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

Seminar

HAT - Der Haskell Tracer

Maik Barz

12. Oktober 2006
betreut durch Dr. Bernd Brassel

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist HAT?

„HAT“ steht für *Haskell Tracer* und ist ein Quellcode Tracing Programm zum Auffinden von normalerweise unsichtbaren Berechnungsinformationen. Zum Einen lässt sich mittels HAT ein Programmablauf nachverfolgen, zum Anderen ist es auch möglich Programmfehler zu lokalisieren. Hierfür wird bei der Ausführung des zu tracenden Programms eine spezielle Trace-Datei erstellt, die alle Ersetzungen und Berechnungen enthält. HAT bietet für diese Datei verschiedene Werkzeuge an, um, je nach Wunsch, bestimmte Codefragmente darzustellen. Später werde ich noch einige Beispiele hierfür zeigen.

1.2 Voraussetzungen

Zur Installation und Benutzung des Haskell Tracer Programms werden die Pakete „hmake“ und „ghc“ oder „nhc98“benötigt. Weiterhin muss neuerdings bei GHC auch das „ghc-prof“ Package installiert werden, damit ausgegliederte Module, die HAT benötigt, vorhanden sind. Ich benutze in diesem Paper „GHC 6.4.2“, „hmake 3.12“ und „HAT 2.04“. Zu beachten ist noch, dass ich bei dieser Version einen Konfigurations-Patch einspielen musste, damit „HAT 2.04“ auch unter SUSE 10.1 kompiliert.

1.3 Installation

1.3.1 SuSE 10.1

Nach der Installation der oben genannten Pakete und des Konfigurations-Patches lässt sich HAT über die folgenden Befehle installieren:

- ./configure
- make
- sudo make install

1.3.2 Kubuntu 6.10 Edgy Eft

Bei dieser Version sind sowohl „ghc-6.4.2“, „ghc-6.4.2-prof“, „hmake“ als auch HAT über den integrierten Paketmanager installierbar und sofort einsetzbar. Unter Umständen müssen die Optionen des Paketmanagers jedoch vorher auf „universe“ und „multiverse“ eingestellt werden.

2 Grundlagen

2.1 Offline Tracing

HAT ist ein Offline Tracer und kein, wie aus imperativen Sprachen bekannter, Debugger, mit dem man Schritt für Schritt durch den Programmablauf gehen kann, um sich zu einem bestimmten Zeitpunkt die Inhalte von Variablen anzeigen zu lassen. Bei funktionalen Sprachen würden hier sehr lange und unter Umständen unausgewertete Ausdrücke stehen. Bei HAT hingegen wird das Programm normal ausgeführt, wobei zusätzlich ein „Trace[1]“ geschrieben wird, das mit einem der in den folgenden Kapiteln vorgestellten Werkzeugen dargestellt werden kann. HAT ist es möglich, normal terminierende, terminierende Programme mit Fehlern, sowie solche Programme, die erst durch Benutzerabbrüche terminieren, zu tracen. Diese Traces bestehen aus High-Level Informationen und beschreiben jede Reduktion von Berechnungen.

2.2 Hmake

Um ein Programm zu tracen, muss es mit Hmake erst übersetzt werden. Hmake benutzt dabei den pre-processor hat-trans, der jedes Modul durch ein neues, tracingfähiges Modul ersetzt. Zu beachten ist, dass das Tracing den benötigten „heap-space“ ca. um den Faktor 3 vergrößert. Die Laufzeit des getraceten Programms ist ca. um den Faktor 50 langsamer und es kann eine Größe von mehreren hundert Megabyte annehmen. Um der höheren Laufzeit und dem größeren Speicheraufkommen entgegen zu wirken, bietet HAT die Möglichkeit des „Trusting“. Hierbei ist es möglich, HAT Funktionen so tracen zu lassen, dass nicht jede Reduktion mit ins Tracing aufgenommen wird. Dazu muss die Option „-trusted“ beim Kompilieren mit übergeben werden. Die Standardbibliotheken und Prelude sind standardmäßig auf „trusted“ gesetzt. Jeder Start, um ein Haskell Programm zu tracen, läuft gleich ab. Zu allererst wird das Programm folgendermaßen kompiliert, ich nenne es hier beispielhaft Hatbsp.hs

```
$ hmake -hat Hatbsp
```

