Prof. Dr. T. Grust, B. Dietrich, C. Duta, D. Hirn WS 2020/2021

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

Übungsblatt 14 (24.02.2021)

Abgabe bis: Mittwoch, 03.03.2021, 14:00 Uhr

Sprachebene „Die Macht der Abstraktion“

Aufgabe 1: [10 Punkte]

Mittels der *Tupperschen Ungleichung* (*Tupper's Formula*) kann man ein Gefühl dafür entwickeln, wieviel Information in einer (*sehr!*) großen natürlichen Zahl `k` steckt. Wir visualisieren diese Ungleichung hier.

Seien (x,y) zwei natürliche Zahlen, dann berechnet die *Tuppersche Ungleichung* daraus einen *Booleschen* Wert (hier bezeichnet $\lfloor x \rfloor$ die bekannte Funktion `(floor` x`)`, eine Implementation von `mod` stellen wir euch in File `tupper.rkt` zur Verfügung):

$$\frac{1}{2} < \left\lfloor \mathrm{mod}\left(\left\lfloor \frac{y}{17} \right\rfloor 2^{-17\lfloor x \rfloor - \mathrm{mod}(\lfloor y \rfloor, 17)}, 2 \right) \right\rfloor \quad .$$

Wenn wir $x \in [0, 105]$ und $y \in [\mathtt{k}, \mathtt{k} + 16]$ wählen, dann definiert die Ungleichung ein Bild aus 106×17 Pixeln (wenn die Ungleichung `#t` liefert, ist das Pixel an der Koordinate (x,y) gesetzt). Diese Pixelbilder sollt ihr zeichnen (dazu benötigt ihr das Teachpack `image2`[1]).

Wichtig: Verwendet in den Teilaufgaben (c) und (d) **keine explizite Rekursion**, sondern nur eingebaute Listenverarbeitungsfunktionen wie `map` und `fold`, sowie die in `tupper.rkt` vorgegebene Funktion `from-to`.

(a) Definiert zunächst eine Funktion

```
(: tupper-formula (natural natural -> boolean))
```

die für zwei gegebene natürliche Zahlen x und y das Ergebnis der *Tupperschen Ungleichung* (Signatur `boolean`) zurückgibt.

(b) Konstruiert eine Funktion

```
(: pixel (boolean -> image))
```

die einen Booleschen Wert in einen schwarzen (`#t`) oder weissen Pixel (`#f`) abbildet. Nutzt die Funktion `(rectangle 1 1 "solid" ...)` aus Teachpack `image2`, um den Pixel zu erzeugen.

(c) Definiert die Funktion

```
(: tupper-pixels (natural -> list-of (list-of image)))
```

so dass `(tupper-pixels k)` die *Tuppersche Ungleichung* für alle Punkte (x,y) im Bereich von $x \in [0, 105]$ und $y \in [\mathtt{k}, \mathtt{k} + 16]$ auswertet und die resultierenden Booleschen Werte in schwarze/weisse Pixel übersetzt. Das Ergebnis ist eine Liste von 17 Zeilen (y-Intervall), die jeweils als Listen mit je 106 Pixeln (x-Intervall) dargestellt sind.

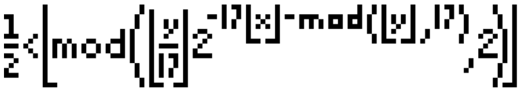
(d) Definiert zuletzt die Funktion

```
(: tupper-image (natural -> image))
```

so dass `(tupper-image k)` die von `tupper-pixels` gelieferten Pixellisten zu einem Bild mittels der Funktionen `above` und `beside` aus dem Teachpack `image2` zusammenfügt. `tupper-image` kann die Funktion `(scale` s img`)` aus dem Teachpack nutzen, um das Pixelbild bzgl. Faktor s auf eine sinnvolle Größe zu skalieren.

[1] Dokumentation zum Teachpack `image2` findet ihr unter https://docs.racket-lang.org/teachpack/2htdpimage.html.

Das File `tupper.rkt` enthält beispielhafte Werte für die große natürliche Zahl `k`. Für die dort vorgegebene Zahl `k1` ≡ `96093`⟨$\cdots$ *534 weitere Ziffern* $\cdots$⟩`04719` ergibt sich übrigens das Bild



Ihr könnt euch auf https://keelyhill.github.io/tuppers-formula/ in einem web-basierten Pixeleditor eure eigenen Bilder und deren `k`-Werte konstruieren.

