Seminar Programmierwerkzeuge
Wintersemester 2006/2007
Debuggen durch Beobachten

Jonas Völcker

4.12.2006
Betreuer: Prof. Dr. Michael Hanus

Zusammenfassung

Die Programmiersprache Haskell ist nicht immer einfach zu Debuggen - daher wurde von Andy Gill im Jahr 2000 einen Debugger namens Hood entworfen, welcher ausschliesslich vielfach implementierte Erweiterungen in Haskell benutzt und einfach als Bibliothek eingebunden wird. Dieser Debugger basiert auf dem Konzept des Beobachtens von Zwischenstrukturen, nicht auf dem in imperativen Sprachen sonst oft verwendeten Paradigma des schrittweisen Ausführens und Untersuchens der Variablen.
Dieser und weitere Ansätze, Haskell zu Debuggen, werden in diesem Paper betrachtet und verglichen.

Inhaltsverzeichnis

1 Einleitung

Debugger erlauben es, ein Programm von innen heraus zu betrachten, während es läuft. Durch Beobachten der erzeugten, manipulierten und wieder zerstörten Datenstrukturen können Informationen darüber gewonnen werden, was das Programm macht und was es nach dem Willen des Programmierers machen sollte. Imperative Debugger zeigen dafür dem Benutzer ansonsten unsichtbare Informationen über die gerade ausgeführten Berechnungen, indem sie dem Benutzer erlauben, das betrachtete Programm Schritt für Schritt auszuführen und dabei den Inhalt von Variablen zu einem bestimmten Ausführungszeitpunkt anzuzeigen.
Dies ist jedoch keine Lösung für eine funktionale Programmiersprache wie HASKELL, da gewisse Ansätze sich nicht von der imperativen in die funktionale Welt übertragen lassen:

- Es gibt keine Variablen, die sich während der Ausführung verändern können.
- Das Konzept der Hintereinanderausführung bestimmter Aktionen oder der Ausführung einer bestimmten Zeile existiert nicht.
- Jedes Berechnungselement hat zwei Bezugselemente: ein statisches (welches die Element und dessen Kontext erzeugt) und ein dynamisches (welches das Element zuerst auswertet). Ein Stack Trace wird so zu einem Parent Tree.
- Wenn eine Funktion aufgerufen wird, sind die Argumente eventuell noch nicht ausgewertet - soll in diesem Fall der Debugger weitere Auswertungen vornehmen?

In diesem Paper soll nun betrachtet werden, welche imperative Debuggingmethoden auf die funktionale Programmierung übertragbar sind und wie die Programmierer des HOOD-Debuggers dies implementieren. Weiterhin soll ein kurzer Überblick gegeben werden, welche anderen Lösungen es gibt, funktionale Programme zu Debuggen und wie der HOOD-Debugger im Vergleich dazu abschneidet.

2 Bisherige Fehlersuche in Haskell

Wollte man bislang die Datenstrukturen sichtbar machen, so bediente man sich einer Funktion namens *trace*. Alle aktuellen HASKELL Implementierungen stellen diese Funktion bereit, mit dem Typ

```
trace :: String -> a -> a
```

Diese Funktion gibt als Seiteneffekt das erste Argument aus und gibt das zweite Argument zurück.

Drei Hauptprobleme bestehen bei der Benutzung von *trace* als Hilfe zur Fehlersuche: Zunächst ist die Ausgabe von *trace* schwer verständlich: Mehrere Instanzen von *trace* können ineinander verschachtelte Ausgaben produzieren und so zusammen mit der Auswertungsreihenfolge der Bedarfsauswertung die Leserlichkeit vollständig zerstören.
Als zweites Problem ergibt sich, dass das Einfügen von *trace* in den HASKELL-Code dazu neigt, die Struktur des Codes zu verändern. Dabei können diese Veränderungen dazu beitragen, das Ergebnis zu verfälschen. Betrachtet man beispielsweise die folgende Variante von `sum`, die ihre eigene Ausführung mittels *trace* ausgibt:

```
tracing_sum xs = trace message res
  where
    res = sum xs
    message = "sum " ++ show xs ++ " = " ++ show res
```

Das Ausführen von `tracing_sum` liefert folgende Ausgabe:

```
Main> tracing_sum [1,2,3]
sum [1,2,3] = 66
Main>
```

Wie man sieht, konnte das Verhalten von `sum` beobachtet werden, jedoch wurden nichttriviale Änderungen am Code notwendig!
Das dritte Problem ist, dass *trace* die Striktheit der Dinge ändert, die es beobachtet, da *trace* hyperstrikt in seinem ersten Argument ist. Dies kann anhand einer neuen Version von `fst` verdeutlicht werden:

```
tracing_fst pair = trace message res
  where
    res = fst pair
    message = "fst " ++ show pair ++ " = " ++ show res
```

Die Ausführung schlägt fehl, da `tracing_fst` zu strikt ist:

```
Main> tracing_fst (99,undefined :: Int)
fst (99,
Program error: {undefined}
Main>
```

3 HOOD - The Haskell Object Observation Debugger [2]

Den Entwicklern von Hood ging es darum, dem Benutzer die Betrachtung von Datenstrukturen zu vereinfachen. Sie argumentieren, dass dem Setzen von Breakpoints und der Auswertung von Variablen die Betrachtung von Zwischenstrukturen, während sie von Funktion zu Funktion gereicht werden, entspricht.
Wenn man sich einmal den Ablauf eines Haskell-Programm ansieht, stellt man schnell fest, dass die benötigten Informationen zwar aus den Zwischenstrukturen und Pipelines besorgen kann, man jedoch mit einer Fülle von Informationen überhäuft wird, die das Auffinden der benötigten Informationen sehr schwierig macht.
Folgendes Programm soll die Situation verdeutlichen:

```
natural :: Int -> [Int]
natural
  = reverse
  . map (`mod` 10)
  . takewhile (/= 0)
  . iterate (`div` 10)
```

Um diese Funktion zu verstehen würde man sie als erstes mit Beispieldaten aufrufen:

```
Main> natural 3408
[3,4,0,8]
```

Daraus kann man ableiten, was die Funktion tut, aber nicht, wie sie arbeitet. Daher werfen wir einen Blick in die Zwischenstrukturen der Ausführung: ($ ist die Notation für den Infix-Operator)

```
natural 3408
-> reverse
 . map (`mod` 10)
 . takeWhile (/= 0)
 . iterate (`div` 10)
 $ 3408
-> reverse
 . map (`mod` 10)
 . takeWhile (/= 0)
 $ (3408 : 340 : 34 : 3 : 0 : _)
-> reverse
 . map (`mod` 10)
 $ (3408 : 340 : 34 : 3 : [])
-> reverse
 $ (8 : 0 : 4 : 3 : [])
-> (3 : 4 : 0 : 8 : [])
```

