P. Thiemann, S. Wehr Wintersemester 2006/2007

Compilerbau

http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2006ws/

Übungsblatt 9

10.1.2006

Aufgabe 1 (Rekursive Funktionen)
Auf der Homepage steht ein Interpreter für eine kleine funktionale Sprache zur Verfügung. Der Interpreter basiert auf dem in der Vorlesung vorgestellten Interpreter mit Continuations; allerdings gibt es keine Exceptions und Referenzen. Zusätzlich liegt dem Interpreter ein Lexer und ein Parser bei. Folgendes Beispielprogramm berechnet das Produkt einer Liste von Zahlen.

```
let sum = fun f(l) =
            if null(l) then 1
            else head(l) * f(tail(l))
 in sum(cons(2, cons(10, cons(3, nil))))
```

Leider läuft obiges Beispiel nicht mit dem aktuellen Interpreter, da dieser keine rekursive Funktionen unterstützt. Erweitern sie den Interpreter um rekursive Funktionen.

Hinweis: Es ist nur eine sehr kleine Änderung in der Datei `evalcc.ml` nötig. Abzugeben ist lediglich die Datei `evalcc.ml`.

Aufgabe 2 (Continuations)
Folgende OCaml Funktion berechnet das Produkt einer Liste von Zahlen:

```
let rec product l =
    match l with
        []      -> 1
      | (x::xs) -> x * product xs
```

Leider ist die Funktion ineffizient wenn die Eingabeliste die Zahl `0` enthält. Sie führt der Ausdruck `product [0;1;2]` zu insgesamt vier Aufrufen von `product` und zwei Multiplikationen, obwohl gleich am Anfang feststeht, dass das Ergebnis `0` sein wird.

Schreiben sie die Funktion mit Hilfe von Continuations so um, dass sofort `0` zurückgegeben wird sobald eine `0` in der Eingabeliste auftaucht. In diesem Fall dürfen keine unnötigen rekursiven Aufrufe und *keinerlei* Multiplikationen ausgeführt werden. Speichern sie ihre Lösung in einer Datei namens `product.ml`.

Aufgabe 3 (`call/cc`)
In manchen Programmiersprachen wie z.B. Scheme gibt es einen Operator `call/cc`, der dem Programmierer Zugriff auf den Rest der Berechnung in Form einer Continuation bietet. Der `call/cc` Operator erwartet dazu eine Funktion als Argument und ruft diese mit einer Continuation auf, welche den Rest des Programms repräsentiert. Beispielsweise kann im Ausdruck `call/cc (fun aux(return) =` e`)` der Subausdruck e die Continuation `return` benutzen um

die Berechnung des Subausdrucks e abzubrechen und direkt in den Kontext zu springen, in dem `call/cc` aufgerufen wurde.

Der in Aufgabe 1 vorgestellte Interpreter unterstützt `call/cc`. Falls ihnen also die bisherige Erklärung von `call/cc` nicht genau genug war, können sie im Sourcecode des Interpreters die genau Definition von `call/cc` nachschlagen.

Implementieren sie die effektive Version der `product` Funktion aus Aufgabe 2 mit Hilfe von `call/cc`. Speichern sie ihre Lösung in einer Datei namens `product-cc.exp`.

Abgabe: 17.1.2007
Die Abgabe erfolgt bis zu Beginn der Übungsstunde. Einreichungen, die nicht den Abgabemodalitäten entsprechen, werden abgelehnt. Alle Aufgaben werden mit vier Punkten bewertet. Für Plagiate werden keine Punkte vergeben.

Abgabemodalitäten:

- Code muss per Email an die Adresse `wehr@informatik.uni-freiburg.de` geschickt werden. Zu diesem Blatt ist kein Papierabgabe nötig.
- Der Code muss in einem Archiv mit dem Namen `vorname_nachname.tar.gz` oder `vorname_nachname.zip` an die Email angehängt werden.
- Entpacken des Archivs muss ein einzelnes Verzeichnis mit dem Namen `vorname_nachname` liefern.
- Innerhalb dieses Verzeichnisses müssen sich folgende Dateien befinden:
  - `evalcc.ml` (für Aufgabe 1 und Aufgabe 3)
  - `product.ml` (für Aufgabe 2)
  - `product-cc.exp` (für Aufgabe 3)
- Alle Dateien müssen compilierbar sein; Teile einer Datei, welche nicht vom Compiler akzeptiert werden, müssen auskommentiert sein. Abgaben, die der Compiler nicht akzeptiert, werden nicht bewertet.
- Wenn eine Aufgabe von der Lösung ein gewisses Format verlangt (wie etwa einen festen Namen oder Typ für eine Funktion, eine vorgegebene Signatur für ein Modul etc.), so ist dieses Format zwingend einzuhalten.