Nur Lösungen, die alle Tests in der Testreihe des MCs bestanden haben, können Wettbewerbspunkte erzielen. Die Richtigkeit ist diese Woche *besonders wichtig*, da der MC möchte, mit der schnellsten Lösung deutschlandweit die Stimmzettel hunderttausender Parteimitglieder auszählen. Programmierfehler hätten ungeheure Konzequenzen für Deutschland, Europa und die Welt.

Wichtig: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können

diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken. Wenn Sie nicht am Wettbewerb teilnehmen, sondern die Aufgabe allein im Rahmen der Hausaufgabe abgeben möchten, lassen Sie bitte die {`-WETT-`} ... {`-TTEW-`} Kommentare weg. Bei der Bewertung Ihrer Hausaufgabe entsteht Ihnen hierdurch kein Nachteil.