Christian Albrechts Universität zu Kiel
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion
Prof. Dr. Michael Hanus

Seminar

Deklaratives Debugging mit buddha

Stefan Junge

13. November 2006
betreut durch Dr. Bernd Brassel

Inhaltsverzeichnis

1 Einleitung

Dieses Seminar behandelt das deklarative Debugging funktionaler Sprachen. Ich benutze dabei das Werkzeug BUDDHA, welches für die funktionale, nicht strikte Programmiersprache HASKELL implementiert wurde. Ich zeige zuerst, dass man mit herkömmlichen Debuggingmethoden Probleme hat, eine fehlerhafte Funktionsdefinition in einer solchen Sprache zu lokalisieren. Danach wird ein besserer Ansatz vorgestellt und gezeigt, wie dieser in dem Werkzeug BUDDHA umgesetzt ist. Man wird sehen, dass man sich nicht mit dem Quellcode befassen muss, sondern nur mit den einzelnen Funktionsaufrufen, von einem zuvor durchgeführtem Programmdurchlauf. Man muss also nur noch entscheiden, ob ein dargestellter Aufruf korrekt ist oder nicht, ohne wissen zu müssen, wie die Funktionsdefinition konkret aussieht.

1.1 Lazy evaluation

In HASKELL werden Ausdrücke grundsätzlich nicht strikt ausgewertet. Ein Ausdruck, welcher als Parameter einer Funktion übergeben wird, wird erst dann ausgewertet, wenn sein Wert innerhalb des Funktionsrumpfes benötigt wird, zum Beispiel für einen arithmetischen Vergleich. Andernfalls wird er unverändert zurückgegeben, womöglich als Teil eines neuen Ausdrucks. Es ist also auch möglich, dass Ausdrücke nie ausgewertet werden. Diese Strategie nennt man CALL BY NEED oder LAZY EVALUATION. [5]
Ein Beispiel für einen Ausdruck, der während der Ausführung nicht ausgewertet wird:

```
Beispiel 1
--- Konstante Funktion.
const           :: a -> b -> a
const x _       = x

Prelude> print (const 2 (head []))
2

Prelude> print (const (head []) 2)
*** Exception: Prelude.head: empty list
```

Der erste Aufruf `(const 2 (head []))` verursacht keinen Ausnahmefehler, da der Ausdruck `(head [])` nicht ausgewertet wird.

1.2 Das Problem

Debuggingmethoden, welche bevorzugt für imperative Sprachen benutzt werden, sind für nicht strikte Sprachen nicht immer einsetzbar. Hier wird auf das TRACING und das schrittweise Ausführen eines Programmes eingegangen.
Mit TRACING (Ablaufverfolgung) bezeichnet man in der Programmierung eine Funktionalität zur Analyse von Programmen, um einen Fehler zu finden. Dabei wird z. B. bei jedem Einsprung in eine Funktion, sowie bei jedem Verlassen eine Meldung ausgegeben,

sodass der Programmierer mitverfolgen kann, wann und von wo welche Funktion aufgerufen wird. Die Meldungen können auch die Argumente an die Funktion enthalten. Zusammen mit weiteren Diagnose-Ausgaben lässt sich so der Programmablauf eines fehlerhaften Programmes häufig sehr schnell bis zu der fehlerverursachenden Funktion zurückverfolgen. [6]

1.2.1 Tracing unendlicher Datenstrukturen

```
allePrimzahlen :: [Int]
allePrimzahlen = sieb [2..]

sieb (x:xs) = x : sieb (filter (istKeinTeiler x) xs)
   where
      istKeinTeiler x y = mod y x == 0

main = print (take 5 allePrimzahlen)
```

Beschreibung der intendierten Bedeutung der Funktionen:

- `main` gibt die ersten 5 Elemente von `allePrimzahlen` wieder.
- `allePrimzahlen` gibt die unendlich lange Liste aller Primzahlen wieder, der Größe nach sortiert.
- `sieb` bekommt eine Liste mit Zahlen und filtert alle Vielfachen, die mindestens doppelt so groß sind, von allen Zahlen heraus.
- `istKeinTeiler` bekommt 2 Zahlen und prüft, ob die zweite Zahl **nicht** durch die erste teilbar ist.
- [2..] gibt die unendliche Liste aller ganzen Zahlen, welche größer oder gleich 2 sind, wieder.

Dieses Programm soll alle Primzahlen mit dem Sieb des Eratosthenes liefern. Ein Aufruf von `main` liefert aber:

```
*Main> main
[2,4,8,16,32]
```

Dies entspricht offensichtlich nicht dem beabsichtigten Ergebnis. Ich füge nun einen `trace` in `sieb` ein, um zu sehen, welche Argumente `sieb` bekommt:

```
import Debug.Trace

sieb arg@(x:xs) =
    trace (show arg)
          (x : sieb (filter (istKeinTeiler x) xs))
```

Ruft man dieses Programm nun erneut auf, bekommt man keine Ausgabe zu sehen und das Programm terminiert auch nicht. Es hat sich bei der Umwandlung einer unendlich langen Liste in eine Zeichenkette in eine Endlosschleife begeben. Das zeigt, es ist nicht sinnvoll, mit `trace` zu arbeiten, wenn man es mit unendlichen Strukturen zu tun hat. Wir werden später sehen, wie man den Fehler besser finden kann und dazu nur die Beschreibungen der Funktionen benötigt werden. Den Quelltext müssen **wir** nicht verändern.