Wie man sieht, ist es schwierig, die nötigen Daten direkt herauszulesen. Die wichtigen Informationen - die Zwischenstrukturen - sind jedoch klar zusammenzufassen:

```
-- after iterate (`div` 10)
 ( 3408 : 340 : 34 : 3 : _ )
-- after takeWhile (/= 0)
 ( 3408 : 340 : 34 : 3 : [])
-- after map (`mod` 10)
 ( 8 : 0 : 4 : 3 : [])
-- after reverse
 ( 3 : 4 : 0 : 8 : [])
```

Genau dieses Ziel verfolgen die Entwickler des HOOD-Debuggers: Informationen über Datenstrukturen dem Benutzer klar zugänglich machen. Da Datenstrukturen sehr reichhaltig und oft in HASKELL vorkommen, kann dies dem Benutzer ein mächtiges Werkzeug in die Hand geben.

Das von den Entwicklern vorgeschlagene Debuggingsystem ist folgendes:

- Sie stellen eine HASKELL-Bibliothek bereit, welche wiederum Kombinatoren für das Debugging bereitstellt. (damit ist sichergestellt, dass das gesamte Haskell untersucht werden kann.)
- Der frustrierte [sic!] Haskell-Programmierer benutzt diese Kombinatoren in seinem Code und lässt sein Programm erneut laufen.
- Das Haskell-Programm wird normal ausgeführt; durch die debugging-Kombinatoren werden keine Verhaltensänderungen im Programm bewirkt.
- Die durch die Kombinatoren markierten Strukturen werden auf der Benutzerkonsole angezeigt, nachdem das Programm beendet wurde.

3.1 Observe - eine neue Methode zu beobachten

Auch wenn die *trace*-Funktion in einigen Fällen nützlich sein kann, so ist ihre Anwendung doch zu problematisch, um höhere debugging-Funktionen darauf aufzubauen. Wie könnte also ein besserer debugging-Kombinator aussehen? Anhand des Beispiels ist zu erkennen, dass eine transparente Beobachtung der Datenstrukturen von Vorteil wäre.
Um die Idee zu verdeutlichen, betrachte man das folgende HASKELL-Fragment:

```
consumer . producer
```

Was wäre, wenn sich nun die Funktion *id* an ihre Argumente erinnern würde? Man könnte *id*s an strategisch günstige Stellen setzen und nachträglich ansehen, was zwischen `consumer` und `producer` an Daten ausgetauscht wurde.

```
consumer . id . producer
```

Die HOOD-Entwickler argumentieren, dass ein *high-level* Kombinator für die Fehlersuche eben diese Form annehmen sollte: ein Argument sollte transparent für das Programm durchgereicht werden, gleichzeitig jedoch beobachtet und gespeichert werden, um später vom Benutzer betrachtet werden zu können.

Um mehrfache Beobachtungen zu vereinfachen, soll jede Beobachtung mit einem Label versehen werden. Den Typ des Kombinators geben sie wie folgt an:

```
observe :: (Observable a) => String -> a -> a
```

Dies würde im obigen Beispiel folgendermassen aussehen:

```
consumer . observe  "intermediate" . producer
```

Dies hat dieselbe Semantik wie `consumer . producer`, jedoch werden vom `observe` die Daten gesammelt, die von Erzeuger zum Verbraucher gesendet werden, und in eine persistente Struktur gespeichert, damit sie später angezeigt werden können. Gegenüber der Ausführung des HASKELL-Programms verhält sich Observe hingegen genau wie *id*. Weiterhin kann mit `observe` *jeder* Ausdruck beobachtet werden, nicht nur die Zwischenwerte in der Pipeline. Beispiele hierzu in 3.2.

`observe` hat eine Typklassenrestriktion für den beobachteten Typ. Dies stellt jedoch kein größeres Problem dar, da die Programmierer für jeden HASKELL98 Basistyp (Int, Bool, Float usw.) und für viele Container (List, Array, Maybe, Tupel usw.) Instanzen bereitstellen. Auf die spezifischen Details wird später im Abschnitt 3.3 eingegangen.

Vergleicht man nun `observe` mit *trace* und seinen drei obengenanntenn Schwächen, so stellt man fest:

- *trace* produziert eine manchmal unleserliche Ausgabe, da es keine strukturierte Ausgabe besitzt und die Auswertungsreihenfolge in funktionalen Programmen nicht unbedingt festgelegt ist. `observe` dagegen erzeugt seine Ausgabe mit einem Pretty-Printer und gibt nur die gewünschten Beobachtungen aus. Dies ist möglich, da `observe` eine strukturierte Beobachtung von HASKELL-Objekten erlaubt.

- Im Gegensatz zu tiefergreifenden Einsätzen von *trace* muss mit `observe` der Code nur minimal verändert werden.

- Am wichtigsten ist jedoch, dass die Striktheit einer beobachteten Datenstruktur nicht verändert wird, da `observe` keine Auswertung des betrachteten Objekts vornimmt. Die Betrachtung einer unendlichen Liste oder einer Liste mit $\perp$ ist somit zulässig, wie später zu sehen sein wird.

3.2 Beispiele für die Anwendung von Observe

Es folgen nun einige Beispiele an, wie `observe` benutzt werden kann, bevor in Abschnitt 3.3 auf die Implementierung eingegangen wird.

3.2.1 Betrachten einer endlichen Liste

Als erstes Beispiel Betrachte:

```
ex1 :: IO ()
ex1 = print
      ((observe "list" :: Observing [Int]) [0..9])
```

Dies liefert folgende Ausgabe:

```
-- list
  0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : []
```

Es ist also gelungen, eine Datenstruktur zu beobachten, ohne das Ergebnis oder die Semantik dieses HASKELL-Programms zu verändern. Die Autoren von HOOD benutzen hier eine Typangabe, um klarzustellen, welcher beobachtete Typ gemeint ist.

```
type Observing a = a -> a
```

Dies ist jedoch optional und wird dem Benutzer freigestellt. Genauso hätte das Programm folgendermassen aussehen können:

```
ex1 :: IO ()
ex1 = print (observe "list" [0..9])
```

Diese Definition verlässt sich allerdings darauf, dass per default Int oder eine integer Liste ausgewählt wird.

Typischerweise ist das `observe` durch seinen Kontext vollständig bestimmt, in einigen Beispielen wird trotzdem eine explizite Typangabe vorgenommen, um dem Leser zu verdeutlichen worum es geht.