Danach wird das Programm folgendermaßen aufgerufen:

```
$ ./Hatbsp
```

Nun wurden einige Dateien erstellt, die im selben Verzeichnis wie die Datei Hatbsp.hs und in dem erstellten Unterverzeichnis „Hat“ liegen. Jetzt ist es möglich, die verschiedenen Trace-Varianten von HAT zu benutzen, um den Ablauf des Programms nach zu verfolgen.

3 Tracing

Ich werde hier nun die einzelnen Werkzeuge und ihre Hauptverwendung erklären. Allgemein gilt, dass der größte Erfolg bei der Fehlersuche erreicht wird, indem man die einzelnen Werkzeuge zusammen benutzt und sich nicht nur auf einen festlegt.

[1] Ein „Trace“ ist eine Spur, die Informationen über z.B. Auswertungen und Reduktionen enthält.

3.1 Spezielle Ausdrücke und Kommandos

3.1.1 Unausgewertete Ausdrücke

Die verschiedenen Werkzeuge werden normalerweise keine unausgewerteten Unterausdrücke ausgeben, sondern diese mit dem Unterstrich „_“ darstellen. Es existiert allerdings eine „uneval“ Option, die die Ersetzung der Unterstriche mit den unausgewerteten Ausdrücken bewirkt.

3.1.2 Lambda-Abstraktion

Lambda Abstraktionen werden, wie aus Haskell gewohnt, mittels des $\backslash$ dargestellt.

3.1.3 Undefinierte Ausdrücke

Berechnungen, die auf Grund von Laufzeitfehlern oder Benutzerabbrüchen zwar angefangen, aber nicht beendet werden konnten, werden mittels $\bot$ als ASCII Zeichen _|_ dargestellt.

3.1.4 Trusted Ausdrücke

Ausdrücke, die, „trusted“ sind und nicht aufgezeichnet wurden, werden mittels $\{?\}$ dargestellt.

3.1.5 Kommandos

- :set ruft bei einigen Werkzeugen die Parameter auf, die man ändern kann.
- :? ruft die allgemeine Hilfe zu dem aktuellen Werkzeug auf.

3.2 HAT Trail

HAT Trail oder HAT Pfad, wie man es übersetzen würde, ist dafür gedacht, folgende Frage zu beantworten. „Wo kam das her?“ Hierbei kann es sich um Werte, Ausdrücke, Ausgaben oder auch Fehlermeldungen handeln. Die direkte Antwort wird ein übergeordneter Aufruf oder ein Name sein. HAT Trail wird über den Konsolenbefehl „hat-trail“ aufgerufen und liefert uns ein interaktives Tool. Mittels der Pfeiltasten wählt man den Ausdruck, den man zurückverfolgen möchte und mit der Enter-Taste bestätigt man die Zurückverfolgung. HAT Trail liefert dann als Ausgabe den Aufruf, der zur ausgewählten Ausgabe führte. Ist dies zum Beispiel eine Funktion mit mehreren Parametern, so kann man über die Pfeiltasten sowohl die Funktion, als auch jeden einzelnen Parameter aufrufen, um heraus zu finden, wo dessen Ursprung liegt. Es ist jederzeit möglich, mit der Backspace-Taste wieder einen Schritt in der Beobachtung zurück zu gehen. So lässt sich einfach feststellen, an welcher Stelle eines Programms mit falscher bzw. nicht erwarteter Ausgabe der Fehler liegen kann. HAT Trail bietet dazu noch eine Ausgabe der Zeilen- und Spaltennummer von der Stelle, an der das ausgewählte Element reduziert bzw. ausgeführt wurde. Dazu kommen noch zwei Kommandos :source (:s) und :quit (:q). Der „source“ Befehl öffnet „HAT View“, welches ein Quelltextfenster öffnet, um den Bereich anzuzeigen, zu dem das ausgewählte Element instantiiert wurde. Eine Beispielausgabe von