1.2.2 Laufzeitfehler durch Tracing

```
wurzel :: Float -> Float
wurzel x = if x < 0
    then error ("negativeZahl: " ++ show x)
    else sqrt x

wlist [] = []
wlist (x:xs) = (x,wurzel x) : wlist xs

wurzelliste liste  =
   let  wliste = wlist liste in
   filter ((> 1) . fst)  wliste -- hier ist der Fehler, richtig: (>= 0)

main :: IO ()
main = print (wurzelliste [(-5)..5])
```

Beschreibung der Funktionen:

- `wlist` bekommt eine Liste mit Zahlen und liefert eine Liste mit 2er Tupeln zurück, diese Tupel enthalten die übergebenen Zahlen und deren Quadratwurzel.
- `wurzelliste` bekommt eine Liste mit Zahlen und wendet `wlist` darauf an. Negative Zahlen werden danach aussortiert.
- `[(-5)..5]` liefert eine Liste mit ganzen Zahlen von -5 bis 5.
- `main` ruft `wurzelliste` mit `[(-5)..5]` auf.
- `wurzel` bekommt eine Zahl und liefert deren Quadratwurzel, falls die übergebene Zahl negativ ist, wird ein Fehler ausgegeben.

Ein Aufruf von `main` liefert:

```
*Main> main
[(2.0,1.4142135),(3.0,1.7320508),(4.0,2.0),(5.0,2.236068)]
```

Dieses Ergebnis entspricht nicht unseren Erwartungen, da einige Zahlen (0 und 1) fehlen. Ich füge nun einen `trace` hinzu, um mir die komplette `wliste` anzuschauen:

```
wurzelliste liste  =
   let  wliste = wlist liste in
   filter ((> 1) . fst)  (trace (show wliste) wliste)
```

Jetzt führe ich `main` erneut aus:

```
*Main> main
[(-5.0,*** Exception: negative Zahl: -5.0
```

Man hat einen Ausnahmefehler erzeugt, den es bei einem normalen Aufruf nicht gegeben hätte. Also darf man nicht an einer beliebigen Stelle einen `trace` hinzufügen, weil Terme ausgewertet werden könnten, welche normalerweise nicht ausgewertet werden.

1.2.3 schrittweises Ausführen

In imperativen Sprachen ist das schrittweise Ausführen des Programms eine gute Methode einen Fehler zu finden. Um jedoch einen Fehler in einem nicht strikten Programm zu finden, ist es keine gute Lösung, da Berechnungen erst durchgeführt werden, wenn sie gebraucht werden. Man bekommt also mitten im Programm noch Berechnungen zu sehen, die z.B. zur Initialisierungsphase gehören. Dies kann sehr verwirrend sein. Wir verkürzen das Beispiel mit den Primzahlen ein wenig und schauen uns an, in welcher Reihenfolge die Funktionen ausgeführt werden.

```
allePrimzahlen :: [Int]
allePrimzahlen = sieb [2..7]

sieb [] = []
sieb (x:xs) = x : sieb (filter (istKeinTeiler x) xs)
  where
     istKeinTeiler x y = mod y x /= 0

main = print allePrimzahlen
```

aktuelle Auswertungen, mit Hilfe von `pakcs` (:set +debug):

```
main
allePrimzahlen
(sieb (enumFromTo 2 7))
(sieb (filter (istKeinTeiler 2) (enumFromTo (2 + 1) 7)))
(istKeinTeiler 2 3)
(sieb (filter (istKeinTeiler 3)
              (filter (istKeinTeiler 2) (enumFromTo (3 + 1) 7))))
(istKeinTeiler 2 4)
(istKeinTeiler 2 5)
(istKeinTeiler 3 5)
```

```
(sieb (filter (istKeinTeiler 5)
              (filter (istKeinTeiler 3)
                      (filter (istKeinTeiler 2)
                              (enumFromTo (5 + 1) 7)))))
(istKeinTeiler 2 6)
(istKeinTeiler 2 7)
(istKeinTeiler 3 7)
(istKeinTeiler 5 7)
(sieb (filter
         (istKeinTeiler 7)
         (filter
             (istKeinTeiler 5)
             (filter (istKeinTeiler 3)
                     (filter (istKeinTeiler 2)
                             (enumFromTo (7 + 1) 7))))))
```

Es ist zu erkennen, dass die Tests `(istKeinTeiler 2 7)` `(istKeinTeiler 3 7)` und `(istKeinTeiler 5 7)` erst gemacht werden, wenn die 7 an der Reihe ist. In einer strikten Sprache würden erst alle 2er Tests, dann alle 3er und 5er Tests durchgeführt werden. Die Länge der unausgewerteten Argumente nimmt in diesem Fall auch zu und kann mit der Zeit sehr unübersichtlich werden.

