Funktionale Programmierung

`http://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2013/`

Übungsblatt 11 (GADTs)

Fr, 2014-01-31

Hinweise

- Lösungen sollen in das persönliche Subversion (svn) Repository hochgeladen werden. Die Adresse des Repositories wird per Email mitgeteilt.
- **Alle** Aufgaben müssen bearbeitet und pünktlich abgegeben werden. Falls das sinnvolle Bearbeiten einer Aufgaben nicht möglich ist, kann eine stattdessen eine Begründung abgegeben werden.
- Wenn die Abgabe korrigiert ist, wird das Feedback in das Repository hochgeladen. Die Feedback-Dateinamen haben die Form `Feedback-<user>-ex<XX>.txt`.
- Allgemeinen Fragen zum Übungsblatt können im Forum (`http://proglang.informatik.uni-freiburg.de/forum/viewforum.php?f=38`) geklärt werden.

Abgabe: Fr, 2014-02-08

Hinweis: In dieser Aufgabe müssen Sie die GHC-Erweiterung `GADTs` einschalten. Fügen Sie dazu das folgende Spezial-Kommentar (Pragma) am Anfang Ihrer Haskell-Quelldatei ein:

```
{-# LANGUAGE GADTs #-}
```

Außerdem empfiehlt es sich weiterhin auch das Pragma

```
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
```

am Anfang der Datei einzufügen. Damit wird Ihnen beim Laden der Datei angezeigt, ob sie in den Fallunterscheidungen bei Funktionsdefinitionen alle möglichen Patterns bedacht haben.

Aufgabe 1 (SafeList)

Definieren Sie einen Listentyp `SafeList`, der eine „sichere" head-Operation `safeHead` unterstützt. Das heißt, der Typchecker sollte die Anwendung von `safeHead` nur dann erlauben, wenn das Argument eine nicht-leere `SafeList` ist:

```
safeHead (Cons 4 Nil) -- ok
-- safeHead Nil       -- Typfehler
```

(Die Operation wird als „sicher" bezeichnet, da sie, im Gegensatz zu head, keinen Laufzeitfehler bei falschen Eingaben verursacht.)

Implementieren Sie außerdem auf sinnvolle Weise `safeDrop` und `safeAppend`.

Aufgabe 2 (Getypter Stack-Rechner)

In Ex02, Aufgabe 2 haben wir einen Stackrechner implementiert. Dieser war recht einfach gestrickt:

- er unterstützte nur arithmetische Operationen,
- und lieferte bei „Unterlauf" immer 0 zurück

Inzwischen können wir das besser!

Der Stack soll nun eine endliche Tiefe haben und sowohl Int- als auch Bool-Werte enthalten können. Die Stack-Programme sollen aus den folgenden Befehle bestehen:

```
sprog ::=
      | noop             -- tut nichts
      | pop              -- entfernt das oberste Element vom Stack
      | push v           -- legt den Wert v auf den Stack
      | dup              -- legt ein Duplikat des obersten Elements auf den Stack
      | dup2             -- legt Duplikate der zwei obersten Eemente auf den Stack
      | flip             -- vertauscht die beiden obersten Elemente
      | add | substract | multiply | negate
                         -- führt eine arith. Operation
                         -- auf den obersten Elementen aus
                         -- und legt das Ergebnis auf den Stack
      | le | ge          -- vergleicht die beiden obersten Elemente
                         -- und legt das Ergebnis auf den Stack
      | not | and | or   -- führt eine logische Operation aus
      | sprog1 ; sprog2    -- führt erst sprog1 aus, dann sprog2
      | if sprog1 sprog2   -- führt sprog1 aus, falls True auf dem Stack liegt,
                         -- ansonsten sprog2
```

1. Definieren Sie den Datentyp `SProg` so, dass er nur Programme enthält, die fehlerfrei ausgeführt werden können.

2. Implementieren Sie einen Tag-freien Interpreter für `SProg`.

3. Schreiben Sie ein `SProg`-Programm, das den Betrag des obersten Stack-Elements berechnet. Schreiben Sie zwei weitere `SProg`-Bespielprogramme und testen Sie diese.

