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

WS 2012/13
04.11.2012
Abgabe: 11.12.2012, 15:30

Einführung in die Informatik 2
8. Übung

Aufgabe G8.1 Brüche sind auch Zahlen

In einer früheren Aufgabe (G2.3) haben wir Tupel genutzt, um Brüche darzustellen. Nun werden wir dies mit einem eigenen Datentyp erledigen. Definieren Sie dazu einen Typ `Fraction` mit einem Konstruktor `Over :: Integer -> Integer -> Fraction`.

1. Damit sich die Brüche nicht benachteiligt fühlen, sollen die üblichen mathematischen Operatoren auch mit Brüchen funktionieren. Definieren Sie daher `Fraction` als eine Instanz der Typklasse `Num`.

```
class  Num a  where
    (+), (-), (*)       :: a -> a -> a
    negate              :: a -> a
    abs                 :: a -> a
    signum              :: a -> a
    -- The functions 'abs' and 'signum' should
    -- satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- Conversion from Integer to a
    fromInteger         :: Integer -> a
```

2. Bei der Definition eines Datentypes kann Haskell diesen automatisch zu einer Instanz von `Eq` machen (mittels `deriving Eq`). Ist diese automatisch abgeleitete Instanz hier sinnvoll? Wenn nein, wie sollte sie stattdessen aussehen?

Weiterführende Hinweise:

1. Die Typklasse `Num` enthält keinen Divisionsoperator, dieser ist in der Typklasse `Fractional` implementiert. In einem echten Programm würde man `Fraction` daher zu darüber hinaus zu einer Instanz von `Fractional` machen.
2. Hinweis: Die Funktion `fromInteger` wird von Haskell benutzt, um ein Integer-Literal in den gesuchten Typ umzuwandeln. Der Ausdruck `3 :: Fraction` ist also der Bruch `fromInteger 3 :: Fraction`. Die Funktion `fromRational` aus der Typklasse `Fractional` macht das gleiche für Dezimalliterale (z.B. `3.14`).
3. Eine ähnliche Konstruktion wie in dieser Aufgabe ist in Haskell mit den Typen `Ratio` und `Rational` bereits implementiert.

Aufgabe G8.2 (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. Für `isSimple` (bzw. `simplify`) ist die Negation vor einer (Bi-)Implikation verboten (bzw. soll "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 G8.3 Arithmetische Ausdrücke (optional)

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

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

Aufgabe G8.4 Die Stunde der Wahrheit(stabelle) (optional)

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 H8.1 Äquivalenz von booleschen Ausdrücken (4 Punkte)

Schreiben Sie eine Funktion `equivalent :: Form -> Form -> Bool`, die genau dann `True` zurückliefert, wenn die beiden übergebenen Formeln äquivalent sind. Zwei Formeln sind genau dann äquivalent, wenn sie für alle Wertebelegungen zum gleichen Wahrheitswert ausgewertet werden.

Aufgabe H8.2 Einfache Wahrheiten (8 Punkte)

Es kommt häufig vor, dass man bereits beim Aufbauen einer Datenstruktur triviale Vereinfachungen vornehmen möchte. Bei booleschen Ausdrücken z.B. bietet es sich an, Verknüpfungen mit T und F sofort auszuwerten:

$$\sim\mathsf{T} \equiv \mathsf{F} \qquad \sim\mathsf{F} \equiv \mathsf{T}$$

$$\mathsf{T} \wedge \varphi \equiv \varphi \qquad \varphi \wedge \mathsf{T} \equiv \varphi \qquad \mathsf{F} \wedge \varphi \equiv \mathsf{F} \qquad \varphi \wedge \mathsf{F} \equiv \mathsf{F}$$

$$\mathsf{T} \vee \varphi \equiv \mathsf{T} \qquad \varphi \vee \mathsf{T} \equiv \mathsf{T} \qquad \mathsf{F} \vee \varphi \equiv \varphi \qquad \varphi \vee \mathsf{F} \equiv \varphi$$

1. Schreiben Sie die folgenden Funktionen:

   ```
   sNot :: Form -> Form
   sAnd :: Form -> Form -> Form
   sOr :: Form -> Form -> Form
   ```

   Jede dieser Funktionen soll die entsprechende logische Verknüpfung herstellen, gegebenenfalls vereinfacht durch die einmalige Anwendung einer der obigen Regeln. Eine rekursive Anwendung der Regeln soll *nicht* erfolgen.