1.2.4 Fazit

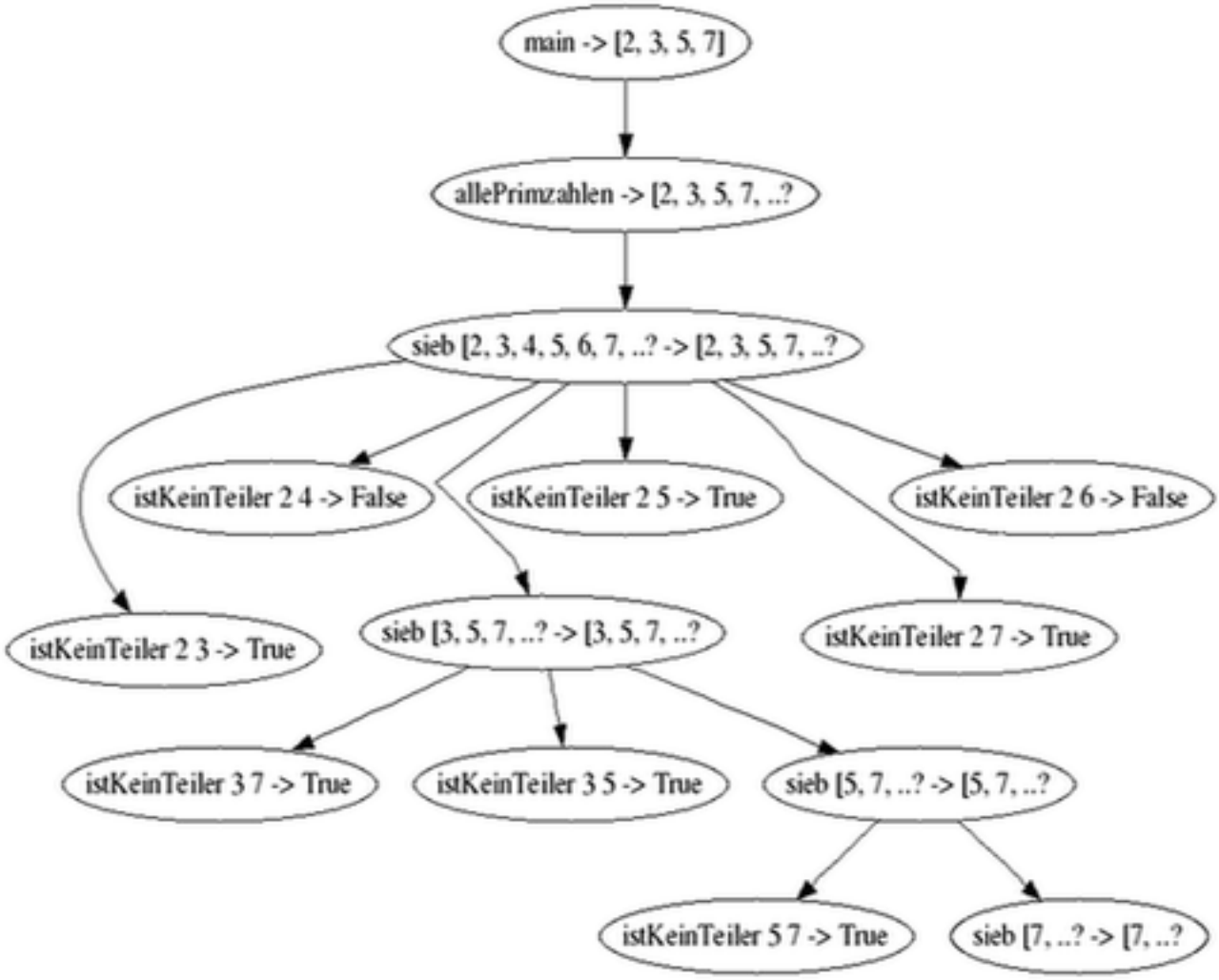
Benutzt man Tracing, werden oft neue Probleme hervorgerufen und verwendet man das schrittweise Debugging, verliert man schnell den Überblick. Also benötigt man andere Methoden, um einen Fehler in einer nicht strikten Sprache zu finden. Hier kommt das Deklarative Debugging ins Spiel.

2 Grundlagen

2.1 Evaluation Dependence Tree

Nilsson und Sparud haben erkannt, dass es sinnvoller ist, den Ablauf eines Programms erst aufzuzeichnen und mit einer baumartigen Struktur darzustellen. Dieser Baum zeigt die Struktur des Quellcodes, anstatt die Reihenfolge der Berechnungen. Er wird von ihnen als Evaluation Dependence Tree (EDT) bezeichnet. Jeder Knoten des EDTs steht für einen Funktionsaufruf, welcher bei der Durchführung des Programmes auftritt. Die Knoten bestehen nicht nur aus den Funktionswerten, sondern auch aus den Funktionsnamen und den Argumenten. Die Kinder dieser Knoten sind die Funktionsaufrufe, welche laut dem Quelltext im Rumpf der Funktion auftreten. Die Argumente und Rückgabewerte werden in ihrer größtmöglich ausgewerteten Form dargestellt, welche womöglich erst am Ende des Programmablaufs feststeht. Alle Ausdrücke, die bis zum Ende hin nicht ausgewertet wurden, werden durch ein „?“ repräsentiert. [2]

Hier der EDT für den Durchlauf von `main` aus dem Beispiel, welches die ersten 4 Primzahlen bestimmt:



Evaluation Dependence Tree

Die Knoten im EDT sind anhand ihrer syntaktischen Position in der Programmdefinition angeordnet. Man kann jeden Knoten für sich betrachten, um herauszufinden, ob der dadurch repräsentierte Funktionsaufruf korrekt ist.

2.2 Zweistufen Debugger Architektur

[2] Nilsson und Sparud haben eine zweistufige Debugger Architektur vorgestellt. Die untere Stufe ist mit der Konstruktion des EDTs beschäftigt, die obere mit dessen Präsentation und der Navigation im EDT, um Fehler zu finden. Im Folgendem wird die untere Stufe als EDT Generator und die obere als EDT Navigator bezeichnet. Diese Trennung bringt einige Vorteile:

- Ein einzelner EDT Generator könnte die Basis für viele Debugger sein, z. B. einen deklarativen Debugger oder einen, mit dem man im EDT navigieren kann.
- Die Konstruktion des EDTs ist nicht so einfach mit einer lazy funktionalen Sprache durchzuführen, daher ist es sinnvoll dies auszugliedern.

2.2.1 EDT Generator

Nilsson benutzte eine veränderte Sprachimplementierung, welche den EDT als Seiteneffekt erstellt. Dies ist aber nicht leicht übertragbar zwischen verschiedenen Implementierungen derselben Sprache.
Sparud benutzte eine Quellcodetransformation, so dass die Funktionswerte nun aus Tupeln bestehen, mit dem ursprünglichem Funktionswert und einem EDT-Knoten. Für die Repräsentation der Argumente und Rückgabewerte wurde ein eigener Datentyp verwendet, da Haskell es nicht erlaubt, verschiedene Typen an der selben Stelle im EDT Knoten zu speichern. [2]
In BUDDHA wurde die Transformation von Naish und Barbour umgesetzt. Sie ähnelt der Transformation von Sparud, allerdings werden die Argumente und Rückgabewerte als String dargestellt. Dazu wird eine Funktion `dirt` verwendet, welche einen beliebigen Term in einen String umwandelt. [1]
Im Kapitel 3.3 findet man eine Implementierung der Transformation.

