Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
L. Noschinski, L. Hupel, Dr. J. Blanchette, M. Eberl

WS 2014/15
05.12.2014
Abgabe: 12.12.2014

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.

Aufgabe G9.2 Irgendwas mit HTML

Wir betrachten eine vereinfachte Variante von HTML, die nur aus (verschachtelten) Elementen und Text besteht:

```
<body>
    <h1>Lorem ipsum</h1>

    <p>Lorem ipsum dolor sit amet, consetetur <em>sadipscing</em>
    elitr, sed diam nonumy eirmod tempor invidunt ut labore
    et dolore magna aliquyam erat, sed diam voluptua.
    </p>

    <p>At vero eos et accusam et justo duo dolores et ea rebum.
    Stet clita kasd gubergren, no sea takimata sanctus est
    Lorem ipsum dolor sit amet.
    </p>
</body>
```

1. Definieren Sie einen geeigneten Datentyp `Html` zur Darstellung von HTML-Dokumenten.
2. Schreiben Sie eine Funktion `htmlShow :: Html -> String`, die ein beliebiges Html-Dokument in einen String umwandelt.

3. In der String-Darstellung von HTML-Dokumenten werden die Zeichen `<` und `>` verwendet, um die Struktur zu markieren. Ein Text in einem HTML-Dokument darf diese Zeichen also nicht enthalten. Um das zu umgehen, soll in Texten `<` durch `<`, `>` durch `>` und `&` durch `&` dargestellt werden. Passen Sie Ihre Funktion `htmlShow` entsprechend an.
4. Registrieren Sie eine `Show`-Instanz für Ihren Datentyp.

Hinweis für den Wettbewerb: Am 12.12.2014 ist Abgabeschluss für die zweiwöchige Wettbewerbsaufgabe, die auf dem vorherigen Blatt gestellt worden ist. Nutzen Sie für Ihre endgültige Abgabe bitte die speziell auf der Übungsseite bereitgestellte Schablone und laden diese im Übungssystem unter *Wettbewerb 8/9* hoch. **Wettbewerbsbeiträge, die als Teil der Hausaufgaben auf diesem Blatt hochgeladen werden, werden nicht berücksichtigt.**

Aufgabe H9.1 Bäume beweisen

Auf Bäumen kann – genau wie auf Listen – eine `map`-Funktion definiert werden.

```
mapTree f Empty = Empty
mapTree f (Node x t1 t2) =
    Node (f x) (mapTree f t1) (mapTree f t2)
```

Zeigen Sie die Gleichung `mapTree (f . g) x .=. mapTree f (mapTree g x)`.

Aufgabe H9.2 Arithmetische Ausdrücke I

In den Gruppenaufgaben haben Sie mit booleschen Formeln gearbeitet. In dieser (und der nächsten) Aufgabe wollen wir stattdessen mit arithmetischen Formeln arbeiten. Zunächst definieren wir einen arithmetischen Ausdruck wie folgt:

```
data Arith = Literal Integer
           | Var String
           | Arith :+: Arith
           | Arith :*: Arith
```

Die Semantik dieses Typs wird durch die Namen der Konstruktoren angedeutet: `Literal` steht für eine konstante Zahl, `Var` für eine benannte Variable, `:+:` für die Summe zweier Ausdrücke und `:*:` für das Produkt.

1. Schreiben Sie eine Auswertungsfunktion für diesen Datentyp. Im Gegensatz zu booleschen Formeln, wo die Belegung von Variablen als Liste `[(String, Bool)]` übergeben wird, erhält Ihre Funktion eine Belegungsfunktion vom Typ `String -> Integer`.
2. Zwei Ausdrücke sind äquivalent, wenn Sie für alle Belegungsfunktionen zum gleichen Resultat ausgewertet werden. Geben Sie eine Funktion `equivalent` an, die für eine gegebene Belegungsfunktion prüft, ob zwei Formeln zum gleichen Resultat auswerten.

Nun wollen wir Ausdrücke vereinfachen. Eine *Vereinfachungsfunktion* ist eine Funktion vom Typ `Arith -> Arith`, die einen zur Eingabe äquivalenten Ausdruck zurückgibt, in dem gewisse Konstrukte nicht mehr vorkommen.

3. Für Addition und Multiplikation auf ganzen Zahlen gelten folgende Gesetze:

$$\begin{array}{lll} x + 0 = x & 1 * y = y & 0 * y = 0 \\ 0 + x = x & y * 1 = y & y * 0 = 0 \end{array}$$

   Desweiteren kann die Addition und Multiplikation zweier Literale zu einem einzigen Literal ausgewertet werden *(constant folding).*

   Definieren Sie einen QuickCheck-Test, der für einen gegebenen Ausdruck prüft, dass dieser (mit den obigen Regeln) nicht weiter vereinfacht werden kann.

4. Definieren Sie einen weiteren QuickCheck-Test, der für eine gegebene Funktion prüft, ob es sich um eine Vereinfachungsfunktion (mit den Regeln aus Teilaufgabe 3) handelt.

   Hinweis: Neben der mutmaßlichen Vereinfachungsfunktion erhält ihr Test eine Belegungsfunktion und einen Ausdruck als Eingabe. Belegungsfunktion und Ausdruck werden vom Testsystem zufällig generiert. Wenden Sie die Vereinfachungsfunktion auf den Ausdruck an und nutzen Sie auch den QuickCheck-Test aus der vorherigen Teilaufgabe.

Aufgabe H9.3 Arithmetische Ausdrücke II

Implementieren Sie nach den Vorgaben der vorherigen Aufgabe eine Vereinfachungsfunktion für arithmetische Ausdrücke: `simplify :: Arith -> Arith`.

Hinweis: Beide Aufgaben werden unabhängig voneinander bewertet, Sie dürfen aber trotzdem Code wiederverwenden.

Aufgabe H9.4 Implementation gesucht

Wir betrachten in dieser Aufgabe den Datentypen `Either` aus der Haskell-Standardbibliothek mit der Definition:

```
data Either a b = Left a | Right b
```

Dieser Datentyp ist immer dann nützlich, wenn man einen Wert hat, der entweder Typ `a` oder Typ `b` haben kann. Man kann so z.B. fehlerhafte Eingaben behandeln, etwa in einer Funktion zum Parsen von Integern: Die Funktion hat den Typ `String -> Either String Integer`, d.h. sie parst einen String und gibt den resultierenden Integer mit `Right` zurück; im Fehlerfall (wenn der String keinen Integer repräsentiert) gibt sie mit `Left` eine Fehlermeldung zurück.

Nun zur eigentlichen Aufgabe: In manchen Fällen ist es interessant, für einen gegebenen Typ eine Funktion zu finden, die diesen Typ hat. Für den Typ `Either a b -> Either b a` gibt es etwa die folgende Implementation:

```
f :: Either a b -> Either b a
f (Left  x) = Right x
f (Right x) = Left  x
```

Ihre Aufgabe ist es, totale Funktionen zu schreiben, die die folgenden Typen haben:

```
g :: (a -> a') -> (b -> b') -> Either a b -> Either a' b'
h :: (a -> Either a b) -> a -> b
```

Total bedeutet hierbei, dass die Funktionen keine Exceptions werfen dürfen (z.B. mit `undefined`) und (soweit möglich) terminieren müssen. Die Implementation `g x = g x` wäre also ungültig.

Hinweis: Es gibt für beide Typen jeweils nur *eine* korrekte Funktion dieses Typs.