Einführung in die Funktionale Programmierung mit Haskell

Debugging & Profiling

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

17. Juli 2013

Lehr-Evaluation

Sagt mir Eure Meinung zu Dieser Vorlesung!

- Am 17. Juli 2013 um ca. 14:30h sollten alle Teilnehmer eine eMail von `evasys.admin <lehrevaluation@lmu.de>` bekommen haben, welche einen Link auf eine **Online-Evaluation** enthält.
- Der Link enthält eine TAN und kann nur wenige Tage lang verwendet werden!
- Die Bearbeitung ist anonym (TAN kann nicht mehr zugeordnet werden)

Klausur

Termin: Freitag, 19.07.2013 9:00-12:00h
Bearbeitungszeit: 120 Minuten

Ort: B U101, Oettingenstr. 67

- Lichtbildausweis und Studentenausweis mitbringen
- Keine Hilfsmittel!
  - Nur Nahrung & Dokumenten-echte Stifte
  - Kein Papier, keine Mäppchen, keine Elektronik!
- Keine Panik!
- Rechtzeitig vor Beginn (9:00h) da sein

Debugging

“Well-typed programs can’t go wrong!”

R. Milner, 1934-2010

Die Praxis zeigt, dass in der Tat sehr viele Probleme während dem kompilieren abgefangen werden.

Im Umgang mit partiellen Funktionen, oder zur Optimierung der Laufzeit eines Programms, braucht man Gelegentlich aber zusätzliche Hilfe.

Hilfe aus dem Web

Es gibt zahlreiche Webseiten zu Haskell. Ein zentrale Auskunft liefert das Haskell Wiki: http://www.haskell.org/haskellwiki

Es gibt aber auch spezialisierte Suchmaschinen:

Hoogle http://haskell.org/hoogle

- Durchsucht Paket-Dokumentation
- Kann auch nach Typen suchen
  `":: (a -> b -> a) -> a -> [b] -> b"`
  findet all möglichen Varianten von der Faltungsfunktion.
- Kann lokal laufen und in Emacs integriert werden

Hayoo http://holumbus.fh-wedel.de/hayoo/hayoo.html

Besonders gut, um alle möglichen Haskell Pakete zu durchsuchen.

Hilfe aus der Gemeinschaft

Bei speziellen Fragen sind die Haskell-Mailinglisten eine gute Anlaufstelle, um Fragen zu stellen:

- `haskell-cafe@haskell.org`
- `beginners@haskell.org`
- `haskell@haskell.org`

Auch auch Code-Probleme spezialisierte Webseiten können Haskell-Programmierer helfen:
`http://stackoverflow.com/questions/tagged/haskell`

Debug.Trace

Eine sehr primitive Debug-Methode bietet das Modul `Debug.Trace`

```
trace      ::      String -> a -> a
traceShow  :: Show a => a -> b -> b
traceStack :: String -> a -> a
traceIO    :: String -> IO ()
```

- All diese Funktionen haben **Seiteneffekte**:
  Ausgabe nach `stdout`
- Sollten nicht in der Endfassung von Code enthalten sein
  $\Rightarrow$ Fehlermeldung bei GHC Option `-XSafe`
- Ausgabe verändert Auswertereihenfolge Fehler/Termination?!
- Aufgrund von Lazy Evaluation kann die Reihenfolge der Ausgaben verwirrend sein

Fazit: Besser GHCI verwenden!

Debugging mit GHCI

GHCI kennt einen Debugging-Modus

- `:break` setzt Breakpoint in Zeile/Spalte oder für Bezeichner
- `:show breaks` zeigt gesetzte Breakpoints
- `:delete` löscht Breakpoint
- `:set -fbreak-on-exception` hält vor Ausnahmen
- `:trace e` protokollierte Auswertung von `e`
- `:continue` setzt Auswertung fort
- `:step` und `:back` am Breakpoint Schritt vor oder zurück
- `:hist` zeigt letzte Schritte
- `:list` gibt aktuell bearbeiteten Code aus
- `:print` gibt Wert von Bezeichnern aus — Thunks: `_t1`
- `:force` erzwingt Auswertung des Bezeichners — ignoriert BPs

Am Breakpoint ausgewertete Expression ist an `_result` gebunden

Beispiel: Debugging

```
> ghci DebugDemo1
Ok, modules loaded: Main.
*Main> main
* Ein Hanoi-Turm der Höhe 15 genau *** Exception: Prelude.head: empty

*Main> :set -fbreak-on-exception
*Main> :trace main
* Ein Hanoi-Turm der Höhe 15 genau Stopped at <exception thrown>

*Main> :back
Logged breakpoint at DebugDemo1.hs:114:23-29
_result :: [(Int, Int)]
i :: Int
j :: Int

[-1: DebugDemo1.hs:114:23-29] *Main> :list
113  hanoi :: Int -> Int -> Int -> [(Int,Int)]
114  hanoi 1 i j = let r = [(i,j)] in return $ head $ tail r
115  hanoi n i j =
```

Testen mit QuickCheck

Test.QuickCheck ist ein Haskell Modul, welches das Testen von Haskell Funktionen mit Zufallswerten erlaubt.

