Dr. Steffen Jost
Leah Neukirchen

Ludwig-Maximilians-Universität München
Institut für Informatik
3. Juli 2018

11. Übung zur Vorlesung

Programmierung und Modellierung

A11-1 ***Redex*** Identifizieren Sie alle Redexe in den folgenden Programmausdrücken, und geben Sie an, welche davon innerste und äußerste Redexe sind.

a) `(1 + 2) * (4 / 5)`

b) `snd (1 + (2 + 3), 4 + 5)`

c) `(\x -> (1 * 2) + x) (3+4)`

d) `(\f -> f (1 * 2)) (\x -> 3+4)`

Wobei die Funktion `snd :: (a,b) -> b` üblicherweise definiert ist durch `\(x,y) -> y`

A11-2 ***Auswertestrategie*** Werten Sie den Programmausdruck entweder mit der Auswertestrategie Call-By-Name oder Call-By-Value Schritt-für-Schritt aus, wie in der Vorlesung gezeigt. Unterstreichen Sie jeweils den Redex, welchen Sie im nächsten Schritt auswerten.

Welche Auswertestrategie haben Sie jeweils gewählt und warum?

a) `(\(x,y) -> y) (1 + (2 + 3), 4 + 5)`

b) `(\x -> x + x) ((\y -> y * y) (1+1))`

c) `(\x -> (\y -> y)) ((\w -> w w) (\w -> w w)) 42`

A11-3 ***Umgang mit Monaden***

Schnell mit Papier und Bleistift: Zu was werten die folgenden Ausdrücke aus?

a) `forM [1..3] (\x -> if even x then Just x else Nothing)`

b) `mapM print [1..4]`

c) `forM [1..3] (\x -> print x >> return (x+1))`

d) `runPar $ do { xs <- forM [1..3] (\x -> spawnP (x*x)); mapM get xs }`

A11-4 ***Verzögerte Auswertung***

Betrachten Sie folgendes Programm:

```
ps = iterate (+1) 0
qs = iterate (*2) 1
rs = iterate (^2) 2

iterate f x =
  x : iterate f (f x)
foo (_:x:xs) ys@(y:_) zs
  | x  > y    = x
  | otherwise = foo ys zs xs
```

Aufgrund der verzögerten Auswertestrategie von Haskell wird z.B. die Liste `ps` anfangs nur als ein Verweis auf den Code `iterate (1+) 0` abgespeichert. Sobald aber auf das erste Element dieser Liste zugegriffen wird, zeigt `ps` danach auf die Liste `0: iterate (+1) (0+1)`. Wird später dann auch noch das zweite Element benötigt, so zeigt der Bezeichner `ps` nun auf die Liste `0:1: iterate (+1) (1+1)` im Speicher.

Bis zu welchem Element werden die Listen `ps`, `qs`, `rs` im Speicher ausgewertet, wenn nur der Aufruf `foo ps qs rs` ausgeführt wird?

H11-1 ***Redex II*** (2 Punkte; Abgabe: `H11-1.txt` oder `H11-1.pdf`)
Identifizieren Sie alle Redexe in den folgenden Programmausdrücken, und geben Sie an, welche davon innerste und äußerste Redexe sind.

a) `(\(x,y,z) -> (0+1,y*z)) (2,3+4,(\u->5) 6)`

b) `(\g z -> (\f x -> f (f x)) (\y -> 1+2) 3)`

H11-2 ***Auswertestrategie II*** (2 Punkte; Abgabe: `H11-2.txt` oder `H11-2.pdf`)
Werten Sie den Programmausdruck entweder mit der Auswertestrategie Call-By-Name oder Call-By-Value schrittweise aus. Unterstreichen Sie jeweils den ausgewerteten Redex.

```
(\x -> \y -> y ( y x )) 9 ((\a -> (\b -> b+b)) (7*(2+3)))
```

Welche Auswertestrategie haben Sie gewählt und warum?

H11-3 ***Kodeknacker*** (2 Punkte; Datei `H11-3.hs` als Lösung abgeben)
Heimtückischerweise konnten Sie eine Übertragung der Musterlösung zur kommenden ProMo-Klausur abfangen! Leider wurde jede Teilaufgabe mit einem anderen Schlüssel verschlüsselt. Glücklicherweise konnte Ihnen ein fleißiger (aber gewissenloser) Kommilitone, der eine Vorlesung über Kryptographie aufmerksam besucht hatte, Ihnen ein Haskell-Programm zum Knacken der Verschlüsselung erstellen (siehe beiliegende Datei `H11-3.hs`) Das Programm funktioniert, doch die Berechnung pro Schlüssel dauert viel zu lange um rechtzeitig vor der Klausur fertig zu werden! Ihr Kommilitone versicherte, dass eine weitere grundlegende Optimierung dieses Algorithmus sehr unwahrscheinlich ist. Als letzter Ausweg bleibt also nur die Parallelisierung des gegebenen sequentiellen Programms!

a) Beschleunigen Sie die Ausführung der Vorlage durch den Einsatz der `Par`-Monade! Mit 4 Kernen sollten Sie eine Beschleunigung um Faktor ≥ 3 erreichen können.