3.2.2 Betrachten einer intermediären Liste

`observe` kann ebenfalls benutzt werden, um Zwischenstrukturen in der Pipeline zu betrachten:

```
ex2 :: IO ()
ex2 = print
    . reverse
    . (observe "intermediate" :: Observing [Int])
    . reverse
    $ [0..9]
```

Dieses `observe` beobachtet das folgende:

```
-- intermediate
  9 : 8 : 7 : 6 : 5 : 4 : 3 : 2 : 1 : 0 : []
```

3.2.3 Betrachten einer unendlichen Liste

Die beiden bisher betrachteten Listen waren endlich. Betrachten wir nun eine unendliche Liste:

```
ex3 :: IO ()
ex3 = print
      (take 10
           (observe "infinite list" ([0..] :: [Int]))
       )
```

Hier wird eine unendliche Liste beobachtet, welche mit 0 anfängt. Es werden dann die ersten 10 Elemente genommen und sie wird ausgegeben. Wird dieses Beispielprogramm ausgeführt, so erhält man:

```
-- infinite list
  0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : _
```

0 bis 9 wurden anscheinend ausgewertet, der Rumpf des 10. *cons* jedoch nicht. Dieser wurde in der Notation _ ausgegeben. Würde mehr von der Liste ausgegeben, so würde man mehr *cons* Zellen usw. sehen.

3.2.4 Betrachten von Listen mit nicht ausgewerteten Elementen

Was also passiert mit nicht ausgewerteten Elementen in der Liste? Was passiert zum Beispiel bei der Frage nach der Länge einer endlichen Liste?

```
ex4 :: IO ()
ex4 = print
      (length
           (observe "finite list" ([1..10] :: [Int]))
      )

-- finite list
  _ : _ : _ : _ : _ : _ : _ : _ : _ : _ : []
```

Und was, wenn die Elemente $\perp$ sind?

```
ex5 :: IO ()
ex5 = print
      (length
       ((observe "finite list" :: Observing [()])
           [ error "oops!" | _ <- ([0..9] :: [Int])]
        )
       )
```

Dies liefert die selbe Ausgabe wie das Beispiel 4.

Da die Elemente nie ausgewertet werden, macht es keinen Unterschied, ob sie definiert sind oder nicht. Um sie zu beobachten müssen die Elemente jedoch einen nicht-polymorphen Typ zugewiesen bekommen.

Was ist, wenn nur ausgewählte Elemente beobachtet werden?

```
ex6 :: IO ()
ex6 = let xs = observe "list" ([0..9] :: [Int])
      in print (xs !! 2 + xs !! 4)
```

Dies liefert:

```
-- list
  _ : _ : 2 : _ : 4 : _
```

`observe` kann also sowohl zur Beobachtung von Daten in Zwischenstrukturen benutzt werden, als auch um die Datenstrukturen selber zu sehen - zum Beispiel um zu sehen, wie weit die die Bedarfsauswertung die Struktur ausgewertet hat. Dies alles passiert, *ohne die Auswertungsreihenfolge zu verändern*! Dies ist die Stärke von `observe`.

3.2.5 Verwendung mehrerer Observes

Ein Programm kann mehrere Instanzen von Observe enthalten. Zum Beispiel kann das Programm aus der Einführung wie folgt umgeschrieben werden:

```
natural :: Int -> [Int]
natural
  = (observe "after reverse"                :: Observing [Int])
  .reverse
  .(observe "after map (`mod` 10)"          :: Observing [Int])
  . map (`mod` 10)
  .(observe "after takeWhile (/= 0)"        :: Observing [Int])
  . takewhile (/= 0)
  .(observe "after iterate (`div` 10)"      :: Observing [Int])
  . iterate (`div` 10)
```

Wird dies nun mit den Beispieldaten aufgerufen, ergibt sich:

```
-- after iterate (`div` 10)
 ( 3408 : 340 : 34 : 3 : _ )
-- after takeWhile (/= 0)
 ( 3408 : 340 : 34 : 3 : [])
-- after map (`mod` 10)
 ( 8 : 0 : 4 : 3 : [])
-- after reverse
 ( 3 : 4 : 0 : 8 : [])
```

Dies ist exakt das Ergebnis, welches wir in der Einleitung erreichen wollten.

3.2.6 Beobachten von Funktionen

Da sowohl die Basistypen als auch Container beobachtet werden können, ist es ebenfalls möglich, Funktionen zu beobachten. Die Autoren argumentieren dabei damit, dass das Beobachten einer Funktion ein finites Abbilden von (beobachtbaren) Argumenten zu (beobachtbaren) Ergebnissen ist. Daher betrachten sie Funktionen ausschliesslich als Argument-Ergebnis-Paare, eins für jeden Funktionsaufruf. Dabei werden die Funktionen nur in der Weise abgebildet, in der sie benutzt

werden - die Argumente beziehungsweise Ergebnisse können dabei noch nicht ausgewertete Terme enthalten.

Was bedeutet dies in der praktischen Anwendung? Folgendes Beispiel soll dies erläutern:

```
ex7 = print
  ((observe "length" :: Observing ([Int] -> Int))
      length [1..3]
  )
```

Dies führt zu folgender Beobachtung:

```
-- length
  { \ (_ : _ : _ : []) -> 3
  }
```

Einige Dinge fallen hierbei auf:

- `observe` bekommt nun hintereinander drei Argumente: Zunächst das Label, dann wird `observe` auf die beobachtete Einheit (die *length*-Funktion) angewandt und diese auf das Argument für *length*. Es sei noch einmal daran erinnert, dass `observe <label>` eine Art *id* ist und *id* ausschliesslich sein Argument zurückgibt. Die Auswirkungen auf das Haskell-Programm können somit durch simples Umschreiben erklärt werden:

  ```
  (observe "length" :: Observing ([Int] -> Int))
       length [1..3]
  -- entfernen der Typklassifizierung
  = observe "length" length [1..3]
  -- benutze id anstelle observe
  = id length [1..3]
  -- id hat nur ein Argument
  = (id length) [1..3]
  -- und dies wird einfach zurückgegeben
  = (length) [1..3]
  ```

  Diese Schlussfolgerung funktioniert auch für weitere Argumente. `observe` kann somit Funktionen beobachten, die mehr als ein Argument haben.

- Anstelle Funktionen als Menge von Paaren darzustellen, wird eine eher Haskell-artige Darstellung verwendet.

- Die *length*-Funktion schaut nicht alle Teile ihres Arguments an, speziell die Teile der Liste. Diese sind nicht notwendig für ihre Ausführung und werden auch nicht von `observe` dargestellt. Selbst wenn jemand anderes diese Liste ausgewertet hätte, würde man es nicht in der Ausgabe von diesem `observe` sehen, da nur im Kontext dieser *length*-Anwendung beobachtet wird.