Aufgabe 2: [4 Punkte]

Listen können als Repräsentation von *Mengen* verstanden werden. Wir gehen davon aus, dass eine solche Liste keine Duplikate enthält.

Die Funktion $\mathcal{P}(S)$ berechnet die Menge aller Untermengen einer Menge S. $\mathcal{P}(S)$ enthält dabei immer die leere Menge $\varnothing$ sowie S selbst:

$$\mathcal{P}(\{1,2,3\}) = \big\{\{1,2,3\}, \{1,2\}, \{2,3\}, \{1,3\}, \{1\}, \{3\}, \{2\}, \varnothing\big\}$$

Schreibt eine Funktion `(: subsets ((list-of %a)-> (list-of (list-of %a))))`, die die Menge aller Untermengen einer Menge mit beliebigen Elementen berechnet.

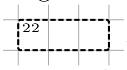
Hinweis: Um die Teilmengen der Liste `(make-pair x xs)` zu berechnen, sammelt alle Teilmengen von `xs` *zweimal* auf:

(a) einmal unverändert und

(b) einmal, nachdem in jede Teilmenge das Element `x` eingefügt wurde.

Da wir Mengen repräsentieren, ist die Reihenfolge der Teilmengen und der Elemente innerhalb der Teilmengen im Ergebnis von `subsets` **nicht relevant**. Geht zudem davon aus, dass die Eingabeliste duplikatfrei ist.

Aufgabe 3: [14 Punkte]

Prof. Grust liebt Sudokus in allen Varianten. In der Variante *Killer Sudoku* gibt es im Sudoku-Grid neben den üblichen 9 Zeilen, Spalten und 3×3-Boxen auch die sog. *Cages*: . In diesem *Cage* der Größe 3 müssen drei unterschiedliche Ziffern (1, . . . ,9) eingetragen werden, deren Summe 22 beträgt. Gute Sudoku-Spieler stellen sich dann Fragen wie: "*Welche Kombinationen aus drei Ziffern ergeben 22?*" oder "*Ist in allen möglichen Kombinationen eine 9 enthalten?*". *Wirklich* gute Spieler kennen sämtliche Kombinationen aus n Ziffern, die eine Summe s ergeben, auswendig (im Beispiel: $n = 3$, $s = 22$).[2] Mittelmässige Spieler benötigen dazu eine tabellarische Aufstellung. Baut eine solche Tabelle für Prof. Grust und geht dazu wie folgt vor:

(a) Wir kümmern uns zunächst um die Ziffernkombinationen selbst:

   i. Definiert eine Signatur `digit`, die nur die gültigen Sudoku-Ziffern `1`, `2`, . . . , `9` akzeptiert.

   ii. Schreibt eine Funktion `(: cage-sums (natural -> (list-of (list-of digit))))`, so dass der Aufruf `(cage-sums` s`)` die Menge aller möglichen duplikatfreien Ziffernkombinationen (jeweils dargestellt als `(list-of digit)`) mit Summe s berechnet. Hierbei gilt $s \in [1, 45]$.
   Beispiel:
   `(cage-sums 7)` ⇝ `(list (list 7) (list 3 4) (list 2 5) (list 1 6) (list 1 2 4))`
   Die Elementreihenfolge ist beliebig.
   Hinweis: Nutzt die Funktion `subsets` aus Aufgabe 2 und Listenfunktionen (`map`, `filter`, . . .) anstatt explizite Rekursion einzusetzen. (So spät in der Vorlesung ist das eine Stilfrage, die in die Bepunktung eingeht.)

(b) Jetzt geht es an die tabellarische Ausgabe. Die Tabelle wird alle Ziffernkombinationen einer gegebenen Größe n enthalten und zeigen, wie diese jeweils die Summen $s = 1, 2, \ldots, 45$ ergeben. Abbildung 1 zeigt die Ausgabe in der REPL für $n = 3$. Die Ausgabe hat *immer* $2 + 45$ Zeilen (Header + Zeilen mit Ziffernkombinationen). Geht wie folgt vor:

