Informatik 1

Forum: https://forum-db.informatik.uni-tuebingen.de/c/ws2021-info1

Übungsblatt 10 (27.01.2021)

Abgabe bis: Mittwoch, 3.2.2021, 14:00 Uhr

Relevante Videos: bis einschließlich `Informatik 1 - Chapter 10 - Video #48`.

https://tinyurl.com/Informatik1-WS2021

Sprachebene „Die Macht der Abstraktion“

Aufgabe 1: [20 Punkte]

Stellen wir uns ein Szenario vor, in dem sehr viele kurze Listen *nacheinander* übermittelt werden und jeweils einzeln *am Ende* einer Ergebnisliste eingefügt werden müssen:

```
(append (append ... (append (append (list 1)
                                    (list 2))
                            (list 3))
                    ...
            (list 9999))
    (list 10000))
```


Beachte, dass diese Kolonne von `append`-Operationen *links-tief* geklammert ist.

Wenn wir uns die Implementation der Listenfunktion `append` vergegenwärtigen, können wir erahnen, dass das obige ein sehr aufwändiges Unterfangen werden kann: Für jeden Aufruf von `append` wird die bisherige Liste durchlaufen, die leere Liste am Ende gesucht und an ihrer Stelle das neue Element eingehängt. Es ergibt sich eine Laufzeit, die *quadratisch* (nicht linear) mit der Länge der Resultatliste anwächst. Die Ursache dieser Ineffizienz wurde in Chapter 09 im Video #042 und auf Slide 17 thematisiert (siehe die Diskussion der Funktion `backwards`). Dem Effekt kann mit einer alternativen Repräsentation von sogenannten *unvollständigen Listen* durch Funktionen begegnet werden.

Eine *unvollständige Liste* ist eine *Funktion* der Signatur `((list-of` t`) -> (list-of` t`))`, die den Anfang einer Liste repräsentiert (bzw. beinhaltet) und als Argument eine weitere, reguläre Liste (Signatur `(list-of` t`)`) erwartet, mit der der Listenanfang zu einer Ergebnisliste (Signatur `(list-of` t`)`) vervollständigt wird.

(a) Definiere zunächst die parametrische Signatur `(incomplete-list-of t)` der *unvollständigen Listen* mit Elementen der Signatur `t` durch `((list-of t) -> (list-of t))`.

(b) Definiere nun eine Funktion höherer Ordnung `list->incomplete` mit der Signatur

```
(: list->incomplete ((list-of %a) -> (incomplete-list-of %a)))
```

die eine gegebene reguläre Liste `xs` in eine unvollständige Liste l umwandelt. Das Ergebnis von `list->incomplete` ist eine Funktion l, die als Argument eine weitere Liste `ys` erwartet und diese am Ende der Liste `xs` anhängt, um diese zu vervollständigen.

(c) Definiere eine Funktion höherer Ordnung

```
(: incomplete-append ((incomplete-list-of %a) (incomplete-list-of %a)
                      -> (incomplete-list-of %a)))
```

die zwei unvollständige Listen l_1 und l_2 zu einer neuen unvollständigen Liste verknüpft. Die resultierende unvollständige Liste nutzt ein gegebenes Listenende zunächst, um l_2 zu vervollständigen, und verwendet das Ergebnis dann zur Vervollständigung von l_1.

(d) Schreibe zuletzt eine Funktion

```
(: incomplete->list ((incomplete-list-of %a) -> (list-of %a)))
```

die eine *unvollständige Liste* in eine reguläre Liste umwandelt, indem die leere Liste zur Vervollständigung genutzt wird.

(e) Nun man kann mit den unvollständigen Listen genauso programmieren, wie mit den klassischen Listen. Man muss nur für die Konversion von klassischen in unvollständige Listen und zurück sorgen. Es gilt etwa:

```
(incomplete->list (incomplete-append (list->incomplete (list 2 1))
                                     (list->incomplete (list 8 7))))
⇝ (list 2 1 8 7)
```

Nutze `check-property`, um das erwartete Zusammenspiel der drei Funktionen zu testen.

(f) Vergleiche nun die Laufzeiten von (sehr) vielen `append`-Operationen auf den klassischen und den neuen unvollständigen Listen. Dazu findest du vorgegebene Funktionen in der Datei `list-benchmark.rkt`.

i. Gib die Laufzeiten für `test-append` und `test-incomplete-append` (wie in der REPL ausgegeben) an.
Wichtig: Um mit dem *Benchmark* aussagekräftige Zeiten messen zu können, **muss** die Signaturüberprüfung deaktiviert werden (in der Menüleiste: *Racket* ↦ *Signaturüberprüfung deaktivieren*). Anderenfalls wird der Aufwand der eigentlichen Berechnung von der zur Laufzeit sehr teuren Überprüfung der Signaturen überlagert. Vergiss nicht, die Signaturüberprüfung anschließend wieder zu aktivieren.