4. Fügen Sie nun das Schleifenkonstrukt `while` zu `SProg` hinzu.

   ```
   sprog ::= ... | while sprog
   -- führt sprog solange aus, bis danach nicht
   -- mehr True auf dem Stack liegt
   ```

5. Schreiben Sie ein `SProg`-Program, welches gegeben x, y das Ergebnis $x \bmod y$ berechnet (falls $y \leq 0$ darf sich das Program beliebig verhalten). Benutzen Sie dafür nur die Operationen aus der obigen Syntax (und `while`, natürlich)

6. Geben Sie ein Beispiel eines prinzipiell fehlerfreien Stack-Programms, das Sie *nicht* mit ihrem `SProg` Datentyp ausdrücken können.

Aufgabe 3 (Roboterkommandos (optional))
Das Modul `Robot.hs` (auf der Homepage) definiert Aktionen für einen (eindimensionalen) Bergbau- und Erkundungsroboter. Er unterstützt folgende Aktionen:

`mine :: Robot Gold`
Betreibe Bergbau und sammle das gefundene Gold ein.

`scan :: Robot Result`
Untersuche die Gegend und speichere das Ergebnis.

`go n c :: Int -> Robot a -> Robot a`
Gehe `n` Schritte weiter und führe Kommando `c` aus.

`c1 >+> c2 :: Robot Gold -> Robot Gold -> Robot Gold`
Führe die Bergbauoperation `c1` aus, danach die Bergbauoperation `c2`. Sammle alles Gold auf

`c1 <-> c2 :: Robot a -> Robot b -> Robot b`
Führe erst `c1` aus, kehre zum Startpunkt zurück um die Ergebnisse/das Gold abzuliefern. Führe danach `c2` aus.

Zwei Beispielkommandos (auch in `Robot.hs` enthalten):

```
robprog     = (go 7 mine) <-> (go 5 mine) <->  (go 6 scan)
robprog'    = (go 7 (mine >+> go (-2) mine)) <-> (go 6 scan)
somemining  = (go 7 (mine >+> go (-2) mine))
```

Testen Sie die Programme mit `runRobot`, um zu sehen was sie tun. Das Typargument von `Robot` klassifiziert die letzte Aktion des Roboters (`Result` bezeichnet eine Scan-Aktion, `Gold` eine Bergbau-Aktion).

Das Programm `robprog'` ist eine effizientere Version von `robprog`, da alles Gold in einem Durchgang eingesammelt wird. Leider hängt diese Verbesserung stark von der „Klammerung" der einzelnen Befehle ab: Der `(>+>)` Operator funktioniert nur auf `Robot Gold` Aktionen; folgendes Programm kann z.B. nicht getypt werden:

```
combine_error = somemining >+> robprog'
```

und muss durch eine langsamere Version ersetzt werden:

```
combine_slow = somemining <-> robprog'
```

Ihre Aufgabe ist es, einen GADT (z.B. `RobotG a`) zu definieren, der Roboterprogramme wiederspiegelt, aber eine Operation `<+>` unterstützt, die aufeinanderfolgende Bergbauoperationen unabhängig von der Kommandoklammerung zusammenfasst.

Implementieren Sie also Alternativen zu den gegebenen Roboter-Kommandos:

```
mine'  :: RobotG Gold
scan'  :: RobotG Result
go'    :: Int -> RobotG a -> RobotG a
(<+>)  :: RobotG a -> RobotG b -> RobotG b
```

und einen Interpreter, `runRobotG :: RobotG b -> Robot b`, der die modifizierten Befehle wieder in echte Roboter-Kommandos übersetzt. Vergewissern Sie sich, dass z.B. die übersetzten `RobotG`-Programme wie

```
p0 = runRobotG $ mine' <+> (mine' <+> scan')
p1 = runRobotG $ (go' 7 mine') <+> (go' 5 mine') <+>  (go' 6 scan')
```

sich äquivalent verhalten zu:

```
p0' = (mine >+> mine) <-> scan
p1' = (go 7 mine >+> (go (-2) mine)) <->  (go 6 scan)
```

Hinweise

- Die Kommandosprache ist sehr einfach strukturiert, und die Klammerung dadurch leicht modifizierbar, sofern man noch Zugriff auf die einzelnen Teile der Programme hat.
- Versuchen Sie zunächst, Programme ohne `go` abzubilden, z.B. `p0`. Erweitern Sie danach ihre Lösung um mit `go` umzugehen.