2.2.2 EDT Navigator

Wenn ein Programm durchgelaufen ist, sind seine Berechnungen in einem EDT gespeichert. Um einen Fehler zu finden, wird der Benutzer auf diesem Baum navigieren. Dafür ist der EDT Navigator verantwortlich. Die deklarative Debuggingmethode ist ein Weg, den Benutzer durch einen evtl. sehr großen EDT zum richtigen Punkt zu führen. In diesem Fall ist es eine fehlerhafte Funktionsdefinition. In welcher Weise der Benutzer nach der Korrektheit von Funktionsaufrufen gefragt wird, sehen wir im Hauptteil, im Kapitel 3.2.1 DAS ORAKEL. Ein anderer Weg, einen Fehler zu finden, ist das Navigieren durch den EDT, man entscheidet also selber, welcher Knoten angezeigt wird. Dies ist nützlich, falls man schon eine grobe Vorstellung hat, wo sich der Fehler befindet. Man kann auch beide Varianten kombinieren, indem man erst zu einem Bereich hin navigiert und dort das deklarative Debugging startet.

3 Hauptteil

3.1 Quellcode transformieren mit buddha

Zuerst muss man `buddha` installieren und dafür sorgen, dass es in der Umgebungsvariablen `PATH` eingetragen ist. Eine entsprechende Beschreibung findet man im Anhang.
Um ein Modul zu transformieren, muss man ins Verzeichnis wechseln, in dem sich die entsprechende Datei befindet. Dieses Modul muss eine Funktion `main :: IO()` enthalten.

Um die Transformation zu starten, führt man `buddha` aus. Eine Übersicht der möglichen Optionen findet man im Anhang. Ich transformiere hier das Modul `Primes.hs` aus dem Beispiel in der Einleitung:

```
./buddha
buddha 1.2: initialising
buddha 1.2: transforming:
Primes.hs buddha 1.2: compiling
Chasing modules from: Main.hs
Compiling Main_B (./Main_B.hs, ./Main_B.o )
Compiling Main   (Main.hs, Main.o)
Linking ...
buddha 1.2: done
```

Es wird an der aktuellen Position ein Verzeichnis `Buddha` mit Dateien erstellt, welche für die Transformation notwendig sind. Danach wird das Modul in `Main_B.hs` transformiert. Sind dabei keine Fehler aufgetreten, ist die Transformation abgeschlossen. Das Fehlen von Typsignaturen war bei mir ein häufiger Grund für das Fehlschlagen der Transformation.

3.1.1 Transformation mehrerer Module

Hat man ein Projekt mit mehreren Modulen, erkennt `buddha` automatisch deren Abhängigkeiten und speichert sie in `./Buddha/depend`. Möchte man nur bestimmte Module transformieren, kann man deren Dateinamen als Argumente an `buddha` übergeben.

```
buddha ModulA.hs ModulB.hs ModulC.hs
```

Zählt man ein benötigtes Modul nicht auf, muss dafür gesorgt werden, dass eine transformierte Version dieses Moduls schon vorhanden ist, sonst kommt es zu einer entsprechenden Fehlermeldung.

3.2 Fehlersuche mit debug

Hat man das Projekt transformiert, kann man in das Verzeichnis `Buddha` wechseln und dort die ausführbare Datei `./debug` aufrufen. Dadurch wird das zu testende Programm gestartet und man kann wie gewohnt damit arbeiten. Ist das Programm terminiert, startet sich `buddha` und man bekommt den ersten EDT Knoten zu sehen, welcher in der Regel den Aufruf der `main` Funktion beschreibt.

```
./debug
[2,4,8]
------------------------------------------------------------------------

Welcome to buddha, version 1.2 A declarative debugger for Haskell
```

```
Copyright (C) 2004, Bernie Pope http://www.cs.mu.oz.au/~bjpop/buddha

Type h for help, q to quit

--------------------------------------------------------------------

[1] Main 9 main
   result = <IO>

buddha:
```

Jetzt kann man mit der Fehlersuche beginnen, denn 2,4 und 8 sind nicht die ersten Primzahlen.

3.2.1 Das Orakel

Ziel der Fehlersuche ist es, den TOPMOST BUGGY NODE zu finden. Die Suche beginnt an der Wurzel des EDTs. Es wird der Funktionsname, seine Argumente und der Rückgabewert angezeigt. Ein so genanntes Orakel muss nun entscheiden, ob dieser Knoten korrekt ist oder nicht. Bei `buddha` ist der Benutzer das Orakel. Ist der Knoten nicht korrekt, werden nach und nach seine Kinder überprüft, von links nach rechts. Ist ein Kind korrekt, wird das nächste betrachtet. Wird ein nicht korrektes Kind gefunden, konzentriert man sich auf diesen Knoten und überprüft rekursiv seine Kinder. Hat ein nicht korrekter Knoten keine oder nur korrekte Kinder und ausschließlich nicht korrekte Vorfahren, haben wir einen TOPMOST BUGGY NODE gefunden. [1] Also tritt ein Fehler im Programm auf, wenn man die Funktion, mit den dargestellten Argumenten, aufruft, welche den Knoten repräsentiert. Somit kann man sich diese Stelle im Quellcode anschauen, um den Fehler zu korrigieren. Hat man den Quellcode verändert, muss man das Modul neu transformieren. Es wird automatisch erkannt, welche Module sich verändert haben und nur diese werden transformiert. Es kann natürlich sein, dass es mehrere Fehler im Programm gibt, also könnte ein erneuter Aufruf einen neuen Fehler zeigen. Somit beginnt die Suche von vorne. Ich zeige nun eine Fehlersuche und benutze dabei das Primzahlenbeispiel aus der Einleitung. Allerdings habe ich die `main` Funktion so geändert, dass nur die ersten 3 Primzahlen angezeigt werden.

```
./debug
[2,4,8]

[1] Main 9 main
   result = <IO>
```

