Informatik 1
Forum: https://forum-db.informatik.uni-tuebingen.de/c/ws2021-info1

Übungsblatt 13 (17.02.2021)

Abgabe bis: Mittwoch, 24.02.2021, 14:00 Uhr

Relevante Videos: bis einschließlich `Informatik 1 - Chapter 13 - Video #065`.
https://tinyurl.com/Informatik1-WS2021

Sprachebene „Die Macht der Abstraktion – fortgeschritten"

Aufgabe 1: [14 Punkte]

In dieser Aufgabe soll das effiziente Sortierverfahren **Merge Sort** implementiert werden.

(a) Implementiert die Funktion `split` mit folgender Signatur:

```
(: split ((list-of %a) -> (tuple-of (list-of %a) (list-of %a)))).
```

`(split xs)` teilt die Liste `xs` in ein Tupel zweier (möglichst) *gleich langer* Listen `ys` und `zs`. Ob ein Element `ys` oder `zs` zugeordnet wird, ist unerheblich. Achtet unbedingt darauf, die Liste `xs` dabei **nur einmal** zu durchlaufen.

Beispiele:

```
(split (list 2 1 8 7)) ⇝ (make-tuple (list 2 1) (list 8 7))
(split (list 3 2 7))   ⇝ (make-tuple (list 3 2) (list 7))
```


(b) Implementiert die Funktion `merge-by` mit folgender Signatur:

```
(: merge-by ((%a %a -> boolean) (list-of %a) (list-of %a) -> (list-of %a))).
```


`(merge-by lt? xs ys)` führt die bezüglich des Sortierkriteriums `lt?` *bereits sortierten* Listen `xs` und `ys` so zusammen, dass als Resultat wieder eine *sortierte* Liste entsteht.

Beispiele:

```
(merge-by < (list 1 3) (list 1 8)) ⇝ (list 1 1 3 8)
```

(c) Implementiert die Funktion `mergesort` mit folgender Signatur:

```
(: mergesort ((%a %a -> boolean) (list-of %a) -> (list-of %a))).
```

`(mergesort lt? xs)` sortiert eine gegebene Liste `xs` ensprechend eines Sortierkriteriums `lt?`. Für eine Liste `xs` mit zwei oder mehr Elementen geht `mergesort` dazu wie folgt vor:

(i) Nutze `split`, um `xs` zu halbieren; erhalte zwei Listen `ys` und `zs` als Zwischenergebnis.
(ii) Wende `mergesort` *jeweils* rekursiv auf `ys` und auf `zs` an. Die beiden entstehenden Listen sind nun sortiert.
(iii) Nutze zuletzt `merge-by`, um die beiden sortierten Listen wieder zu einer einzelnen (ebenfalls sortierten) Ergebnisliste zusammenzufügen.

Beispiel:

```
(mergesort < (list 8 1 3 1)) ⇝ (list 1 1 3 8)
```

Aufgabe 2: [16 Punkte] **Hinweis:** Testfälle sind für diese und die folgende Aufgabe **nicht** notwendig. Wir stellen euch eine Datei `lambda.rkt` zur Verfügung, in der ihr die relevanten Definitionen aus dem Chapter 13 vorfindet.

In der Vorlesung haben wir bereits einige Literale und Operationen darauf im λ-Kalkül definiert. Jetzt seid ihr gefragt. Baut die folgenden Operationen ganz ähnlich wie im Chapter 13, Slides 20 ff.

(a) Implementiert die beiden Booleschen Operationen `(: OR_ λ-term)` und `(: NOT_ λ-term)`. Es soll gelten (hier steht $\overline{a}$ für die Darstellung des Booleans a im λ-Kalkül, genau wie auf Slide 20 im Chapter 13 definiert):

$$\begin{array}{rcl} \texttt{(ao (list (list OR\_ } \overline{a}\texttt{) } \overline{b}\texttt{))} & \rightsquigarrow & \overline{a \vee b} \\ \texttt{(ao (list NOT\_ } \overline{a}\texttt{))} & \rightsquigarrow & \overline{\neg a} \end{array}$$

`ao` ist die euch bekannte Funktion, die die *Applicative Order*-Reduktion eines λ-Ausdrucks durchführt.

(b) Implementiert die arithmetische Operation `(: EXPT_ λ-term)`. `EXPT_` berechnet die Exponentiation x^y zweier natürlichen Zahlen x und y, die als *Church-Numerale* repräsentiert sind. Es soll gelten:

$$\texttt{(ao (list (list EXPT\_ (CHURCH } x\texttt{)) (CHURCH } y\texttt{)))} \rightsquigarrow \texttt{(CHURCH } x^y\texttt{)}$$

Spoiler-Warnung: Hinweis auf eine mögliche Implementation: `((ercrng l ((pheel *) k)) 1)`.

(c) Implementiert die beiden **rekursiven** Funktionen `(: LENGTH_ λ-term)` und `(: SUM_ λ-term)`. Beiden werden auf Listen in der Repräsentation im λ-Kalkül angewandt, wie sie auf Slide 21 im Chapter 13 definiert wurde.

