Christian-Albrechts-Universität zu Kiel
AG Programmiersprachen und Übersetzerkonstruktion
Prof. Dr. M. Hanus

Christian-Albrechts-Platz 4
24118 Kiel
Tel.: 0431 / 880-7271

7. Übung zur Vorlesung „Prinzipien von Programmiersprachen“

Wintersemester 2008/2009

Abgabe: 13. Januar 2009 in der Vorlesung

Aufgabe 23 (Präsenzaufgabe)

In dieser Aufgabe geht es um unterschiedliche Auswertungsstrategien.

a) Wir betrachten das folgende funktionale Programm:

```
double x = x+x
```

Geben Sie alle möglichen Auswertungen des Ausdrucks `double (double 3)` an. Welches sind die leftmost-innermost (LI) und die leftmost-outermost (LO) Auswertungen?
Skizieren Sie die lazy Auswertung von `double (double 3)` (nicht die Launchbury-Semantik!).

b) In der Vorlesung wurde die Semantik von `if-then-else` definiert. Zeigen Sie, dass es auch in einer strikten Sprache, wie ML oder Scheme wichtig ist, dass dieses Konstrukt nicht strikt in seinem zweiten und dritten Argument ist. Geben Sie hierzu die Semantik der in der Vorlesung definierten `fac`-Funktion für den Ausdruck `fac 1` an.

Aufgabe 24

Implementieren Sie in Haskell folgende Funktionen:

a) `rev :: [Int] -> [Int]` zum Umdrehen einer Liste von Zahlen.
`rev [1,2,3]` ergibt z.B. `[3,2,1]`.
Welche Komplexität hat ihre Implementierung (ggf. experimentelle Komplexitätsermittlung)? Können Sie eine lineare Implementierung angeben?

b) `insert :: Int -> [Int] -> [[Int]]`
zum Einfügen einer Zahl an allen möglichen Positionen einer Liste von Zahlen.
`insert 3 [1,2]` ergibt z.B. `[[3,1,2],[1,3,2],[1,2,3]]`
Geben Sie eine zweite Version dieser Funktion an, welche kein Pattern-Matching (auch nicht im `case`) verwendet.

c) `perm :: [Int] -> [[Int]]`
zur Berechnung aller möglichen Permutationen einer Liste von Zahlen.
`perm [1,2,3]` ergibt z.B. `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`.
Ihre Funktion kann die Permutationen in der Ergebnisliste auch in einer anderen Reihenfolge ausgeben. Wichtig ist nur, dass alle Permutationen enthalten sind.

Aufgabe 25
Definieren Sie einen Datentyp `Exp` zur Repräsentation arithmetischer Ausdrücke in Haskell. Neben den zweistelligen Konstruktoren `Add`, `Sub`, `Mult` und `Div` soll `Exp` einen einstelligen Konstruktor `Num` zur Repräsentation von `Int`-Zahlen enthalten.
Implementieren Sie eine Funktion `eval :: Exp -> Int` zur kanonischen Auswertung eines Ausdrucks.
Machen Sie Ihr Programm robust gegen den "Division-durch-0-Fehler". Definieren Sie hierzu einen Datentyp `OptInt`, welcher neben einem Ergebnis (vom Typ `Int`) auch den Fehlerwert repräsentiert. Unter Verwendung dieses Fehlertyps ändert sich der Typ der Auswertungsfunktion wie folgt: `eval :: Exp -> OptInt`. Passen Sie die Implementierung entsprechend an.

Aufgabe 26
In der Vorlesung wurde die operationale Semantik von lazy funktionalen Sprachen mit Hilfe der Launchbury-Semantik formalisiert.

a) Berechnen Sie die lazy Auswertung des folgenden Programms, d.h. geben Sie die in jedem Schritt entstehenden Term- bzw. Graphstrukturen an:

```
main = let ones = 1:ones in (head ones) + (head (tail ones))

head (x:xs) = x
tail (x:xs) = xs
```

b) Was liefert die Launchbury-Semantik für

```
main = let ones = 1:ones in (head ones, head (tail ones))
```

c) Wie verändert sich die Auswertung, wenn `ones` als Funktion definiert wird:

```
main = let o = ones in (head o) + (head (tail o))

ones = 1:ones
```

Hinweis: Beachten Sie, dass die Programme nicht normalisiert sind! Ersetzen Sie zunächst das Pattern-Matching durch `case`-Ausdrücke und definieren dann in einem `let`-Ausdruck Variablen für alle nicht-variablen Argumente.

Frohe Weihnachten und einen guten Rutsch in das Jahr 2009!