Die erste Zeile der Darstellung des EDT Knotens besteht aus 4 Elementen. Das erste Element ist die eindeutige Knotennummer, hier `[1]`. Das zweite Element ist der Name des Moduls, hier `Main`, in dem sich die Funktion befindet. Das dritte Element ist die Zeilennummer im Quellcode, hier `9`, wo sich die Definition der Funktion befindet. Das

vierte Element ist der Funktionsname, hier `main`. In den Zeilen darunter folgen die Argumente für diesen Aufruf und zum Schluss der Rückgabewert. Hier stößt man schon auf ein Problem, denn mit `buddha` kann man Rückgabewerte von IO Aktionen nicht darstellen, also muss man anhand des zuvor durchgeführten Programmdurchlaufs entscheiden, ob dieser Funktionsaufruf korrekt ist. Der Funktionsaufruf von `main` hat die Liste `[2,4,8]` zurückgeliefert, da `main` die ersten 3 Primzahlen zurückliefern sollte, ist dieser EDT Knoten falsch, also geben wir `e` für `erroneous` ein. Eine andere Möglichkeit mit IO Funktionen umzugehen ist, sich mit `k` alle Kinder anzeigen zu lassen und z. B. mit `j 2` zum Knoten 2 zu springen. Wenn man `h` eingibt, bekommt man eine Übersicht aller Eingabemöglichkeiten, diese werden im Anhang kurz beschrieben.

```
buddha: e

[2] Main 3 allePrimzahlen
   result = [2, 4, 8, ..?
```

Nun bekommen wir das erste und einzige Kind von Knoten 1 zu sehen. Der Rückgabewert von `allePrimzahlen` ist eine unendliche Liste, es wurden während des Programmdurchlaufs nur die ersten 3 Elemente dieser Liste benötigt, also werden diese 3 auch nur angezeigt. Durch das `..?` wird angedeutet, dass diese Liste noch mehr Elemente hat, welche aber nicht ausgewertet wurden. Dieser Knoten ist auch falsch, da 4 und 8 keine Primzahlen sind, also geben wir wieder `e` ein.

```
buddha: e

[3] Main 5 sieb
   arg 1  = [2, 3, 4, 5, 6, 7, 8, ..?
   result = [2, 4, 8, ..?
```

Die Funktion `sieb` entfernt alle Vielfache, die mindestens doppelt so groß sind, von allen Zahlen, es wurde aber auch die 3 entfernt, also ist dieser Knoten falsch.

```
buddha: e

[4] Main 5 sieb
   arg 1  = [4, 6, 8, ..?
   result = [4, 8, ..?
```

Auch dieser Knoten ist nicht korrekt, da zum einen die 6 entfernt wurde, obwohl `arg 1` keinen Teiler von 6 enthält und zum anderen die 8 nicht entfernt wurde, obwohl die 4 ein Teiler von 8 ist.

```
buddha: e

[7] Main 5 sieb
   arg 1  = [8, ..?
   result = [8, ..?
```

Dieser Knoten scheint korrekt zu sein, da `sieb` die 8 nicht entfernt hat. Was diese Funktion mit dem Rest der Liste anstellt, kann uns egal sein, da sie beim Programmdurchlauf nicht auf ihn angewandt wurde, also keine Ursache des aktuellen Fehlers ist. Wir geben deswegen `c` für `correct` ein.

```
buddha: c

[10] Main 7 istKeinTeiler
   arg 1  = 4
   arg 2  = 6
   result = False
```

6 ist nicht teilbar durch 4, also müsste `True` zurückgeliefert werden, was hier aber nicht der Fall ist, also geben wir `e` ein.

```
buddha: e

Found a bug: [10] Main 7 istKeinTeiler
   arg 1  = 4
   arg 2  = 6
   result = False
```

Der Debugger hat den Fehler gefunden. Er befindet sich im Modul `Main.hs`, Zeile 10, in der Funktion `istKeinTeiler` und zwar wenn es die Argumente 4 und 6 bekommt. Und wenn wir nachschauen, hat der Debugger recht.

```
istKeinTeiler x y = mod y x == 0
```

ist falsch, es muss so aussehen:

```
istKeinTeiler x y = mod y x /= 0
```

Wenn wir den Fehler korrigieren und neu transformieren, erhalten wir:

```
 ./debug
[2,3,5]
```

Was korrekt ist, also sind wir fertig.

3.2.2 Funktionen höherer Ordnung

[3] In funktionalen Sprachen kann man Funktionen als Argumente übergeben oder als Rückgabewert bekommen. Hier ein Beispiel:

```
plus :: Int -> Int -> Int
plus x y = x - y

myMap _ [] = []
myMap f (x:xs) = f x : myMap f xs

myMap2 _ [] = []
myMap2 f (x:xs) = f (f x) : myMap2 f xs

main = print (myMap (plus 1) [1,2,3])
```

Ich musste eine eigene `map`-Funktion erstellen, da Funktionen, die aus dem Modul `Prelude` stammen, als korrekt angenommen werden. Es ist zu beachten, dass das erste Argument von `myMap` und `myMap2` eine Funktion ist. Im Laufe der Fehlersuche bekommen wir einen Knoten mit `myMap` zu sehen, wie sollte dessen erstes Argument angezeigt werden? In `buddha` wird eine Liste mit Funktionsaufrufen angezeigt `(x,g(x))`, wobei `x` aus der Definitionsmenge von `g` stammt, und `g` die übergebene Funktion ist. Es werden nur Elemente genommen, die auch während des Programmaufrufs berechnet wurden, da das Programm terminiert ist, sind es nur endlich viele. Ich transformiere nun das Programm und springe zu einem Knoten mit `myMap`:

```
buddha: j 2

[2] Main 5 myMap
   arg 1  = { 1 -> 0, 2 -> -1, 3 -> -2 }
   arg 2  = [1, 2, 3]
   result = [0, -1, -2]
```