Sei `xs` $\equiv$ `(list->λ-list (list (CHURCH` x_1`) (CHURCH` x_2`) ... (CHURCH` x_n`)))`, wobei die x_i natürliche Zahlen sind. Es soll gelten:

$$\begin{array}{rcl} \texttt{(no (list LENGTH\_ xs))} & \rightsquigarrow & \texttt{(CHURCH } n\texttt{)} \\ \texttt{(no (list SUM\_ xs))} & \rightsquigarrow & \texttt{(CHURCH } (x_1 + x_2 + \cdots + x_n)\texttt{)} \end{array}$$

Hinweise: Orientiert euch an der Implementation der rekursiven Funktion `FAC` auf Slide 25. Für diese Teilaufgabe benötigt ihr den Kombinator **Y**, wie er als `Y` auf Slide 23 im Chapter 13 definiert wurde. Da **Y** im Spiel ist, **müsst** ihr die vordefinierte Funktion `no` zur *Normal Order*-Reduktion einsetzen, um die resultierenden λ-Ausdrücke reduzieren zu können.

Aufgabe 3: [10 Punkte] **Hinweis:** Testfälle sind auch für diese Aufgabe **nicht** notwendig.

Achtung, festhalten! *Jede Funktion* im λ-Kalkül lässt sich gleichwertig durch einen Ausdruck darstellen, in dem *ausschliesslich* die **drei Kombinatoren S, K, I** vorkommen. Da der λ-Kalkül jede überhaupt berechenbare Funktion ausdrücken kann, sind also *alle* Programme allein auf eine Kombination von **S**, **K**, **I** zurückzuführen—*mind blown!*[1]

Wir haben die drei Kombinatoren[2] bereits in Chapter 13 definiert (drei analoge Racket-Definitionen `S`, `K`, `I`—jeweils mit Signatur `λ-term`—findet ihr ebenfalls im File `lambda.rkt`):

$$\begin{array}{rcl} \mathbf{S} & \equiv & (\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z))))) \\ \mathbf{K} & \equiv & (\lambda x.(\lambda y.x)) \\ \mathbf{I} & \equiv & (\lambda x.x) \end{array}$$

Die Transformation $\mathcal{T}[e]$ tritt den Beweis für die Behauptung oben an: der λ-Ausdruck e wird in einen Ausdruck übersetzt, der ausschliesslich **SKI**-Kombinatoren enthält (insbesondere enthält das Resultat keine λ-Abstraktionen mehr):

$$\begin{array}{crcll} ① & \mathcal{T}[(\lambda x.e)] & \to & (\mathbf{K}\ \ \mathcal{T}[e]), & \text{falls } x \text{ keine freie Variable in Ausdruck } e \text{ ist} \\ ② & \mathcal{T}[(\lambda x.(\lambda y.e))] & \to & \mathcal{T}[(\lambda x.\mathcal{T}[(\lambda y.e)])] & \\ ③ & \mathcal{T}[(\lambda x.(e_1\ e_2))] & \to & ((\mathbf{S}\ \ \mathcal{T}[(\lambda x.e_1)])\ \ \mathcal{T}[(\lambda x.e_2)]) & \\ ④ & \mathcal{T}[(\lambda x.x)] & \to & \mathbf{I} & (x \text{ ist eine Variable}) \\ ⑤ & \mathcal{T}[(e_1\ e_2)] & \to & (\mathcal{T}[e_1]\ \ \mathcal{T}[e_2]) & \\ ⑥ & \mathcal{T}[x] & \to & x & \end{array}$$

(Die Regeln ②, ③, oder ④ kommen nur zum Einsatz, falls ① nicht zutrifft.)

[1] Es gibt tatsächlich Implementationen von Programmiersprachen, die Programme allein nach **SKI** übersetzen und diese Kombinatoren dann mittels eine (der JVM ähnlichen) virtuellen Maschine ausführen. Die Programmiersprache *Miranda* ist ein Beispiel: https://www.cs.kent.ac.uk/people/staff/dat/miranda/.

[2] Ein *Kombinator* ist ein Ausdruck des λ-Kalküls, der keine freien Variablen beinhaltet.

Schreibt eine Funktion `(: ski (λ-term -> λ-term))`, die die Transformation $\mathcal{T}$ implementiert.

ACHTUNG: In eurer Implementation von `ski` **müsst** ihr die Kombinatoren durch die Symbole `'S`, `'K` und `'I` darstellen (Beispiel: Regel 1 wird Code der Form `(list 'K ...)` enthalten). Wir haben die *Applicative Order*-Reduktion `ao` im File `lambda.rkt` so modifiziert, dass die Symbole `'S`, `'K` und `'I` korrekt erkannt und reduziert werden. Ihr könnt eure `ski`-Funktion also beispielsweise debuggen, in dem ihr die Ausgabe von

`(ao` e`)` und `(ao (ski` e`))`

vergleicht (die resultierenden Terme sollten äquivalent—wenn auch nicht unbedingt identisch—sein).