Die Beobachtung von Funktionen ist vielseitig und mächtig. Wenn das `observe` auf der aufrufenden Seite steht, kann man die Auswirkungen einer bestimmten Funktion in diesem Kontext sehen, eingeschlossen Funktionen höherer Ordnung.

```
ex8 = print
  ((observe "foldl (+) 0 [1..4]"
    :: Observing ((Int -> Int -> Int)
                  -> Int -> [Int] -> Int)
   ) foldl 0 [1..4]
  )

-- foldl (+) 0 [1..4]
  { \ { \ 6 4 -> 10
      , \ 3 3 -> 6
      , \ 2 1 -> 3
      , \ 0 1 -> 1
      }
      0
      (1 : 2 : 3 : 4 : [])
      -> 10
  }
```

Bei der Beobachtung von `foldl` konnte man ebenfalls dessen Argumente beobachten, einschliesslich eines funktionalen Arguments. Man sieht in diesem Beispiel deutlich, wie eine Funktion höherer Ordnung benutzt wurde.

Sehr nützlich ist die Beobachtung von Funktionen bei der Betrachtung von Pipelines. Bei dem Ausgangsbeispiel (`natural`) können so die einzelnen Transformationen angesehen werden, anstelle der Zwischenstrukturen:

```
natural :: Int -> [Int]
natural
  = (observe "reverse"            reverse
  . (observe "map (‘mod‘ 10)"     map (‘mod‘ 10)
  . (observe "takeWhile (/= 0)"   takewhile (/= 0)
  . (observe "iterate (‘div‘ 10)" iterate (‘div‘ 10)
```

Wie man sieht, sind hier keine `<.>` zwischen den `observe`s und dem Originalcode. Bespielhaft soll hier die Ausgabe von *iterate*... und *takeWhile*... angegeben werden:

```
-- iterate (‘div‘ 10)
  { \ { \ 3 -> 0
      , \ 34 -> 3
      , \ 340 -> 34
      , \ 3408 -> 340
      } 3408
       -> 3408 : 340 : 34 : 3 : 0 : _
  }
```

```
--takeWhile (/=0)
  { \ { \ 0 -> False
      , \ 3 -> True
      , \ 34 -> True
      , \ 340 -> True
      , \ 3408 -> True
      } (3408 : 340 : 34 : 3 : 0 : _)
       -> 3408 : 340 : 34 : 3 : []
  }
```

Daran kann man deutlich sehen, was die Tranformatoren getan haben. `iterate` bekam einen Integer und produzierte einen Stream von absteigenden Zahlen, wovon die ersten 5 ausgewertet wurden. Zudem sieht man, wie das funktionale Argument zu `iterate` benutzt wurde, um aus einer unendlichen Liste eine endliche zu machen.

3.2.7 Erweiterte Verwendung von Observe

In seinem Paper[2] stellt der Autor von HOOD weitere Möglichkeiten vor, wie HOOD benutzt werden kann. Zum Beispiel ist es möglich, die State und IO Monaden mittels `observe` anzuschauen. Der interessierte Leser wird hier an das genannte Paper verwiesen.

3.3 Wie Observe funktioniert

HOOD wurde in diesem Paper als nützliches debugging-Tool vorgestellt, jedoch fehlt bislang die Betrachtung der Implementierung. Im Folgenden soll dargestellt werden, wie die Entwickler HOOD portabel implementiert haben.

Sei folgendes HASKELL-Fragment gegeben:

```
ex12 = let pair = (Just 1, Nothing)
       in print (fst pair)
```

Welche Ausführungsschritte hat ex12 hinter sich? Alle Ausdrücke starten als nicht ausgewertete *Thunks* (also verzögerte Berechnungen).

```
...pair = <thunk> -- start
```

Zunächst mal ist `print` hyper-strikt in seinem Argument, daher startet es die Auswertung von (fst pair). Dadurch wird pair über fst ausgewertet, was ein Tupel zurückgibt, welches wiederum zwei Thunks beinhaltet:

```
...pair = (<thunk>, <thunk>)      -- after step1
```

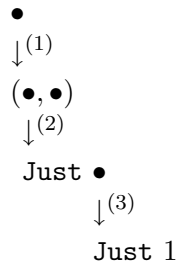
Nun gibt die Funktion fst das erste Element dieses Tupels zurück, welches von print ausgewertet wird:

```
...pair = (Just <thunk>, <thunk>) -- after step2
```

Schliesslich wird der Thunk im `Just`-Konstruktor ausgewertet, mit dem Ergebnis:

```
...pair = (Just 1, <thunk>)       -- after step3
```

Diese Auswertung kann in einem Diagramm dargestellt werden, wobei die drei Schritte aufgezeigt werden sollen, die diese Datenstruktur durchläuft:



Anhand dessen kann das Prinzip hinter `observe` erläutert werden:

- Anstelle der beschrifteten Pfeile werden automatisch Seiteneffekt-Funktionen eingesetzt, welche nicht nur das korrekte Ergebnis der Auswertung zur schwachen Normalform liefern, sondern zusätzlich einen Agenten informieren, dass eine Reduktion stattgefunden hat. Alle Thunks (auch interne) werden demnach durch Funktionen ersetzt, die beim Aufruf den gewünschten Seiteneffekt ausführen.
- Hierzu wird der Typklassenmechanismus benutzt, um eine systemweite (in Laufzeit) Ersetzung der Funktionen zu gewährleisten.

3.3.1 Übermitteln der Form der Datenstruktur

Um eine lokale Kopie der beobachteten Datenstruktur herzustellen, müssen einige Informationen gesammelt werden:

- Welche Auswertung hat stattgefunden? (An welchem Ort?)
- Zu was wurde die Auswertung reduziert? ((:), 3, Nothing, usw.)

In unserem Beispiel würden folgende Informationen von der Seiteneffektfunktion übermittelt:

Name	Location	Constructor
`<>`	root	tuple constructor with two children
`<1>`	first thunk inside root	The Just constructor with one child
`<1,1>`	first thunk inside first thunk inside root	The integer 1

Dies reicht aus, um die beobachtete Struktur wiederzugeben. Zunächst wird nur der unausgewertete Thunk dargestellt:

$$\bullet^{root}$$

Daraufhin wird der erste Schritt, `<>` dargestellt:

$$(\bullet^{<1>}, \bullet^{<2>})$$

Hierbei bezeichnet `<1>` den ersten Thunk im Konstruktor, welcher im ersten Schritt produziert wurde, `<2>` den zweiten Thunk aus dem selben Reduktionsschritt.
Nun wird `<1>` akzeptiert und dargestellt, woraus

$$(\texttt{Just}\bullet^{<1.1>}, \bullet^{<2>})$$

entsteht. Dabei repräsentiert `<1.1>` den ersten (und einzigen) Thunk im vom Thunk `<1>` erzeugten Konstruktor. Nun können wir schliesslich den letzten Thunk bearbeiten. Die Informationen über `<1.1>` liefern:

$$(\texttt{Just1}, \bullet^{<2>})$$

Standardmäßig wissen wir nichts über einen Thunk, er ist unausgewertet wie `<2>`.
Im weiteren wird betrachtet, wie die Nachrichtenübermittlung sich in die Datenstrukturen einbauen lässt.