[2] Wer sich für Sudokus in irgendeiner Form interessiert, der/dem kann nur der YouTube-Channel *Cracking the Cryptic* ans Herz gelegt werden. Im folgenden Videosegment seht ihr, wie Simon Anthony—vormals Mitglied des britischen Sudoku-Nationalteams—einen (virtuellen) 22-*Cage* der Größe 3 für den nächsten Lösungsschritt nutzt: https://youtu.be/11pjL7saUno?t=2420.

```
Cage sum    Possible digit sets
--------    -------------------
1
2
3
4
5
6           123
7           124
8           134, 125
9           234, 135, 126
10          235, 145, 136, 127
11          245, 236, 146, 137, 128
12          345, 246, 237, 156, 147, 138, 129
13          346, 256, 247, 238, 157, 148, 139
14          356, 347, 257, 248, 239, 167, 158, 149
15          456, 357, 348, 267, 258, 249, 168, 159
16          457, 367, 358, 349, 268, 259, 178, 169
17          467, 458, 368, 359, 278, 269, 179
18          567, 468, 459, 378, 369, 279, 189
19          568, 478, 469, 379, 289
20          578, 569, 479, 389
21          678, 579, 489
22          679, 589
23          689
24          789
25
26
[... die Zeilen 27 bis 43 enthalten keine Ziffernkombinationen ...]
44
45
```

Abbildung 1: Tabelle mit den *Cages* der Größe $n = 3$ und ihre Summen. `124` ist die Ausgabe für die Ziffernkombination `(list 1 2 4)`. Ein *Cage* der Größe 3 kann bspw. nicht die Summe 4 ergeben (leere Zeile 4).

i. Konstruiert zunächst die Funktion `(: digits->string ((list-of digit) -> string))`, die eine Ziffernkombination kompakt als String repräsentiert. Dabei kann die eingebaute Funktion `number->string` hilfreich sein.
   Beispiel:

   `(digits->string (list 1 2 4))` ⇝ `"124"`

ii. Baut eine Funktion `(: intersperse (%a (list-of %a) -> (list-of %a))`, die ein Element *zwischen* alle Elemente einer Liste platziert.
   Beispiele:

   ```
   (intersperse ", " (list "S" "K" "I")) ⇝ (list "S" ", " "K" ", " "I")
   (intersperse 1    (list 42))          ⇝ (list 42)
   (intersperse #t   empty)              ⇝ empty
   ```

iii. Baut eine Funktion `(: row (natural natural -> string))`, die eine Tabellenzeile der Ausgabe erzeugt.
   Beispiel:

   `(row 3 9)` ⇝ `"9\t\t234, 135, 126"`

   erzeugt die Zeile der Ziffernkombinationen der Größe $n = 3$ mit Summe $s = 9$ in der Ausgabe in Abbildung 1 (`"\t"` steht für einen *Tabulator*, siehe die Hinweise unten).

iv. Baut schliesslich die Funktion `(: table (natural -> string))`, so dass `(table` n`)` die Ausgabe der *Cage*-Tabelle für alle Ziffernkombinationen der Größe n berechnet. Abbildung 1 wurde mittels

   `(write-string (table 3))`

   in der REPL erzeugt.

Hinweise:

- Die Funktion `strings-list->string` konkateniert eine Liste von Strings zu einem String.
- Trennt die Zeilen der Tabelle mit einem *Newline* (String `"\n"`) und trennt die Spalten der Tabelle mit einem horizontalen *Tabulator* (String `"\t"`). Letzteres bewahrt euch davor, für die formatierte Ausgabe Spaltenbreiten berechnen/ausgleichen zu müssen.

Aufgabe 4: [5 Punkte]

Die Operation `(: delay (%a -> (promise %a)))` verzögert die Auswertung ihres Argumentes und erzeugt stattdessen ein Versprechen (Signatur `promise`), das erst später—bei eventuellem Bedarf—mittels `(: force ((promise %a) -> %a))` eingelöst werden kann (vgl. *Chapter 13 Folien 22 und 23*).

Dabei ist es bedeutend, dass `delay` als *syntaktischer Zucker* und nicht als reguläre Funktion definiert ist. Zeige dies mit Hilfe der *Reduktionsregeln* $[\![\cdot]\!]^k$ für Scheme. **Wichtig:** Halte dich dabei **streng** an die Notation aus der Vorlesung zu *Chapter 3 Folien 3ff.* und lasse keine Schritte aus!