Man sieht wie `(plus 1)` auf die Zahlen 1, 2 und 3 angewendet wird und dass die Ergebnisse nicht richtig sind, da wir 2,3 und 4 erwartet hätten. Aber dies ist zur Zeit nicht wichtig, da wir hier nicht entscheiden sollen, ob `(plus1)` richtig ausgeführt wird, sondern ob `myMap` richtig funktioniert. `myMap` wendet die Funktion `arg 1` auf alle Elemente von `arg 2` an, 1 wird zu 0, 2 zu -1 und 3 zu -2 und dies entspricht dem Resultat, also ist dieser Knoten korrekt und wir geben `c` ein.

```
buddha: c

[5] Main 3 plus
   arg 1  = 1
   arg 2  = 2
   result = -1
```

Und hier haben wir den Fehler.
Ich zeige nun wie eine fehlerhafte `map`-Funktion aussieht, also ersetze ich `myMap` durch `myMap2`:

```
buddha: j 2

[2] Main 8 myMap2
   arg 1  = { 0 -> 1, 1 -> 0, -1 -> 2, 2 -> -1, -2 -> 3, 3 -> -2 }
   arg 2  = [1, 2, 3]
   result = [1, 2, 3]
```

`arg1` wird auf alle Elemente aus `arg2` angewendet, 1 wird zu 0, 2 zu -1 und 3 zu -2, dies stimmt aber nicht mit dem Resultat überein, also kann man hier sagen, dass dieser Knoten falsch ist.

3.2.3 Trust

[3] Damit man bei großen Programmen nicht zu viele Fragen beantworten muss, kann man bestimmen, welche Funktionen man sehen möchte und welche nicht. Je mehr Funktionen man ausblendet, desto weniger Knoten hat der EDT und desto schneller findet man den Fehler.
Um `buddha` mitzuteilen, welche Funktionen man sehen möchte, muss man für jedes Modul eine Textdatei erstellen. Für das Modul `./Primes.hs`, lautet der Speicherort `./Buddha/Primes.opt`.

```
allePrimzahlen :: [Int]
allePrimzahlen = sieb [2..]

sieb (x:xs) = x : sieb (filter (istKeinTeiler x) xs)
   where
      istKeinTeiler x y = mod' y x == 0
        where
            mod' = mod

main = print (take 5 allePrimzahlen)
```

Hier ein Beispiel für eine solche Textdatei:

```
_ ; Trust
main ; Suspect
sieb ; Suspect
```

Optionen müssen zeilenweise angegeben werden, jede Zeile enthält einen Funktionsnamen, ein Semikolon und eine Option, die entscheidet, ob wir dieser Funktion trauen oder nicht. Mögliche Optionen sind:

Trust Funktionsaufrufe werden nicht beachtet, es werden nur Funktionsaufrufe im Rumpf gesammelt.

Suspect Es wird ein vollwertiger EDT-Knoten für jeden Funktionsaufruf erstellt. Also werden Funktionsname, Argumente und der Rückgabewert gespeichert. Dies ist der Standartwert.

In der ersten Zeile wird ein neuer Standartwert definiert. Unterstriche stehen für alle Funktionen, in Anlehnung an HASKELL. Es werden also nur die Funktionen `main` und `sieb` betrachtet. Optionen für die lokale Funktion `istKeinTeiler` sind auch möglich:

```
sieb istKeinTeiler ; Suspect
```

Und für `mod'` welche local zu `istKeinTeiler` ist:

```
sieb istKeinTeiler mod' ; Suspect
```

Dies hat keinen Einfluss auf die Option von `sieb`. Man kann auch eine Option für alle direkten Kinder definieren:

```
sieb _ ; Suspect
```

Man kann nur **einen** Unterstrich benutzen und falls einer benutzt wird, muss unmittelbar ein Semikolon folgen. Folgende Einträge sind nicht erlaubt:

```
_ istKeinTeiler ; Trust
sieb _ mod' ; Suspect
sieb _ _ ; Suspect
```

3.3 Implementierung des EDT Generators

Hier der Datentyp für einen EDT Knoten, der bei buddha verwendet wird:

```
data Tree = Node String [Term] Term [Tree] String
```

Bei der aktuellen Version ist `Term` ein Synonym für `String`. `Node` enthält folgende Inhalte:

- `String` Funktionsname
- `[Term]` Argumente
- `Term` Rückgabewert
- `[Tree]` Kinder
- `String` Quellcode der Funktionsdefinition

Hier die Transformationsvorschrift:
Für alle Gleichungen der Form `f` a_1 a_2 `...` a_n `= r` sind folgende 3 Schritte durchzuführen:
1. Schreibe die Funktion um:

```
f a1 a2 ... an = (v0,t0) where v0 = r
```

2. Wiederhole folgenden Schritt, bis es in `r` keinen Funktionsaufruf mehr gibt: Falls `g` b_1 b_2 `...` b_m ein innerster Funktionsaufruf in `r` ist, ersetze diesen Aufruf durch v_i und füge folgende Gleichung in die oberste `where`-Klausel von `f` ein (v_i und t_i sind neue und eindeutige Variablen in `f`):

```
(vi,ti) = g b1 b2 ... bm
```

3. Füge folgende Gleichung in die oberste `where`-Klausel von `f` ein (t_1`,...,`t_p gehören zu den t_i aus dem vorherigem Schritt):

```
t0 = Node "f"
          [dirt a1,...,dirt an]
          (dirt v0)
          [t1,...,tp]
          (text reference)
```

Es wird nun die Transformation folgender Funktion vorgestellt:

```
insort :: Ord a => [a] > [a]
insort [] = []
insort (x:xs) = insert x (insort xs)

insert :: Ord a => a > [a] > [a]
insert x [] = [x]
insert x (y:ys)
    | y > x = y : (insert x ys)
    | x < y = x:y:ys
    | otherwise = y:ys
```

1. Gleichung umschreiben, wie im 1. Schritt beschrieben:

```
insort (x:xs) = (v0, t0)
  where
      v0 = insert x (insort xs)
```