3.3.2 Beobachtungen der Zwischenstrukturen einbauen

Um den Agenten über stattfindende Reduktionen zu informieren und weitere Observer auf allen neuen unter-Thunks aufzurufen, wird eine Arbeiterfunktion benutzt. Ein möglicher Typ so einer Funktion wäre:

```
observer
  :: (Observable a) =>[Int] -> String -> a -> a
```

Dabei wird das [Int] benötigt, um den Pfad zur Wurzel anzugeben, wie im obigen Beispiel geschehen. `observe` kann durch diese Funktion dargestellt werden:

```
observe = observer []
```

Betrachte nun einen generischen Fall, über einen Pseudokonstruktor, für `observer`. Dies ist gleichzeitig eine informelle Semantik für `observe`.

```
data Cons = Cons ty1  ... tyn

observer path label (Cons v1  ... vn
  = unsafePerformIO
    { send "Cons" path label
    ; return (
        let y1  = observer (1:path) label vn
          ...
            yn  = observer (n:path) label vn
        in Cons y1  ... yn)
    }
```

Aus diesem Pseudocode kann eine Reihe von Dingen ableiten:

- `observer` ist strikt in seinem Konstruktor-Argument. Dies ist kein Widerspruch zur Behauptung, dass `observe` die Striktheit der beobachteten Funktionen nicht verändert, genau wie

  ```
  forall xs :: [a] . foldr (:) [] xs = xs
  ```

  Damit `observer` sein eigenes Konstruktor-Argument anschauen kann, muss er ebenfalls in WHNF ausgewertet werden.

- Die einzige Möglichkeit, dass `observe` festlaufen kann (also zu $\perp$ ausgewertet wird), ist bei dem Aufruf von `send`. Es wird (begründet) angenommen, dass dies nicht blockiert oder fehlschlägt.

- Der Pfad wird in einer strikten Weise benutzt (Dabei wird angenommen, dass `send` strikt ist).

- `observe` *kann* den Speicherverbrauch der betrachteten Programme ändern, da bei seiner Anwendung jegliches Sharing ausgeschlossen wird.

Wenn also angenommen wird, dass der Pfad ein konstanter String ist und das `send` nicht fehlschlägt, kann einfach über die Gleichung gezeigt werden, dass:

```
forall (cons :: Cons) . cons = observe "lab" cons
```

für jedes `cons` der obigen Form.

- Strikte Felder re-triggern nur Auswertungen von schon ausgewerteten Dingen.

- Die Basistypen (Int, Integer, usw.) können als große abzählbare Typen angenommen werden und mit der Annahme über Konstruktoren im Allgemeinen von oben abgedeckt werden.

Funktionen werden von einer anderen Instanz erfasst:

```
observer path label fn arg
 = unsafePerformIO $ do
  {send "->" path label
  ; return (
    let arg' = observer (1:path) label arg
        res' = observer (2:path) label (fn arg')
    in res')
  }
```

Dies ist eine Vereinfachung (da `observer` für jeden Funktionsaufruf eine eindeutige Referenz generieren muss), aber es zeigt das Verhalten insoweit die Ausführung in Haskell betroffen ist. Wiederum wird die oben verwendete Beweisführung benutzt, um zu zeigen, dass:

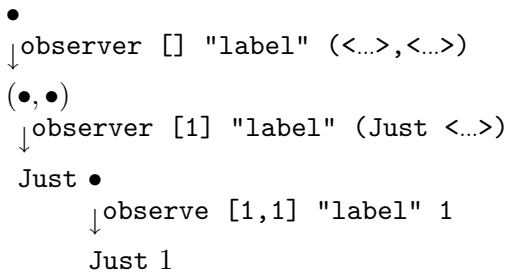
```
forall fn arg . fn arg = observe "lab" fn arg
```

3.3.3 Die Klasse Observable

Es wird der Typklassenmechanismus benutzt, um die wiederholten Aufrufe der Arbeiterfunktion `observer` zu implementieren. Hauptakteur ist die Klasse Observable, für die es zu jedem beobachtbaren HASKELL-Objekt eine Instanz gibt.

```
class Observable a where
  observer :: a -> ObserveContext -> a
```

Im Diagram von oben werden drei Aufrufe an `observer` geschickt:



Der erste Aufruf benutzt eine 2-Tupel Instanz von `Observable`, der zweite eine *maybe*-Instanz und der dritte eine Int-Instanz. Jedem Aufruf wird zudem ein bestimmter Kontext zugewiesen, welcher Informationen enthält, wo sich der Thunk in Bezug auf seine Parent-Node befindet.

In ihrer Implementierung benutzen die Entwickler den Kombinator *send* um die geläufige Ausdrucksweise beim schreiben von `Observer`-Instanzen zu berücksichtigen. Die `Observable`-Instanz für 2-Tupel ist:

```
instance (Observable a, Observable b)
          => Observable (a,b) where
     observer (a,b) = send ","
                           (return (,) << a <<b)
```

Wenn `observer` zu einem 2-Tupel Typ aufgerufen wird, verschickt er ein Paket mit Informationen in dem steht, dass er ein Tupel gefunden hat. Zudem erstellt er zwei neue Thunks die die Komponenten des Tupels werden. Der Typ von *send* wird mit

```
send :: String
     -> MonadObserver a
     -> Parent
     -> a
```

angegeben. `MonadObserver` ist eine lazy state Monade, welche sowohl die Anzahl der sub-Thunks dieses Konstruktors zählt als auch einen eindeutigen Kontext für diese erstellt. Parent ist einfach der Name dieses Kontexts.

3.3.4 Das Tool

Diese Ideen wurden im Haskell Object Observation Debugger (HOOD), einem kompletten HASKELL Debugging-Tool, implementiert. Zum Abschluss dieses Kapitels folgt noch eine kurzer Überblick. Weitere Betrachtungen werden bei dem Vergleich verschiedener Debugger (4) angestellt. Kurz gesagt, wird HOOD wie folgt benutzt:

- Der Benutzer ist dafür veratwortlich, die HOOD-Bibliotheken in seinem Programm zu importieren, welches mehrere Debugging-Funktionen wie `observe` bereitstellt, und entsprechende Beobachter in seinen Code einzufügen.
- Die Benutzung der Observe-Bibliothek liefert einen internen Trace dessen, was während des Programmablaufs beobachtete wurde.
- Sobald der Code ausgeführt wurde, sorgt Code in der Observe-Bibliothek dafür, dass dieser Trace ausgewertet wird und die Strukturen dem Benutzer angezeigt werden.