(a) Zeige zunächst, dass während der Reduktion von `(delay (+ 40 2))` das Argument `(+ 40 2)` *nicht* ausgewertet wird, sofern `delay` als *syntaktischer Zucker* definiert ist:

`(delay` e`)` $\equiv$ `(lambda ()` e`)`

(b) Zeige dann, dass während der Reduktion von `(delay (+ 40 2))` das Argument wider Erwarten *doch* ausgewertet wird, sofern wir `delay` als reguläre Funktion definieren:

```
(define delay        ; /!\ fragwürdige Definition
  (lambda (e)
    (lambda () e)))
```

(c) Zeige zuletzt, dass mit Hilfe von `force` das Versprechen `(lambda () (+ 40 2))` tatsächlich zur Auswertung gebracht wird. Reduziere dazu den Ausdruck `(force (lambda () (+ 40 2)))` und nutze die bekannte Definition von `force`:

```
(define force
  (lambda (p) (p)))
```

Aufgabe 5: [7 Punkte]

Die *Folien 7 und 8* in *Chapter 9* zeigen drei Konstruktionsanleitungen `(: f (natural ->` t`))` für Rekursion über natürlichen Zahlen.

(a) Wählt $t \equiv$ `natural` und ergänzt die drei Konstruktionsanleitungen zu drei Funktionen `f0`, `f1`, `f2`, deren Resultat jeweils angibt, wieviele rekursive Aufrufe die Funktion zur Berechnung durchgeführt hat.[3] Damit gilt beispielsweise `(f1` n`)` $\rightsquigarrow$ `4` für $n = 3$.

Wie verhält sich die Anzahl der rekursiven Aufrufe der drei Funktionen, wenn wir $n = 0, 1, 2, \ldots$ wachsen lassen? Das wollen wir mit zweidimensionalen Plots visualisieren. (Dazu benötigt ihr das Teackpack `image2`.)

(b) Definiert dazu die Signatur `(define point (signature (tuple-of real real)))`, die Punkte $p = (x,y)$ in der Ebene repräsentiert. Konstruiert eine Funktion

`(: add-lines (image (list-of point) -> image))`

so dass `(add-lines` img `(list` p_1 p_2 p_3 `...` p_m`))` die Liste der Punkte zu einem **durchgehenden Linienzug** p_0—p_1—p_2—$\cdots$—p_{m-1}—p_m verbindet und diesen zum bestehenden Bild img hinzufügt. Dabei hilft euch die Funktion `add-line` aus dem Teachpack `image2` (setzt den Parameter `pen-or-color` der Funktion `add-line` auf `"black"`).[4]

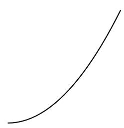
(c) Nutzt eure Funktion `add-lines` aus (b), um die Funktion

`(: plot/xy (real real natural natural (natural -> natural) -> image))`

zu konstruieren. Ein Aufruf `(plot/xy` sx sy s e f`)` geht wie folgt vor:

- Funktion f wird für die Parameter $n \in \{s, s+1, s+2, \ldots, e-1, e\}$ ausgewertet. Bilde daraus die Punkte $p_n = (n, (f\ \ n))$.
- Konstruiere einen Linienzug aus den Punkten $p_s, p_{s+1}, \ldots, p_e$.
- Das Resultat ist ein Bild, das diesen Linienzug enthält, nach dem dieser in x/y-Richtung mit den Faktoren sx/sy skaliert wurde (dabei hilft die Funktion `scale/xy` aus dem Teachpack `image2`).

Beispiel: Plot für die Funktion x^2 im Wertebereich $x = \{1,2,\ldots,10\}$:

`(plot/xy 5 0.5 0 10 (lambda (x) (* x x)))` $\rightsquigarrow$ 

(d) Nutzt `plot/xy` dreimal, um für die Funktionen `f0`, `f1` und `f2` aus (a) die jeweils sehr charakteristische Entwicklung der Anzahl der rekursiven Aufrufe bei wachsendem $n = 0, 1, 2, \ldots$ zu visualisieren. Wählt dazu die Parameter sx, sy, s, e von `plot/xy` jeweils geeignet.

[3] Funktion `f0` entspricht der Konstruktionsanleitung auf *Folie 7*, `f1` und `f2` entsprechenden den beiden Konstruktionsanleitungen auf *Folie 8*.

[4] Die Dokumentation zu `add-line` findet ihr auf https://docs.racket-lang.org/teachpack/2htdpimage.html.