„HAT Trail“ kann wie in Abbildung 1 aussehen. Dies ist ein Beispiel für ein Sortierverfahren und das Wort „program“ soll dabei sortiert werden. Das Wort in Zeile 2 zeigt die Ausgabe des sortierten „program“. Wir wählen diesen aus und erhalten als Antwort in Zeile 5 den Aufruf, der ihn auf die Ausgabe schreiben ließ. Wählt man bei jeder Anfrage den String aus, erhalten wir eine Ansicht, wie hier dargestellt, und können zurückverfolgen, wie das Programm den String „program“ in „agmoprr“ sortierte.

```
Output: ------------------------------------------------
agmoprr\n

Trail: ------- Insort.hs line: 3 col: 25 ---
<- putStrLn "agmoprr"
<- insert 'p' "agmorr" | if False
<- insert 'r' "agmor" | if False
<- insert 'o' "agmr" | if False
<- insert 'g' "amr" | if False
<- insert 'r' "am" | if False
<- insert 'a' "m" | if True
<- insert 'm' []
```

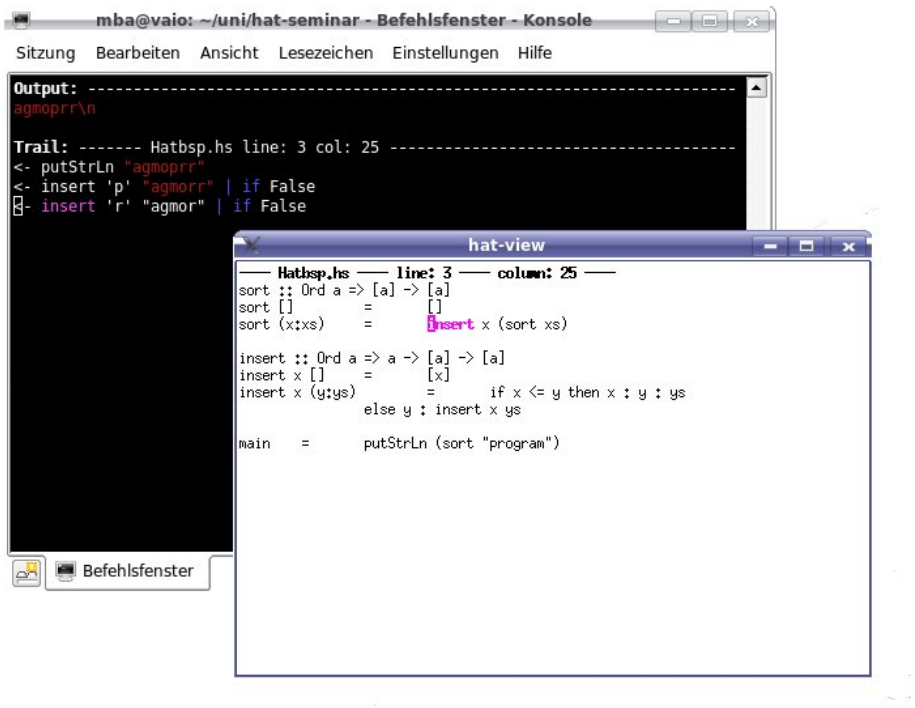

Abbildung 1: HAT Trail Ausgabe und Quellcode Referenz mittels HAT View

3.3 HAT Observe

Mit HAT Observe erhält man Antworten auf die Frage: „Was liefert mir der Aufruf einer bestimmten Funktion?“ HAT Observe wird mittels des Befehls „hat-observe“ gestartet und benötigt nur zwei wesentliche Kommandozeilenbefehle:

:info liefert eine Liste von beobachtbaren Funktionen und definierten Werten, sowie deren Anzahl an Aufrufen, die über das Pattermatching hinaus ausgewertet werden konnten.