4 HOOD im Vergleich mit anderen Debuggern für Haskell [1]

Zum tracing von lazy funktionalen Programmen wurde seit Mitte der 80er Jahre viel geforscht. Hier sollen tracing Systeme verglichen werden, die zumindest drei Kriterien erfüllen:
Sie sollen

- einen Großteil der existierenden standard lazy funktionalen Sprachen unterstützen (also Haskell 98).
- öffentlich verfügbar sein.
- weiterhin entwickelt werden.

Dies wird von den Debuggern FREJA, HAT und HOOD erfüllt. Diese Debugger arbeiten im einzelnen wie folgt: FREJA erzeugt einen *Evaluation-Dependence-Tree* als Trace, HAT gibt einen Trace aus, der die Abhängigkeiten zwischen den Redexes (meistens Anwendungen von Funktionen), welche von den Berechnungen reduziert worden. HOOD erlaubt es, Datenstrukturen an vom Programmierer vorgegebenen Punkten im Programm zu betrachten, wie es in imperativen Programmen mit dem *print* -Befehl geschehen kann, jedoch wird dabei nicht die lazy Auswertungsreihenfolge beeinträchtigt und es können ebenfalls Funktionen betrachtet werden.

Debugger sind interaktiv gesteuerte Programme, keine vorgegebenen Rechenabläufe - daher soll hier auf die Nützlichkeit dieser Systeme für den Programmierer eingegangen werden. Laufzeit und Speicherverbrauch sind zwar nicht unwichtig, werden aber gegenüber der Nützlichkeit zurückgestellt und in anderen Papern betrachtet. Weiter soll es nicht darum gehen, einen Gewinner nach Punkten zu küren, sondern Vor- und Nachteile der verschiedenen Systeme zu beleuchten und einen Einblick in die Gestaltung von Debuggern zu geben.

4.1 Ein Überblick

Um dem Leser eine Idee zu geben, wie die einzelnen Debugger funktionieren, soll hier kurz die Arbeitsweise der drei Programme vorgestellt werden. Dies geschieht anhand des folgenden Beispiels:

```
main = let xs = [4*2, 3+6] :: [Int]
       in (head xs, last xs)

head (x:xs) = x

last (x:xs) = last xs
last [x]    = x
```

4.1.1 Freja

Freja ist ein Debugger für eine Teilmenge von Haskell98. Das Debugging besteht darin, dem User eine Abfolge von Fragen zu stellen. Jede Frage steht für die Reduktion eines Redexes, also der Anwendung einer Funktion, zu einem Wert. Der Benutzer antwortet mit *ja*, wenn die Berechnung korrekt erscheint, sonst *nein*. Am Schluss gibt Freja aus, welche Reduktion zu der falschen Aussage geführt hat, beziehungsweise welche Funktionsdefinition falsch ist.

Zuerst wird abgefragt, ob die Funktion `main` das richtige Ergebnis ausgibt. Wenn eine Frage zu einer Reduktion mit *nein* beantwortet wird, so wird die nächste Frage die Auswertung der rechten Seite der Funktionsdefinition betreffen. Dies ist eine Ähnlichkeit mit konventionellen Debuggern, da die Eingabe von *nein* bedeutet, dass der Debugger in den betrachteten Funktionsaufruf hineinschauen soll, während die Eingabe *ja* bedeutet, dass der Debugger mit dem nächsten Funktionsaufruf weitermachen soll. Wenn die Reduktion eines Funktionsaufrufs inkorrekt ist, aber alle Reduktionen für die Auswertung der rechten Seite dieser Funktion korrekt sind, dann muss die Funktion falsch definiert sein für die übergebenen Argumente.

Als Beispiel sei hier eine Debugging-Sitzung mit Freja dargestellt. Wie oben angemerkt wird das Beispielprogramm verwendet. Das Symbol $\perp$ steht für `bottom` und `?` für einen nicht ausgewerteten Ausdruck.

`main` $\Rightarrow (8, \perp)$	*no*
`4*2` $\Rightarrow 8$	*yes*
`head [8,?]` $\Rightarrow 8$	*yes*
`last [8,?]` $\Rightarrow \perp$	*no*
`last [?]` $\Rightarrow \perp$	*no*
`last []` $\Rightarrow \perp$	*yes*

`Bug located! Erroneous reduction; last [?]` $\Rightarrow \perp$

4.1.2 Hat

Hat besteht aus einer modifizierten Version des nhc98 Haskell Compilers und einem seperaten Browser. Ein kompiliertes Programm wird normal ausgeführt, jedoch wird neben der normalen Berechnung ein Redex-Kette auf dem Heap aufgebaut. Anstelle zu terminieren wartet das Programm am Ende der Berechnungen darauf, dass der Browser sich mit ihm verbindet. Der Benutzer kann im Browser die Ausgaben des Programms angucken und dann für eine bestimmte Ausgabe den *Parent-Redex* anfordern. Der Parent-Redex ist derjenige Ausdruck, durch dessen Reduktion den abgefragten Ausdruck erschaffen hat. Jeder Teil dieses Redexes hat wiederum einen Parent-Redex, bis hin zur Funktion `main`, welche keine Vorgänger hat. Bei dieser Debugging-Methode geht der Anwender von einer falschen Ausgabe oder einer Fehlermeldung zurück, bis der Fehler gefunden wurde.

Grundsätzlich wird wie folgt beim Debuggen mit Hat vorgegangen: Das Programm bricht mit einer Fehlermeldung ab und der Browser springt direkt zum Parent-Redex: `last []`. Dieser Aufruf von `last` mit einer leeren Liste sollte garnicht vorkommen, daher lässt sich der User den Parent-Redex von `last []` anzeigen. Der Browser zeigt daraufhin `last (3+6:[])` an, wodurch klar ist, dass `last` für einelementige Listen nicht korrekt definiert ist. Im Browser wird die

Redex-Kette wie folgt angezeigt. Um zu zeigen, wie der Vorgänger von einem Unterausdruck dargestellt wird (`4*2` ist der Vorgänger von `8`), ist hier mehr zu sehen, als zur Fehlerfindung nötig.

```
last []
last (3+6:[])
last (8:3+6:[])
∇4*2
main
```

Der Browser kann zusätzlich zeigen, wo im Quelltext `last` mit dem Argument `[]` aufgerufen wurde.