Man spezifiziert boolsche Bedingungen, üblicherweise QuickCheck-Properties, welche QuickCheck dann versucht zu brechen:

```
Prelude> :m + Test.QuickCheck
> quickCheck (\s -> (reverse.reverse) s == s)
+++ OK, passed 100 tests.
()

> quickCheck (\s -> (reverse.('x':).(drop 1).reverse) s == s)
*** Failed! Falsifiable (after 1 test):
""
()
```

QuickCheck gibt auch das Gegenbeispiel aus!

Testen mit QuickCheck

Üblicherweise definiert man in seinem Programm **QuickCheck-Properties**:

```
prop_numberLength0 x
  | x > 0     = numberLength x <= (fromIntegral x)
  | otherwise = True -- generiert sinnlose Test
```

Vorsicht: (1)

Auch wenn der Name QuickCheck-"Properties" dies suggerieren:

Testen $\neq$ Beweisen

Nur weil QuickCheck keinen Fehler findet, ist der Code noch nicht korrekt, aber QuickCheck ist trotzdem sehr nützlich, um Gegenbeispiele zu finden.

Testen mit QuickCheck

Üblicherweise definiert man in seinem Programm **QuickCheck-Properties**:

```
prop_numberLength1 x
 | x > 0  = label "Good " $ numberLength x <= (fromIntegral x)
 | otherwise = label "Boring" True  -- generiert sinnlose Test

> quickCheck prop_numberLength1
+++ OK, passed 100 tests:
51% Boring
49% Good
```

Vorsicht: (2)

Standard Generatoren für Testfälle sind nicht immer die richtigen.
Im obigen Beispiel sind die Hälfte aller Tests vollkommen nutzlos!

```
prop_numberLength2 (Positive x) =
  numberLength  x <= (fromIntegral x)
```

Besser, aber noch nicht gut genug.

QuickCheck beim Kompilieren

Template Haskell erlaubt Metaprogrammierung:
Haskell Programme erzeugen Haskell Programme.

Eine Anwendung: QuickCheck während dem Kompilieren

```
{-# LANGUAGE TemplateHaskell #-} -- for QuickCheck.All
import Test.QuickCheck
import Test.QuickCheck.All

prop_map (NonNegative n) (NonNegative m) =
  map (+ m) [1..n] == [1+m..n+m]

prop_foldr l = foldr (:) [] l == l
prop_foldl l = foldl (flip (:)) [] l == reverse l

-- | Execute @runTests@ in GHCi upon loading
runTests = $quickCheckAll
```

Profiling

GHC erlaubt Profiling, d.h. zur Laufzeit wird protokollierte, was das Programm wie lange wirklich macht.
Das Programm muss speziell übersetzt werden:

```
> ghc MyProg.hs -O2 -prof -fprof-auto -rtsopts
> ./MyProg +RTS -p
```

erstellt Datei `MyProg.prof`, in der man sehen kann wie viel Zeit bei der Auswertung der einzelnen Funktionen verwendet wurde.

- Genutzte Module müssen mit Profiling-Unterstützung installiert sein `cabal install mein-modul -p`
- Viele Optionen verfügbar.
  Ohne `-fprof-auto` werden z.B. nur Module gezählt.

⇒ `> ./MyProg +RTS -s` für Statistiken reicht oft auch schon

Beispiel: Profiling

```
COST CENTRE    MODULE  %time %alloc

fakultät       Main     56.8   88.8
numberLength   Main      9.9    1.4
collatzLength  Main      9.9    0.9
hanoi          Main      8.6    4.6
collatzStep    Main      7.4    2.1
fibs           Main      4.9    2.0
main0          Main      2.5    0.0
```

COST CENTRE	MODULE	no.	entries	individual %time	individual %alloc	inherited %time	inherited %alloc
MAIN	MAIN	58	0	0.0	0.0	100.0	100.0
main0	Main	118	0	2.5	0.0	2.5	0.0
printTimeDiff	Main	137	1	0.0	0.0	0.0	0.0
printLocalTime	Main	120	0	0.0	0.0	0.0	0.0
CAF	Main	115	0	0.0	0.0	97.5	100.0
fakNumber	Main	135	1	0.0	0.0	0.0	0.0
cseqLength	Main	133	1	0.0	0.0	0.0	0.0
numbersWithCSequenceLength	Main	130	1	0.0	0.1	17.3	3.2
collatzLength	Main	132	173813	9.9	0.9	17.3	3.0
collatzStep	Main	134	171350	7.4	2.1	7.4	2.1
hanoiHeight	Main	125	1	0.0	0.0	0.0	0.0
fibs	Main	124	1	4.9	2.0	4.9	2.0
fibNumber	Main	121	1	0.0	0.0	0.0	0.0
printLocalTime	Main	119	1	0.0	0.0	0.0	0.0
main0	Main	117	1	0.0	0.0	75.3	94.8
printTimeDiff	Main	138	0	0.0	0.0	0.0	0.0
fakultät	Main	136	1	56.8	88.8	56.8	88.8
numberWithCSequenceLength	Main	129	1	0.0	0.0	0.0	0.0
numberWithCSequenceLength.\	Main	131	2463	0.0	0.0	0.0	0.0
main0.r	Main	126	1	0.0	0.0	8.6	4.6
hanoi	Main	127	32767	8.6	4.6	8.6	4.6