:quit beendet die Beobachtung

Die Anzahl an Beobachtungsausgaben ist standardmäßig in Zehner Blöcke unterteilt. Der übliche Weg ist es, mit :info zu beginnen. Ein solcher Aufruf kann wie in Abbildung 2 aussehen. Mittels „:info“ in Zeile 1 werden die beobachtbaren Funktionen und Werte in Zeile 2 angezeigt. Durch die Eingabe von „`sort`“ werden ab Zeile 4 die Auswertungen von „`sort`“ mit verschiedenen Parametern dargestellt. Eine Darstellung über Auswertungen der Funktion „≤“

```
hat-observe> :info                                                    1
19 <=     21 insert     1 main     1 putStrLn     1 sort             2
hat-observe> sort                                                     3
1 sort "program" = "agmoprr"                                          4
2 sort "rogram" = "agmorr"                                            5
3 sort "ogram" = "agmor"                                              6
4 sort "gram" = "agmr"                                                7
5 sort "ram" = "amr"                                                  8
6 sort "am" = "am"                                                    9
7 sort "m" = "m"                                                      10
8 sort [] = []                                                        11
```

Abbildung 2: HAT Observe: Sort Darstellung

findet man in der Abbildung 3.

```
19 <=     21 insert     1 main     1 putStrLn     1 sort
hat-observe> <=
1 'a' <= 'm' = True
2 'r' <= 'a' = False
3 'g' <= 'a' = False
...
```

Abbildung 3: HAT Observe: $<=$ Darstellung

```
19 <=      21 insert      1 main      1 putStrLn      1 sort
hat-observe> insert 'g' _
1 insert 'g' "amr" = "agmr"
2 insert 'g' "mr" = "gmr"
hat-observe> insert _ _ = [_]
1 insert 'm' [] = "m"
2 insert 'r' [] = "r"
hat-observe> sort in main
1 sort "program" = "agmoprr"
```

Abbildung 4: HAT Observe: Einschränkung mittels Muster

3.3.1 Einschränkung durch Muster

Es ist möglich Anfragen in HAT Observe mittels Muster (pattern) einzuschränken. Die Syntax sieht dabei folgendermaßen aus

identifier <pattern>* [= <pattern>] [in <identifier>]

pattern* kann dabei mehrfach oder auch gar nicht auftreten, die folgenden beiden Parameter sind optional. Beispielabfragen sind in Abbildung 4 zu sehen.

3.4 HAT Detect

HAT Detect ist ein interaktives, halbautomatisches Tool zur Auffindung von Fehlerquellen im Programm. Zum Zeitpunkt dieser Arbeit ist es HAT Detect allerdings noch nicht möglich, monadische Bindungen wie die do-Notationen richtig zu erfassen, deshalb funktioniert es bislang nur mit einfachen IO Aktionen wie z.B. putStrLn oder print. Auch ist es nur anwendbar bei Programmen mit fehlerhaftem Output. Die sinnvolle Vorgehensweise ist im Moment die Suche nach einer fehlerhaften Reduktion mittels HAT Observe, um danach HAT Detect darauf anzuwenden. Die Funktionsweise ist relativ simpel. Nach dem Aufruf des zu untersuchenden Programms hat der Benutzer die Möglichkeit mittel yes (y) und no (n) die Korrektheit einer Gleichung anzugeben. Hat das Programm genug Informationen mittels der Benutzereingaben gesammelt, gibt es eine Fehlerbeschreibung aus.[2]

3.4.1 Antworten aufschieben

Ist man sich einer Antwort nicht sicher, bietet HAT Detect die Möglichkeiten an, mit ?n und ?y zu antworten. HAT Detect geht dann davon aus, dass n bzw. y gilt. Findet es aber dann im Folgenden keinen Fehler, wiederholt es diese Frage und man kann den entsprechend anderen Weg wählen.