4.1.3 Hood

Hood ist, wie oben geschrieben, eine Haskell-Library. Der Benutzer versieht sein Programm an den zu beobachtenden Stellen mit `observe`, welches in der Library definiert ist. Während das Programm läuft, werden die Aktionen an den beobachteten Stellen aufgezeichnet. Danach werden sie in benutzerfreundlicher Form ausgegeben.

Im Beispiel soll nun das Argument von `last` beobachtet werden:

```
main = let xs = [4*2, 3+6]
       in (head xs, last (observe "last arg" xs))
```

Wenn das abgeänderte Programm terminiert, gibt Hood uns das folgende aus:

```
-- last arg
  _ : _ : []
```

(_ ist ein unausgewerteter Ausdruck, s.o.)
Zu sehen ist, dass das erste Element der Liste nicht ausgewertet wird, da zwar das Programm es auswertet, aber nicht `last`.

Um mehr Einblick in das Geschehen zu bekommen, wird nun die Funktion `last` inklusive ihrer rekursiven Aufrufe betrachtet:

```
last = observe "last fun" last'

last' (x:xs) = last xs
last' [x] = x
```

Dies gibt den Wert der Funktion als Zuordnung der Argumente zu ihren Ergebnissen:

```
-- last
  { \ (_ : _ : []) -> throw <Exception>
  , \ (_ : []) -> throw <Exception>
  , \ [] -> throw <Exception>
  }
```

Nun ist klar, dass `last` mit einer leeren Liste aufgerufen wird. Der Benutzer muss nun daraus schliessen, dass der Aufruf mit einer einelementigen Liste dazu geführt hat, dass der fehlerhafte Funktionsaufruf stattfindet, was genaugenommen nicht von der Information, die Hood ausgibt impliziert wird.

4.2 Vergleich der Prinzipien

Zunächst scheint es so, als haben die drei Systeme nicht viel gemein, ausser dem Ziel, dem Benutzer beim Debugging zu helfen. Dennoch bedienen sich alle drei eines zweistufigen Ansatzes: Zunächst werden Informationen gesammelt, während das eigentliche Programm ausgeführt wird. Dann werden die gesammelten Informationen dem Benutzer zugänglich gemacht, damit dieser den Fehler finden kann. Jedes der drei Programme stellt dem Benutzer die Informationen in einer Art Browser dar; in Freja ist dies der Teil, wo dem Benutzer Fragen gestellt werden, in Hat ist es der Browser, der die Vorgänger anzeigen kann und in Hood ist es der Teil, der die Beobachtungen auf dem Schirm ausgibt.

Alle drei Systeme sind geeignet, Programme zu debuggen, welche einen oder mehrere der folgenden beobachtbaren Fehler haben: falsche Ausgabe, Abbruch mit Fehlermeldung oder Nichttermination. Bei letzterem kann das Programm abgebrochen werden und danach der Browser verwendet werden.

4.3 Evaluation der Systeme

Die Unterschiede der Systeme werfen verschiedene Fragen auf. Ist es wünschenswert, ein Merkmal eines System in ein anderes zu integrieren? Würde eine alternative Entwicklung Sinn machen? Inwieweit ist ein Merkmal eines bestimmten Systems inherent mit ihm verbunden, möglicherweise durch besonderheiten in seiner Implementierung oder der Tracingmethode? Da die Möglichkeiten, einen Tracer zu implementieren sehr vielfältig sind, ist es sinnvoll, die Evaluation früh in der Praxis stattfinden zu lassen. Im Paper von O.Chitil et al.[1] wenden die Autoren diese drei Systeme auf eine Vielzahl von Programmen an, in die sie absichtlich Fehler eingebaut haben. Dabei wurden die oben genannten Fehler berücksichtigt. Die nachfolgenden Abschnitte beziehen sich auf ihr Paper.

In ihren Experimenten gingen sie nach dem folgenden Protokoll vor: Mindestens zwei Programmierer waren involviert. Zunächst erklärt der Autor eines korrekt arbeitenden Programms, wie dieses Funktioniert. Daraufhin werden von einem anderen Programmierer Fehler in den Code eingebaut, die der Compiler nicht erkennt. Der erste Programmierer muss nun alle Fehler in dem veränderten Quelltext mit Hilfe eines Debuggers finden und beheben. Dabei sind sie angehalten, laut mitzudenken und Notizen zu machen.

Teilnehmer waren ausschliesslich erfahrene Haskell-Programmierer.

Die benutzten Programme in diesem Versuch waren von mittlerer Komplexität. Das größte Programm, PsaCompiler, ein Compiler für eine Spielzeug-Programmiersprache, besteht aus 900 Zeilen Code in 13 Modulen und berechnet 20.000 Reduktionen für die Eingabe, welche ihm gegeben wurde. Das am längsten laufende Programm, Adjoxo, ein Schiedsrichter für TicTacToe (auch Kreuze und Kreise genannt), besteht nur aus 100 Zeilen Code. Dafür berechnet es bis zu 830.000 Reduktionen für den gegebenen Input. Dieser Versuch war eingeschränkt durch die Teilmenge von Haskell98, die Freja unterstüzt. Daher konnten nur solche Programme teilnehmen, die z.B. keine Klassen verwendeten. Zudem ist nicht jedes Freja-Programm ein gültiges Haskell-Programm. Weiterhin konnten keine Programme verwendet werden, die monadischen Input/Output benutzen, da Freja dies nicht unterstützt und Hat nur wenige Operationen zulässt. Die verwendeten Programme sind jedoch durchaus keine abstrakten Beispiele, sondern Programme, wie sie oft im wirklichen Leben vorkommen.

4.3.1 Lesbarkeit

Die Ausgaben der Debugger sind auch bei großen Programmen durchaus noch lesbar und nicht zu groß. Obwohl der Benutzer nicht nur den Trace, sondern auch das Programm vor sich hat, hat sich gezeigt, dass informative Funktionsnamen den Bedarf, in den Quellcode zu gucken, deutlich reduziert und dadurch den Debugging-Vorgang beschleunigt. Die Autoren loben die Verkürzung von nicht ausgewerteten Ausdrücken zu `?` bzw. undefinierten Werten als $\perp$ in Freja und Hood. Dies erhöhe die Lesbarkeit der Traces. In seltenen Fällen wünschen sie sich jedoch, bei Bedarf diese Symbole wieder in Langform ansehen zu können.

