Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz/Softwaretechnologie

Fachbereich Informatik (20)
Johann Wolfgang Goethe-Universität Frankfurt am Main

Einführung in das funktionale Programmieren

Wintersemester 1995/96

Übungsblatt Nr. 1

Auch Leinwand, Zeug und allerlei Gerät,
Das viel seitdem genützt: so, aus Leutseligkeit,
Da ihm bekannt, ich liebe meine Bücher,
Gab er mir Bänd aus meinem Büchersaal,
Mehr wert mir als ein Herzogtum.

Shakespear: Der Sturm

Aufgabe 1

In dieser Aufgabe sollen Sie zwei kleine Funktionen in CORE-FP schreiben und mit *kleinen* Testauswertungen die korrekte Funktionalität Ihrer Funktionen demonstrieren.

a) Schreiben Sie eine Funktion `next` mit einem Argument i. `next` liefert als Ergebnis eine Funktion, die ein weiteres Argument hat und dieses um den Wert von `i` erhöht. Sie dürfen hierbei die Natürlichen Zahlen, sowie die Boolschen Konstanten `true` und `false` als in CORE-FP vordefinierte Konstanten betrachten. Weiterhin sollen in CORE-FP numerische Vergleichsoperatoren `>`, `<`, `=`, sowie die numerischen Operatoren `+`, `-`, `*`, `/` zur Verfügung stehen.

b) Schreiben Sie eine Funktion `summe` mit drei Argumenten `s`, `e` und `i`. `summe` liefert die Summe von

```
summe = s + i + i + i ...,
```

so daß

```
summe < e + i.
```

Das Ergebnis von `summe` soll dabei maximal sein. Verwenden Sie hierzu die Funktion `next` aus Aufgabenteil a).

c) Versuchen Sie Ihre CORE-FP Programme im Scheme Interpreter zum Laufen zu bringen. Da das `case` in der Form von CORE-FP in Scheme nicht vorhanden ist, versuchen Sie folgende Definition:

```
(define (case c x y z)
  (if (equal? c x)
      y
      z))
```

Sie werden bemerken, daß das so definierte `case` nicht das gewünschte Ergebnis liefert. (Warum?) Banale Frage: Wie funktioniert's und warum, wenn man's anders macht? (Experimentieren ist erlaubt; ein `case` muß nicht direkt im Programmtext auftauchen!)

Aufgabe 2
In der Vorlesung wurde CORE-FP definiert. Wir wollen in dieser Aufgabe CORE-FP-Ausdrücke als binäre Bäume schreiben. Hierbei wird die Klammerung aus CORE-FP in die Baumstruktur aufgelöst. Ein CORE-FP-Ausdruck ist nun entweder ein Blatt, das mit einer Konstanten oder einer Variablen markiert ist, oder ein Knoten der mit `@` markiert ist und zwei Nachfolgerbäume hat. Ein `@`-Knoten bezeichnet die Anwendung des linken Nachfolgers auf den rechten Nachfolger als Argument.

a) Beschreiben Sie die normale Reihenfolge der Auswertung für als Bäume dargestellte CORE-FP-Ausdrücke algorithmisch. Benutzen Sie bei der Beschreibung einen Stack, um die Argumente für Funktionsanwendungen zu halten.

b) Reduzieren Sie die Bäume der folgenden Ausdrücke nach den Algorithmus unter a), wobei folgende Funktionsdefinitionen vorliegen:

```
hochvier x = (* (quadrat x))(quadrat x)
quadrat x = ((* x) x)

fromto x y = ((((case True) ((> x) y)) Nil)
                ((Cons x) ((fromto ((+ x) 1)) y)))
```

   – `(hochvier 3)`
   – `((fromto 0) 1)`

c) Können Sie die Länge der Ableitung in obigen Beispiel verringern, indem Sie die CORE-FP-Ausdrücke nicht als Bäume sondern als Graphen darstellen?

Aufgabe 3
Im Prinzip können Sie CORE-FP Programme direkt in Gofer interpretieren lassen. Das einzige nicht vordefinierte Merkmal ist die Funktion `case` aus CORE-FP.

a) Schreiben sie eine Funktion `casefp` in Gofer, die das Case-Konstrukt aus CORE-FP realisiert. Lassen Sie ein paar CORE-FP Programme in Gofer reduzieren. In wiefern schränkt Gofer die Menge der CORE-FP-Programme ein?

b) Auf was für Schwierigkeiten wären Sie gestoßen, wenn Sie Teil a) in Scheme hätten realisieren müssen. (im Prinzip sind das zwei Probleme)

Zusatzaufgabe
Was hat das obige Zitat mit funktionaler Programierung zu tun?