   Die folgenden Beispiele werten zu `True` aus:

   ```
   sNot T == F
   sNot (Not T) == Not (Not T)
   sAnd (Not T) T == Not T
   sAnd (Var "x") (Var "z") == Var "x" :&: Var "z"
   sOr F (Var "x" :|: Var "y") == Var "x" :|: Var "y"
   sOr F (T :|: Var "y") == T :|: Var "y"
   ```

2. Schreiben Sie eine Funktion `sForm :: Form -> Form`, die die obigen Regeln verwendet, um alle T und F aus einer Formel zu entfernen (oder diese gegebenfalls zu F oder T zu vereinfachen). Verwenden Sie die Funktionen aus der vorherigen Teilaufgabe.

   Die folgenden Beispiele werten zu `True` aus:

```
sForm T == T
sForm (Not T) == F
sForm (Not T :&: T) == F
sForm (Var "x" :&: Var "z") == Var "x" :&: Var "z"
sForm (F :|: (T :|: Var "y")) == T
sForm (F :|: (Var "x" :|: Var "y")) == Var "x" :|: Var "y"
```

Aufgabe H8.3 Konjunktive Normalform (8 Punkte)

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 $\sim a$ (d.h. `Not` a).
- Eine *Klausel* ist eine n-stellige, nach links geklammerte Disjunktion $((\cdots((l_1 \vee l_2) \vee l_3) \vee \cdots) \vee l_{n-1}) \vee l_n$ von Literalen l_j (für $n \geq 1$).
- Eine *Formel in konjunktiver Normalform* (*KNF*) ist eine n-stellige, nach links geklammerte Konjunktion $((\cdots((c_1 \wedge c_2) \wedge c_3) \wedge \cdots) \wedge c_{n-1}) \wedge c_n$ von Klauseln c_j (für $n \geq 1$).

Beispiele:

- $\sim$ F ist ein Literal, eine Klausel und eine KNF-Formel.
- $\sim \sim$ F ist kein Literal (wegen der doppelten Negation).
- F $\vee$ T ist eine Klausel und eine KNF-Formel.
- F $\wedge$ T ist eine KNF-Formel.
- x $\wedge$ (y $\vee \sim$ z) ist eine KNF-Formel.
- x $\wedge$ ((y $\vee \sim$ z) $\vee$ w) ist eine KNF-Formel.
- x $\wedge$ (y $\vee$ ($\sim$ z $\vee$ w)) ist keine KNF-Formel (wegen der nicht kanonischen Klammerung).

Aufgaben:

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

   Hinweis: Schreiben Sie Hilfsfunktionen `isAtom`, `isLiteral` und `isClause` und testen Sie sie gründlich, bevor Sie sie in `isCnf` 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:

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

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

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

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 `toCnf :: Form -> Form`, die eine beliebige Formel in eine äquivalente KNF-Formel übersetzt.

   Hinweis: Die einfachste Lösung verwendet `pushNot`, `pushOr` und `balance` sowie die in Aufgabe H5.1 eingeführte Funktion `fixpoint`. Die Grundidee besteht darin, `pushNot`, `pushOr` und `balance` wiederholt anzuwenden, bis die Formel sich nicht mehr ändert.

6. Geben Sie QuickCheck-Tests für `toCnf` 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 W9.1 Ein boolescher Löser (zweiwöchige Wettbewerbsaufgabe, **0 Punkte**)

Diese Wettbewerbsaufgabe ist am 18.12.2012 als Teil von Blatt 9 abzugeben. Sie bringt dem Löser bis zu 40 Wettbewerbspunkte, dafür aber *keine* Hausaufgabenpunkte.

In der Vorlesung führte der Meta-Master die Funktion `satisfiable` ein (auch in `Form.hs` zu finden), die überprüft, ob eine boolesche Formel erfüllbar ist. Leider hat diese Funktion zwei Nachteile: Sie ist ineffizient und gibt keine erfüllende Wertebelegung zurück, falls diese existiert. In dieser Wettbewerbsaufgabe wollen wir – d.h. die (Co-)Masters of Competition – diese Probleme zumindest ansatzweise beheben.