Bei der Darstellung von Funktionen, welchen in Haskell natürlich eine besondere Bedeutung zukommt, wird angemerkt, dass die Darstellung in Hood vom Haskell-Standard abweiche und man einige Zeit brauche, um sich an sie zu gewöhnen. Dafür sei diese eher abstrakte Darstellung geeignet, über die Korrektheit des Programms zu entscheiden.
In Freja und Hat werden Funktionen entweder als teilweise Anwendung einer Funktion oder als λ-Abstraktionen dargestellt, wobei ersteres der Lesbarkeit zugute kommt. Die genannten Programme können auf Wunsch diese Abstraktionen vollständig anzeigen, was jedoch das Lesen sehr erschwert.

4.3.2 Lokalisierung eines Fehlers

Mit allen drei Systemen ist es den Autoren gelungen, jeden Fehler in den benutzten Programmen zu finden. Dabei seien, um einen Fehler im größten betrachteten Programm zu finden, in Freya zwischen 10 und 30 Fragen beantwortet, 0 bis 6 Vorgänger in Hat angeschaut und bis zu 3 `observe` in den Quelltext eingebaut worden, wenn Hood verwendet wurde. Diese Zahlen können man jedoch nicht direkt vergleichen, da ihnen vollkommen verschiedene Arbeitsabläufe zugrunde liegen. Angesprochen wurde, dass es signifikante Unterschiede im Zeitverbrauch der einzelnen Methoden gäbe und dass das Nachdenken, wo am besten für Hood die `observe`s eingefügt würden, das Ändern des Quellcodes und das nochmalige Kompilieren einen bedeutend höheren Zeitfaktor darstelle, als das Beantworten einiger Fragen oder das Auswählen eines Vorgängers.

4.3.3 Hilfen und Strategien

Freja leitet den Anwender durch Fragen zu einem Fehler, Hat startet mit der Ausgabe des Programms, einer Fehlermeldung oder dem zuletzt ausgewerteten Redex und die Nach Auffassung der Autoren ist die Hauptaufgabe des Benutzers, die richtigen Redexe auszuwählen. Da es meist viele Unterausdrücke gäbe, zwischen denen der Benutzer wählen kann und das Programm nicht ausgibt, an welcher Position sich der Fehler befinde, sei es leicht, sich in irrelevanten Regionen zu verirren. Hood gebe dagegen dem Benutzer die volle Freiheit, jede Variable im Programm zu überwachen. Hier habe sich eine top-down Strategie bewährt.

4.3.4 Generelle Benutzbarkeit

Hat sei der Debugger mit der steilsten Lernkurve, da der Redex-Browser recht komplex sei. Dagegen sei Freja am Anfängerfreudlichsten, da das Frage-Antwort Prinzip einfach zu verstehen sei. Die Anwendung von Hood könne man mit der Benutzung von `print`-Statements vergleichen, was wiederum ein bekanntes Konzept sei.

4.3.5 Informationsverminderung

Bei HOOD steuert der Anweder selbst die Menge der Ausgaben, indem er entsprechend mehr oder weniger `observe`s platziert. Die Autoren begrüßen den `trust`-Mechanismus in HAT und FREJA, mit dem ein Anwender steuern kann, welche Funktionen beobachtet werden und welche nicht. FREJA hat einen dynamischen `trust`-Ansatz, mit dem die Frage nach einer Funktion nur einmal beantwortet werde muss und mit der weitere Fragen vermieden werden. Dieser fehlt anscheinend in HAT, da sie sich einen ebensolchen Mechanismus in HAT wünschen. Dafür fragt FREJA manchmal dieselbe Frage mehr als einmal, wenn eine Reduktion mehrfach vorgenommen wird. Hier wäre nach Ansicht von Chitil et al. eine Speicherung der Fragen und Antworten wünschenswert.

4.3.6 Laufzeitverbrauch

Da FREYA eine *Low-Level*-Implementierung ist, ist kaum ein Unterschied zum normalen Kompilieren festzustellen. Im Vergleich ist HAT zehnmal langsamer. Dasselbe gilt für HOOD, wenn die `observe` an Stellen gesetzt sind, die sehr oft aufgerufen werden und die zu großen Beobachtungen führen. Das bedeutet, der Zeitverbrauch von HOOD ist bedeutend, aber proportional zur Menge der beobachteten Daten.

5 Zusammenfassung

Es wurden die Tracing und Debbugging-System FREJA, HAT und HOOD verglichen anhand ihrer Nützlichkeit für einen Programmierer. Auf HOOD wurde im 3. Kapitel ausführlich eingegangen, da dies das Hauptthema dieser Ausarbeitung ist. HOOD zeigt sich als vielseitiges Debugging-Tool und zeigt informativ die Strukturen in einem Programm auf, überlässt es jedoch dem Benutzer, die Fehler zu finden und zu beheben. Hier leisten andere Tools mehr, indem sie dem Benutzer an den Ort des Fehlers führen. Zur Ehrenrettung HOODs muss hierbei jedoch gesagt werden, dass auch bei den angesprochenen Tools letztendlich der Benutzer als *Orakel* fungiert und dem Debugger den Fehler aufzeigen muss.

Debugging-Werkzeuge für funktionale Programmiersprachen haben sich beachtlich weiterentwickelt. Die drei vorgestellten sind allesamt praktische und benutzbare Tools zur Fehlersuche auch ausserhalb von Universitäten. Leider werden FREJA und HAT dadurch eingeschränkt, dass sie kein komplettes HASKELL98 unterstützen. HOOD ist dagegen dadurch beschränkt, dass es seit 2001 keine weitere Entwicklung erfahren hat und daher mit aktuellen Compilern nur über Umwege läuft bzw. Funktionen nutzt, welche als *deprecated* gekennzeichnet sind. Eine erfreuliche Ausnahme stellt hierbei der Compiler HUGS dar, der die Funktionalität von HOOD übernommen hat und in leicht abgewandelter Form dem Benutzer zugänglich macht.

Jedes der angesprochenen Werkzeuge hat einen eigenen Ansatz und daher spezifische Stärken. Im Einzelnen: FREJA hat eine systematische Methode, Fehler zu finden. HAT fängt beim beobachteten Fehler an und ermöglicht es, zurückzugehen um den Verlauf jedes Unterausdrucks anzuschauen. HOOD beobachtet den Datenfluss an definierten Punkten nach Bedarf. Alle drei arbeiten nach einem zweichichtigen Modell, welches erst aufzeichnet und dann diese Daten verarbeitet. Dies scheint ein Ansatz zu sein, der auch in Zukunft von vielen HASKELL-Debuggern benutzt werden wird.

Literatur

[1] Chitil, O., C. Runciman und M. Wallace: *Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs*. Lecture Notes in Computer Science, 2011:176–193, 2001.

[2] Gill, A.: *Debugging Haskell by observing intermediate data structures.*. Electronic Notes in Theoretical Computer Science, 41, 2001.