ii. Betrachte die Ausdrücke `(append (append xs ys) zs)` und `(append xs (append ys zs))`. Sind die Ausdrücke äquivalent? Wenn du die Wahl hättest, welche Variante würdest du einsetzen und wieso?

iii. Ersetze in dem folgenden *links-tief* geklammerten Ausdruck zunächst die Funktionen `incomplete->list`, `incomplete-append` und `list->incomplete` durch ihre Definitionen, so dass nur die Funktion `append`, `lambda`-Ausdrücke und Listen-Literale verbleiben.
Führe dann (wiederholt) die Reduktionsregel `apply` λ durch, bis du einen Ausdruck erhältst der keine Lambdas mehr beinhaltet. Begründe mit dem Resultat, warum *unvollständige Listen* so effizient sind.

```
(incomplete->list
 (incomplete-append (incomplete-append (list->incomplete xs)
                                       (list->incomplete ys))
                    (list->incomplete zs)))
```


Aufgabe 2: [20 Punkte]

Die Kreiszahl π lässt sich durch die folgende Gleichung annähern:

$$\pi = \sqrt{6 \cdot \sum_{i=1}^{\infty} \frac{1}{i^2}}$$

Der Ausdruck $\sum_{i=1}^{\infty} \frac{1}{i^2}$ stellt dabei eine (unendliche) Reihe dar. Ihr n-tes Glied (d.h. die n-te Partialsumme) wird berechnet als $s_n = \sum_{i=1}^{n} \frac{1}{i^2}$.

Die ersten Glieder nehmen also die folgenden Werte an:

$$\begin{aligned}
s_1 &= 1 \\
s_2 &= 1 + \frac{1}{4} = 1.25 \\
s_3 &= 1 + \frac{1}{4} + \frac{1}{9} \approx 1.36 \\
s_4 &= 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} \approx 1.42 \\
s_5 &= \cdots
\end{aligned}$$

In dieser Aufgabe gilt es, mithilfe von *Streams* die obige Annäherung von π zu implementieren.

Hinweis: Die im Forum bereitgestellte Datei `streams.rkt` enthält die für die Arbeit mit *Streams* nötigen Definitionen `promise`, `force` und `stream-of` sowie den parametrischen Record `cons`. Zusätzlich sind darin die aus der Vorlesung bekannten Funktionen `from` und `stream-take` vorgegeben.

(a) Schreibe zunächst eine Funktion

```
(: stream-drop (natural (stream-of %a)-> (stream-of %a))),
```

die, gegeben eine natürliche Zahl n, die ersten n Elemente eines Eingabestreams verwirft. **Beispiel**:

```
(stream-take 3 (stream-drop 5 (from 1))) ⇝ (list 6 7 8)
```

(b) Schreibe eine Funktion

```
(: stream-map ((%a -> %b)(stream-of %a)-> (stream-of %b))),
```

die eine Funktion f und einen Stream s akzeptiert und den Stream zurückliefert, der aus der Anwendung von f auf die Elemente von s entsteht.
Beispiel:

```
(stream-take 3 (stream-map (lambda (x) (* x x)) (from 1))) ⇝ (list 1 4 9)
```

(c) Verwende `steam-map` und die aus der Vorlesung bekannte Funktion `from`, um eine Funktion

```
(: pi-series (stream-of real))
```

zu formulieren, die einen Stream der Zahlen $\frac{1}{i^2}$ für $i = \{1, 2, \ldots, \infty\}$ erzeugt.
Beispiel:

```
(stream-take 3 pi-series) ⇝ (list 1 1/4 1/9)
```

(d) Schreibe eine Funktion

```
(: stream-sum ((stream-of number) -> (stream-of number)))
```

die einen Stream der laufenden Summe über die Elemente des übergebenen Streams s erzeugt. Wenn s der Stream $x_1, x_2, x_3, \ldots$ ist, dann ist `(stream-sum s)` der Stream $x_1, x_1 + x_2, x_1 + x_2 + x_3, \ldots$.
Mathematisch ausgedrückt: Fasst man den Eingabestream als Folge auf, berechnet `stream-sum` einen Stream der Partialsummen der entsprechenden Reihe.
Beispiele:

```
(stream-take 4 (stream-sum (from 1))) ⇝ (list 1 3 6 10)

(stream-take 3 (stream-sum pi-series)) ⇝ (list 1 5/4 49/36)
```


(e) [2 Punkte] Verwende nun `pi-series`, `stream-sum` und `stream-drop`, um eine Funktion

```
(: approx-pi (natural -> real))
```

mit einem Parameter n zu definieren, die mittels der n-ten Partialsumme der obigen Reihe eine Näherung für π berechnet.
Beispiele:

```
(approx-pi 1) ⇝ 2.449
(approx-pi 2) ⇝ 2.739
(approx-pi 3) ⇝ 2.858
...
(approx-pi 1000) ⇝ 3.141
```