Gesucht ist eine Funktion `satisfyCnf :: [[Int]] -> Maybe [Int]`, die eine KNF-Formel φ als Argument bekommt und eine erfüllende Wertebelegung zurückgibt, falls φ erfüllbar ist und `Nothing`, falls φ nicht erfüllbar ist.

Hier wird eine kompakte Darstellung von KNF-Formeln als Listen von Listen benutzt, wobei die äußerste Liste eine mehrstellige Konjunktion und die inneren Listen mehrstellige Disjunktionen darstellen. Die Variablen werden als positive Zahlen kodiert. Die Negation einer Variable ist die ensprechende negative Zahl. Auf diese Weise kann man $\mathsf{x}_1 \wedge ((\mathsf{x}_2 \vee \sim \mathsf{x}_3) \vee \mathsf{x}_4)$ als $[[1], [2, -3, 4]]$ darstellen. Die Zahl 0 ist kein gültiges Atom.

Die ausgegebene Wertebelegung enthält die positive Zahl n (bzw. die negative Zahl $-n$), wenn die Variable n für die Lösung wahr (bzw. falsch) sein muss. Dabei müssen nicht notwendigerweise alle in der Formel vorkommenden Variablen einen Wert erhalten, um die Erfüllbarkeit der Formel zu garantieren. Zum Beispiel sind alle folgenden Optionen erlaubt:

```
satisfyCnf [[1, 2]] == Just [1]
satisfyCnf [[1, 2]] == Just [2]
satisfyCnf [[1, 2]] == Just [1, 2]
satisfyCnf [[1, 2]] == Just [2, -1]
```

Dies ist aber nicht korrekt:

```
satisfyCnf [[1, 2]] == Just [-1]
```

Gegenbeispiel: Die angebliche Lösung erzwingt $\mathsf{x}_1 := \mathsf{F}$, sagt aber nichts über die andere Variable. Für $\mathsf{x}_2 := \mathsf{F}$ ist $\mathsf{x}_1 \vee \mathsf{x}_2$ falsch. ↯

Der effizienteste bekannte Algorithmus verhält sich im schlechtesten Fall genauso wie der naive Algorithmus von Prof. Nipkow. In der Praxis können wir jedoch wichtige Fälle optimieren, indem wir Heuristiken einsetzen und bestimmte Klassen von Eingaben gesondert behandeln.

Einige Anregungen zu den Heuristiken:

1. Für Variablen, die ausschließlich positiv vorkommen, gibt es im Allgemeinen nur eine sinnvolle Belegung, nämlich `True`.
2. Typische Algorithmen wählen wiederholt eine beliebige Klausel und anschließend eine beliebige Variable aus dieser Klausel und bearbeiten sie. Es kann sinnvoll sein, kürzere Klauseln den Längeren vorzuziehen.
3. Bei der Wahl der Variablen kann es ebenfalls sinnvoll sein, häufig vorkommende Variablen zu bevorzugen.

Zwei bekannte Eingabenklassen:

4. Eine *Horn*-Klausel ist eine Klausel, die höchstens ein positives Atom enthält. Falls alle Klauseln einer Eingabe Horn-Klauseln sind, kann das Problem effizient mit einer Prozedur ähnlich dem Vorgehen eines Prolog-Interpreters gelöst werden.
5. Eine *2KNF*-Klausel ist eine Klausel, die maximal zwei Literale enthält. Falls alle Klauseln einer Eingabe 2KNF-Klauseln sind, gibt es auch hierfür eine bekannte effiziente Lösung.

Bei der Punktevergabe sind die oben genannten Optimierungen entscheidend, wobei Punkt n unendlich wichtiger als Punkt $n+1$ ist (für $n \in \{1, 2, 3, 4\}$). Bei Gleichstand werden Probleme, die mit Variationen von den obigen Optimierungen sowie mit anderen bekannten Optimierungen effizient lösbar sind, eingesetzt.

Bei der Jagd nach Effizienz sollten Sie die Korrektheit Ihrer Funktion nicht aus den Augen lassen. Lösungen, die die Korrektheitstests nicht bestehen, bekommen keine Punkte. Sie können die Korrektheit selbst testen, indem Sie auf die im Internet verfügbaren Datenbanken mit sogenannten SAT-Problemen zugreifen.

Falls Sie eine Lösung finden, die in polynomieller Zeit in der Größe der Eingabe die Antwort liefert, wird Ihnen Prof. Nipkow Ihren Doktortitel sicher mit Freuden überreichen.