   Das verwendete Kryptoverfahren ist zur Lösung der Aufgabe nicht so wichtig. Es reicht hier, die zeitaufwändige Funktion zu identifizieren:

   ```
   slowFactor :: (Integer,Integer) -> Integer -> Maybe Integer
   ```

   Der Aufruf `slowFactor (lo,hi) n` sucht einen Faktor von `n` im Intervall `(lo,hi)`. Damit ist die Idee zur Parallelisierung auch schon klar: Wir zerlegen das gegebene Intervall in mehrere ungefähr gleich-große Teilintervalle. Anstatt *einem* langwierigem Aufruf von `slowFactor` mit einem großen Intervall, führen wir dann *mehrere* parallele Aufrufe von `slowFactor` mit kleinen Teilintervallen durch. Damit können wir die Suche nach einem der Primfaktoren beschleunigen!

   Hintergrundwissen für Interessierte, zur Lösung nicht relevant:
   In dieser Aufgabe arbeiten wir mit dem RSA Kryptoverfahren. Dies ist ein asymmetrisches Verfahren, d.h. jeder kann mit einem öffentlich bekannten Schlüssel eine Nachricht kodieren, welche nur der Empfänger mit seinem privaten Schlüssel lesen kann. Die Sicherheit beruht dabei auf einer sogenannten "Einwegfunktion mit Falltür": Die Berechnung der Kodierung geht schnell (wie z.B. Multiplikation zweier Primzahlen) während die Umkehrfunktion (im Beispiel also Primfaktorzerlegung) eine aufwendige Berechnung ist, falls man nicht bereits einen der beiden Primfaktoren im Voraus kennt. Bei RSA gibt es einen öffentlich bekannten Schlüssel (e, N) und einem geheimen Schlüssel (d, N).

Dabei ist N ein Produkt aus zwei großen Primzahlen. Zum Brechen der Kodierung muss man "nur" N in seine beiden Primfaktoren zerlegen.

b) Sobald einer der beiden Primfaktoren gefunden wurde, kann die Suche abgebrochen werden, da der andere Primfaktor wegen $N = p \cdot q$ dann ja leicht durch eine einzige Division berechnet werden kann. Deshalb wäre es doch noch schneller, wenn die komplette Suche abgebrochen wird, sobald irgendeiner der beiden Primfaktoren gefunden wurde!

Bei Verwendung der `Par`-Monade können Sie Ihr Programm so schreiben, dass die Suche abgebrochen wird, wenn der kleinere Primfaktor gefunden wurde. Sie können es auch so schreiben, dass abgebrochen wird, wenn der größere Primfaktor gefunden wurde. Sie können Ihr Programm mit der `Par`-Monade jedoch nicht so formulieren, dass die Suche abgebrochen wird so bald *einer* der beiden Primfaktoren gefunden wurde (egal ob der kleinere oder größere Primfaktor; der Faktor, dessen Berechnung am schnellsten gelang).

Warum ist dies mit der `Par`-Monade grundsätzlich nicht möglich?

U-Need-To-Work

Bitte unterstützen Sie uns bei der Entwicklung des `UniWorX`-Nachfolgers `Uni2work`!

Laden Sie hierfür Ihre Abgabe zu diesem Übgunsblatt entweder im alten System (`UniWorX`) oder im neuen System (`Uni2work`) hoch — bitte nicht in beiden Systemen abgeben.

Falls Sie sich für `Uni2work` entscheiden, loggen Sie sich dort einfach wie gewohnt mit Ihrem Campus-Account ein und melden sich dann dort erneut zum Kurs ProMo an. Die URL von `Uni2work` lautet (ab 04.07.2018):

`https://uni2work.ifi.lmu.de/`

Als Anreiz das neue System zu benutzen sind die Abgabe-Deadlines für Blätter 10 und 11 auf `Uni2work` um 32h verlängert. Es entstehen Ihnen ansonsten keine Nachteile wenn Sie das alte `UniWorX` für die Abgabe nutzen.

Egal für welches System Sie sich entscheiden: Bitte beantworten Sie möglichst zeitnah im Anschluss an Ihre Abgabe diesen Fragebogen (Dauer: 5 - 10 Minuten):

`https://goo.gl/forms/pW0mGQ2xuIN1f2VC2`

Für das vollständige Ausfüllen des Fragebogens und befolgen der Anweisungen im Fragebogen erhalten Sie 4 Bonuspunkte. Bonuspunkte werden nicht auf das erreichbare Maximum angerechnet, d.h. Bonuspunkte können lediglich bisher nicht erreichte Punkte ausgleichen.

Abgabe: Lösungen zu den Hausaufgaben können bis Samstag, den 14.7.18, mit UniWorX nur als `.zip` abgegeben werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bis zu 3 Studierende können gemeinsam als Gruppe abgeben. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ss-2018/promo/`).