2. Der innerste Funktionsaufruf ist `(insort xs)`, dies wird durch eine neue Variable v_1 ersetzt und es wird eine neue Gleichung in die `where`-Klausel hinzugefügt:

```
insort (x:xs) = (v0, t0)
  where
    v0 = insert x v1
    (v1, t1) = insort xs
```

3. Der innerste Funktionsaufruf ist `(insert x v1)`, dies wird durch eine neue Variable v_2 ersetzt und es wird eine neue Gleichung in die `where`-Klausel hinzugefügt:

```
insort (x:xs) = (v0, t0)
  where
    v0 = v2
    (v1, t1) = insort xs
    (v2, t2) = insert x v1
```

4. v_0 hat keine Funktionsaufrufe mehr, also wird der EDT Knoten hinzugefügt:

```
insort (x:xs) = (v0, t0)
  where
    v0 = v2
    (v1, t1) = insort xs
    (v2, t2) = insert x v1
    t0 = Node "insort"
              [dirt (x : xs)]
              (dirt v0)
              [t1,t2]
              "insort ..."
```

Die Funktion `dirt` spielt eine entscheidende Rolle, denn sie muss jeden beliebigen Datentyp in einen einzigen Datentyp verwandeln. Dabei darf sie selbst keine Auswertungen vornehmen, sondern muss darauf achten, wie weit das aktuelle Programm das übergebene Argument auswertet. Nicht ausgewertete Terme werden durch ein Sonderzeichen z.B. `?` ersetzt. Um dies zu ermöglichen, muss `dirt` die innere Laufzeitrepräsentation eines Objektes bekannt sein, ohne ihn selber auszuwerten, dies verstößt gegen Haskells deklarative Semantik, da man mit Haskell nur den Wert eines Objekts anschauen kann und nicht den Fortschritt der Auswertung.
In dem Artikel [1] wird noch auf die Transformation von Funktionen mit `where`-Klauseln und Funktionen höherer Ordnung eingegangen, darauf gehe ich aber an dieser Stelle nicht mehr ein, da es den Rahmen des Seminars sprengen würde.

3.4 Probleme und Grenzen

`buddha`:

- `buddha` ist für Haskell 98 entwickelt worden, viele Spracherweiterungen werden noch nicht unterstützt, wie z. B. eine hierarchische Modulstruktur.
- Rückgabewerte von IO Aktionen werden nur durch `<IO>` dargestellt, daher ist es schwer, über deren Korrektheit zu entscheiden.
- Benutzt man globale Variablen in lokalen Funktionen, welche mit `where` oder `let` erstellt wurden, kann es sehr schwer sein, eine Aussage über die Korrektheit des Funktionsaufrufs zu machen. Der Knoten `[10]` aus dem Kapitel 3.2.1 Das Orakel würde wie folgt aussehen, falls man die Definition von `sieb` derart verändern würde:

```
sieb (x:xs) = x : sieb (filter istKeinTeiler xs)
   where
      istKeinTeiler y = mod y x /= 0

[10] Main 7 istKeinTeiler
   arg 1  = 6
   result = False
```

Man hat zu wenig Informationen, um zu entscheiden, ob dieser Aufruf korrekt ist.

- Fehlende Typsignaturen können Fehler bei der Transformation verursachen.

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = (rev xs) ++ [x]

main = print (rev [1..10])
```

Die Transformation liefert folgenden Fehler:

```
./Main_B.hs:1:
    Ambiguous type variable ‘a’ in these top-level constraints:
      ‘Enum a’ arising from use of ‘enumFromTo’ at ./Main_B.hs:16
      ‘Num a’ arising from use of ‘fromInteger’ at ./Main_B.hs:16
```

Folgende Änderung löst das Problem: `main = print (rev ([1..10]::[Int]))`

- Das Projekt muss sich in **einem** Verzeichnis befinden.

allgemein:

- Um einen Fehler zu lokalisieren, muss man ein konkretes Beispiel gefunden haben, welches auch einen Fehler enthält.

4 Zusammenfassung

Mit Buddha hat man die Möglichkeit, eine fehlerhafte Funktionsdefinition im Quellcode eines Haskell-Programms zu finden. Es werden dabei keine Ausdrücke ausgewertet, welche nicht auch bei einem normalem Programmaufruf ausgewertet werden. Die zusätzliche Auswertung von Ausdrücken könnte nämlich weitere Fehler hervorrufen.
Die Methode, welche Buddha umsetzt, heißt deklaratives Debugging und ist hier in 2 Stufen unterteilt. Die erste Stufe transformiert den Quellcode so um, dass er bei der Ausführung einen Evaluation Dependence Tree erstellt. In ihm sind alle Funktionsaufrufe gespeichert, welche bei dieser Ausführung aufgetreten sind. In der zweiten Stufe navigiert man auf diesem Baum, um einen topmost buggy node zu finden,

einen Knoten mit ausschließlich korrekten Kindern und nicht korrekten Vorfahren. Der Fehler befindet sich in der Definition der Funktion, dessen Aufruf durch diesen Knoten repräsentiert wird.
BUDDHA ist in HASKELL geschrieben und für HASKELL 98 vorgesehen. Es werden noch nicht alle Spracherweiterungen unterstützt, welche später hinzugekommen sind. Dieser Debugger wurde zwar weiterentwickelt, allerdings ist der letzte offizielle Beitrag auf dessen Homepage im 28. Mai 2004 erstellt worden.
Hat man alle Installationsvoraussetzungen erfüllt, ist das Programm sehr einfach zu installieren. Die Transformation des Quellcodes ist ebenfalls recht einfach, man muss nur das Programm `buddha` ausführen. Bei der Fehlersuche braucht man nur `e` für einen fehlerhaften und `c` für einen korrekten Knoten eingeben, um den TOPMOST BUGGY NODE zu finden.
Ich finde, man kann mit diesem Programm gut arbeiten. Es ist allerdings etwas nervig, dass man schon während der Entwicklungsphase eines Programms Typsignaturen angeben muss und dass die aktuelle Version nur `ghc 6.2` unterstützt. Es gibt aber schon einen inoffiziellen Patch für `ghc 6.6`, dieser verursacht aber noch einige Probleme.

5 Literatur

Literatur

[1] B. Pope. Buddha: A declarative debugger for Haskell, In Technical Report, Dept. of Computer Science, University of Melbourne, Australia, June 1998. Honours Thesis.

[2] J. Sparud and H. Nilsson. The Architecture of a Debugger for Lazy Functional Langauges. In Proceedings of AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging, 1995.

[3] buddha Version 1.2 User's Guide, Bernie Pope, http://www.cs.mu.oz.au/~bjpop/buddha/

[4] Manual page, buddha: a declarative debugger for Haskell, Bernie Pope 28 May 2004 Version 1.2,http://www.cs.mu.oz.au/~bjpop/buddha/

[5] Seminar, Arne Steinmetz, 12.05.2002, Lazy evaluation, Vor- und Nachteile gegenüber Parameterübergabe per Wert und per Referenz, http://www.fh-wedel.de/~si/seminare/ss02/Ausarbeitung/3.lazy/lazy0.htm

[6] Wikipedia, http://de.wikipedia.org/wiki/Tracing

6 Anhang

6.1 Installation von buddha

6.1.1 Voraussetzungen

Man benötigt folgende Programme:

- GHC 6.0.x - 6.2.x (GHC 6.4 ist nicht mehr kompatibel).
- profiling Bibliotheken für GHC
- Gnu readline und die dazugehörige GHC Bibliothek
- autoconf und automake wird bei der Installationsroutine benutzt

Den Quellcode von buddha kann man unter `http://www.cs.mu.oz.au/~bjpop/buddha/` herunterladen. Das dort zu findende Archiv `buddha-1.2.tar.gz` ist in einem eigenem Verzeichnis zu entpacken.

6.1.2 Konfiguration

Es muss zuerst eine automatische Konfiguration durchgeführt werden. Hierbei werden in jedem Verzeichnis Makefiles erstellt, welche maschinenspezifische Einstellungen enthalten. Um diese Konfiguration zu starten, muss man das Skript `configure` aufrufen. Hierbei wird analysiert, welche Programme vorhanden sind und stellt fest, ob etwas fehlt.
Diesem Skript kann man mehrere Parameter übergeben:

–prefix=/eigenes/verzeichnis Mit dieser Option kann man ein eigenes Zielverzeichnis eingeben. Der Standardwert ist `/usr/local/`. Wählt man einen anderen Ort, sollte man diesen auch in der Umgebungsvariable `PATH` einbinden.

–with-ghc=/verz/zu/ghc Dies ist nützlich, falls man mehrere GHC Versionen installiert hat oder den GHC nicht in der Umgebungsvariable `PATH` hat.

Beispiel:
```
./configure --prefix=/home/sjun/buddha --with-ghc=/home/sjun/ghc6-2/ghc
```

6.1.3 make

Hat man die Konfiguration durchgeführt, kann man ein `make` aufrufen. Dadurch werden alle Programme kompiliert. Nun kann man mit `make install` die erstellten Programme und Pakete in die dafür vorgesehene Verzeichnisse kopieren, hierfür sind ggf. root-Rechte notwendig. Mit einem `make clean` kann man das Arbeitsverzeichnis wieder aufräumen.

6.2 buddha

6.2.1 Optionen

Die wichtigsten Optionen von `buddha` sind:

-v Zeigt die Versionsinformation. Damit kann man schnell testen, ob buddha erfolgreich installiert wurde

-h Zeigt die Hilfeseite an

-t cheapHo Normalerweise speichert `buddha` auch Informationen von Funktionen höherer Ordnung, welche als Argumente oder Rückgabewerte von Funktionen übergeben werden können. Hiermit kann man dies unterbinden. Damit kann man Speicherplatz sparen und entsprechende Werte werden durch `<function>` ersetzt.

-t noprelude Das Module `Prelude` wird nicht importiert.

-c silent Es werden bei der Transformation nur wichtige Information gezeigt

-c normal Es werden nur so viel Informationen angezeigt, um zu zeigen, dass die Transformation noch läuft. (Standartwert)

-c verbose Es werden sehr viele Informationen angezeigt.

6.2.2 Konfiguration

Es gibt 2 Konfigurationsdateien, eine globale und eine lokale, wobei die lokale optional ist. Die Position der globalen ist `<prefix>/share/buddha/buddha.conf` und die der lokalen `.buddha.conf` im `home`-Verzeichnis.

6.3 debug

6.3.1 Optionen

Eine Übersicht der wichtigsten Optionen:

c,correct Die aktuelle Ableitung entspricht den Erwartungen, ist also korrekt.

e,erroneous Die angegebene Funktion liefert mit diesen Argumenten ein falsches Ergebnis, also gibt es einen Fehler

i,inadmissible Die Funktion hat falsche oder unerwartete Argumente bekommen, so dass man nicht entscheiden kann, ob die Funktion ein richtiges oder falsches Ergebnis geliefert hat.

u,unknown Wenn man keine Idee hat, ob diese Ableitung `correct`, `erroneous` oder `inadmissible` ist, sollte man diese Option wählen.

d,defer Falls man zur Zeit keine Aussage über die aktuelle Ableitung machen möchte, kann man sie zurück stellen. Wird aber kein Fehler gefunden, kommt man wieder hierher zurück.

q,quit Beendet das Programm.

k,kids Die Kinder der aktuellen Ableitung werden angezeigt.

p,parents Der Vater der aktuellen Ableitung wird angezeigt.

set Zeigt oder ändert eine von buddhas Optionen.

r,refresh Der aktuelle Knoten wird neu gezeichnet.

forget Alle Orakeleingaben werden vergessen.

restart Es wird von vorne begonnen.

observe Alle Aufrufe einer Funktion oder Konstanten wird angezeigt.

j,jump Springt zu einem angegebenem Knoten

b,back Springt zum Knoten, von dem man gekommen ist.