SAARLAND UNIVERSITY
COMPUTER SCIENCE

Proseminar "13 Wochen - 13 Sprachen"

Prof. Dr. Andreas Zeller, Jun.-Prof. Dr. Sebasitan Hack
Kontakt: Martin Burger, Christoph Mallon

5. Aufgabenblatt - Scheme

Ausgabe: 26. Mai 2009 Abgabe: 2. Juni 2009 Revision: 3241

1 Continuations

Im Kontext der Programmierung versteht man unter "Continuation" eine abstrakte Representation der Steuerstruktur, die sozusagen den "Rest einer Berechnung" oder den "Rest des auszuführenden Codes" darstellt. Im Beispiel `(- 1 (+ 7 5))` ist `(- 1 [...])` die Continuation von `(+ 7 5)`.

"Current continuation" bezeichnet einen Schnappschuss des aktuellen Steuerflusses und des zugehörigen Zustandes.

Führen Sie folgende Scheme-Beispiele aus, um ein Verständnis von Continuations zu erlangen. Wir werden dieses Konzept im Proseminar diskutieren.

1.1 Continuations in Scheme: call-with-current-continuation

Mit Hilfe der Prozedur `call-with-current-continuation` *`proc`*, oft abkürzend `call/cc` genannt, können Sie Continuations in Scheme verwenden: Der Operator `call/cc` *`proc`* ruft sein Argument auf, welches eine Prozedur *`proc`* mit einem einzigen Parameter ist, und übergibt als Wert die "current continuation".

So gibt beispielsweise die Eingabe von

```
1 (+ 1 (call/cc
2        (lambda (k)
3          (+ 2 (k 3)))))
```

unter Guile[1] `4` zurück; die Anweisung `+ 2` wird sprichwörtlich über Bord geworfen.

Im Tutorial "Teach Yourself Scheme in Fixnum Days"[2] finden Sie eine Reihe von erklärenden Beispielen.

[1] Unter Guile können Sie den kurzen Namen z.B. im Interpreter nach Eingabe von `(define call/cc call-with-current-continuation)` verwenden. Siehe Abschnitt 4 für Informationen über Guile.
[2] `http://tinyurl.com/scheme-jumps`

1.2 Exit Continuations

Mit der Prozedur `for-each` *proc* $list_1$ $list_2$ ... können Sie eine gegebene Prozedur auf die Elemente der gegebenen Listen anwenden. So können Sie z.B. `for-each` verwenden, um einen Eintrag in einer Liste zu suchen[3]:

```
;;; Gibt das erste Element in der Liste LST zurück,
;;; für das WANTED? true zurück liefert.
;;; Hinweis: Dies ist kein funktionierender
;;; Scheme-Code.
(define (search wanted? lst)
  (for-each (lambda (element)
              (if (wanted? element)
                  (return element)))
            lst)
  #f)
```

Listing 1: Nicht funktionierende Suche in einer Liste.

Leider können Sie obige Prozedur unter Scheme so nicht verwenden. Finden sie heraus, warum!

Folgendes Beispiel nutzt `call/cc`, um das gefundene Element zurückzuliefern:

```
;;; Gibt das erste Element in der Liste LST zurück,
;;; für das WANTED? true zurück liefert.
(define (search wanted? lst)
 (call/cc
   (lambda (return)
     (for-each (lambda (element)
                 (if (wanted? element)
                     (return element)))
               lst)
     #f)))
```

Listing 2: Suche in einer Liste unter Zuhilfenahme von `call/cc`.

Finden Sie heraus, was der Effekt von `call/cc` in diesem Beispiel ist und was er bewirkt.

1.3 Full Continuations

Im obigen Fall, der sog. Exit Continuation, wurde `call/cc` verwendet, um aus einem beliebig tiefen Aufruf zurückzukehren. Es kann aber auch allgemeiner verwendet werden, um eine Berechnung zu verlassen und später fortzuführen. Betrachten Sie folgende Interpretersitzung mit Guile:

[3] Normalerweise würde man eine solche Prozedur in Scheme rekursiv implementieren.

```
1 guile> (define call/cc call-with-current-continuation)
2 guile> (define return #f)
3 guile> (+ 1 (call/cc
4        (lambda (cont)
5          (set! return cont)
6          1)))
7 2
8 guile> (return 22)
9 23
```

Listing 3: Beispiel einer Full Continuation.

Wieso wird in Zeile 7 der Wert 2 und in Zeile 9 der Wert 23 zurückgegeben? Was bewirkt `call/cc` in diesem Beispiel? Was passiert in Zeile 8 genau?

2 Weitere Diskussion

Diskutieren Sie weiter folgende Fragestellungen schriftlich auf etwa einer DIN A4 Seite.

2.1 Programmierparadigma

Mit Prolog haben Sie bereits eine **logische** Programmiersprache kennengelernt. Scheme ist eine **funktionale** Sprache. Damit gehören beide Sprachen der **deklarativen** Programmierung an.

Fortran ist hingegen eine **strukturierte** Programmiersprache, die also das **imperative** Paradigma implementiert.

Wodurch unterscheiden sich imperative und deklarative Programmierung? Erläutern Sie anhand eines Beispieles. Hierzu bietet sich die Fibonacci-Funktion an.

2.2 Referenzielle Transparenz

Eine wichtige Eigenschaft der deklarativen Programmierung ist die **referenzielle Transparenz**: der Wert eines Ausdruckes hängt nicht vom Zeitpunkt seiner Auswertung ab, sondern ausschließlich von seiner Umgebung. So wird z.B. der Ausdruck $x = x + 1$ als Gleichung interpretiert, die folglich in einer rein funktionalen Sprache nicht gültig sein kann. In imperativen Sprachen ist eine solche Anweisung jedoch möglich und gültig.

In diesem Kontext spricht man auch von **Seiteneffekten**. Eine rein funktionale Programmiersprache ist somit **seiteneffektfrei**. Was ist der Vorteil von Seiteneffektfreiheit?

2.3 Syntax und Lesbarkeit

Wie bewerten Sie die **Lesbarkeit** von Programmen, die in Scheme implementiert sind? Wie kann man etwaigen Problemen entgegnen?

3 Abgabe Ihrer Lösung

Drucken Sie Ihre Lösung aus und werfen Sie den Ausdruck bis zum **2. Juni 2009 um 9:00 Uhr** in den Briefkasten des Lehrstuhles für Softwaretechnik[4]. Vergessen Sie dabei nicht, Ihren Ausdruck mit Ihrem Namen und Ihrer Matrikelnummer zu versehen.

Bringen Sie bitte zum Proseminar ebenfalls einen Ausdruck mit, so dass Sie ihn bei etwaigen Diskussionen vorliegen haben.

4 Links und Hinweise

Sie können den Interpreter "Guile"[5] verwenden, dieser ist auf den Rechnern des Studentenrechnerpooles im Verzeichnis `/installer/import/linux/compilers` installiert. Es handelt sich dabei um Version 1.6.4. Loggen Sie sich bitte auf einem der Sun Compute Server[6] ein.

Ein guter Startpunkt für die Beschäftigung mit Scheme ist das Community-Scheme-Wiki[7].

[4] Gebäude E1 1, neben dem InfoPointes des Rechenzentrums.
[5] `http://www.gnu.org/software/guile/guile.html`
[6] `appsrv1.studcs.uni-saarland.de` oder `appsrv2.studcs.uni-saarland.de`
[7] `http://community.schemewiki.org/`