Threadscope

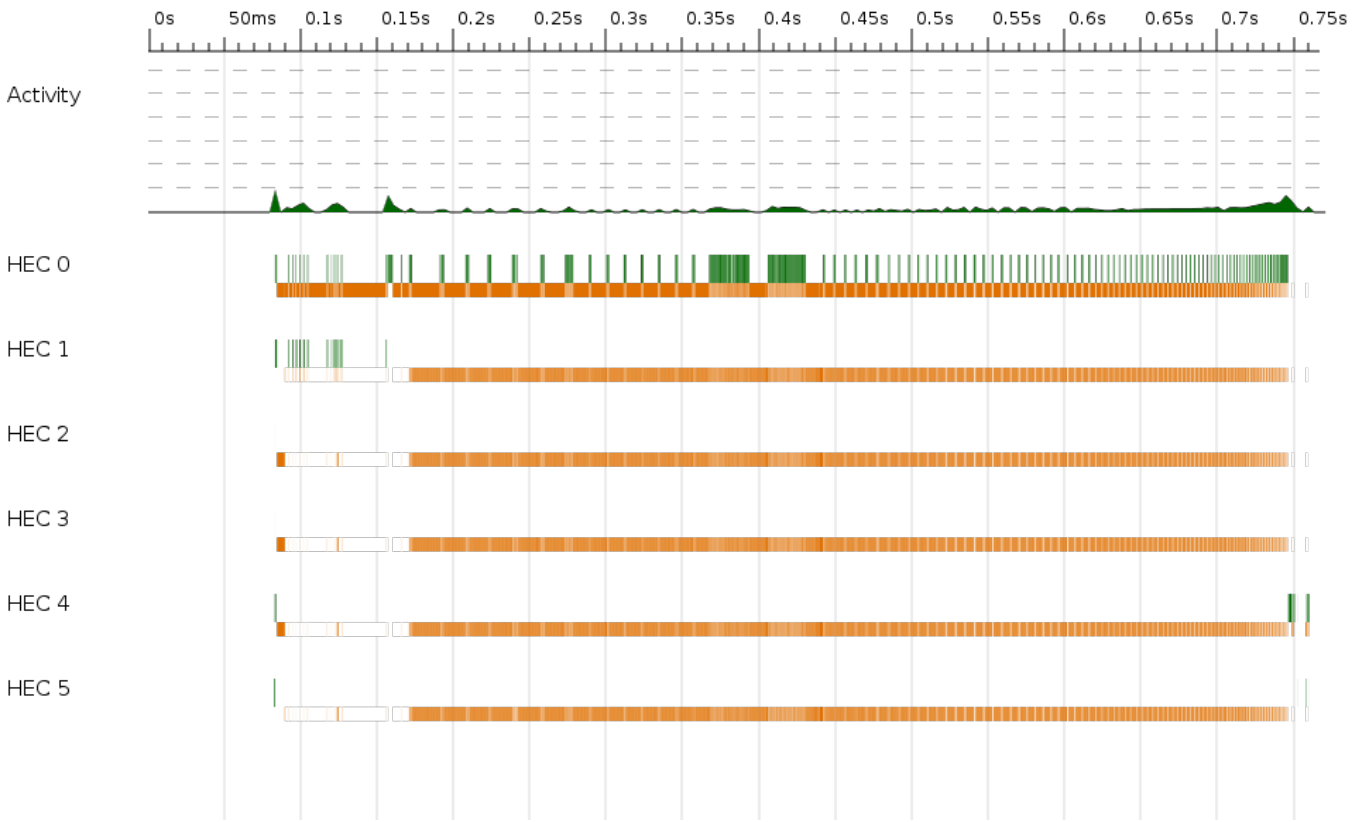
Für das Profiling von parallel ausgeführten Haskell Programmen gibt es den **Threadscope** Profiler; erhältlich mit
`cabal install threadscope`

Verwendung

```
> ghc MyPrg.hs -threaded -eventlog -rtsopts
> ./MyPrg +RTS -N4 -ls
> threadscope MyPrg.eventlog
```

- Zuerst Programm mit Event-Logging zu übersetzen
- Dann mit Event-Logging ausführen
- Threadscope visualisiert dann das Event-Log

Threadscope



Threadscope zeigt uns hier, dass unsere Parallelisierung schlecht ist: Bis auf eine kurze parallele Arbeitsphase am Anfang (grün), waren die Prozessoren eigentlich nur mit GarbageCollection beschäfigt (orange).