[2] Siehe auch deklaratives Debugging mittels Buddha

3.4.2 Funktionsweise

HAT Detect benutzt eine Baumstruktur zum Auffinden von Fehlern im Code. Hierbei stellt der Wurzelknoten die Reduktion von main dar. Kinder sind die zu reduzierende Ausdrücke des Rumpfes der Knoten-Funktion. Bewertet man eine Reduktion mit Falsch (n) aus, so werden danach die Kinderknoten überprüft. Ist die Antwort wahr (y), so sucht man nach einem Knoten, der näher an der Wurzel liegt, bzw. nach weiteren Kindern des Vaterknotens. Existiert ein Knoten, der mit Falsch bewertet wurde und dessen Kinder mit Wahr bewertet wurden, hat man eine fehlerhafte Reduktion gefunden.

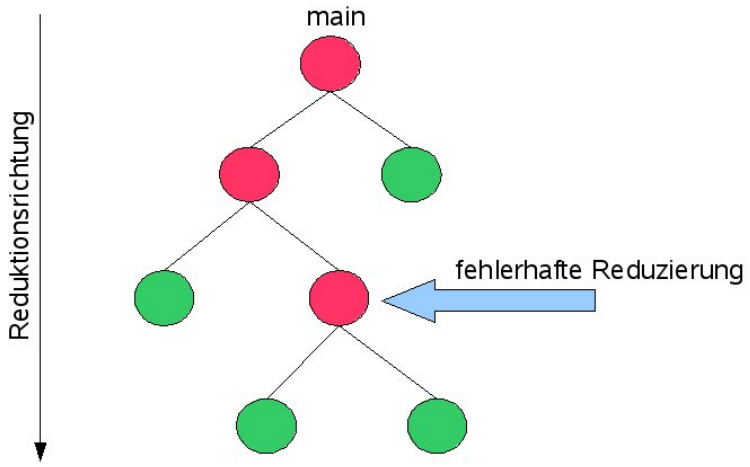

Abbildung 5: HAT Detect: Baumsuche nach Reduktionsfehlern

3.5 HAT Stack

HAT Stack ist ein Werkzeug mit dem man unterbrochene oder mit einem Fehler beendete Programme tracen kann. Es lässt einen virtuellen Stack anzeigen, der an der Spitze den fehlerhaften Ausdruck enthält. Jeder Funktionsaufruf im Stack führt zum Aufruf der direkt darüber liegenden Funktion. Siehe auch Abbildung 7. Der allgemeine Programmaufruf ist `hat-stack <programmname>`.

3.6 HAT Explore

HAT Explore ist ein interaktives Werkzeug zum freien Quellcode-Tracen. Es ist möglich durch die Funktionsaufrufe mittels der Pfeiltasten zu navigieren. Dabei wird die Stelle im Quellcode angezeigt, an der die aktuelle Ausgabe produziert wird. Hinzu wird ein Verlauf der Funktionsaufrufe bis zum aktuellen angezeigt. Auswertungen können sowohl mit Wahr als auch mit Falsch markiert werden und sind dann farblich im Explorer gekennzeichnet. Siehe dazu auch Abbildung 8 in dem Kapitel 3.7.3. Der allgemeine Programmaufruf ist `hat-explore <programmname>`.

3.7 Fehlerhafte Programme tracen

Da der sinnvolle Einsatz von HAT jedoch eher in den Bereich fehlerhafter Programme gehört, betrachten wir nun im Folgenden Programme mit Laufzeitfehlern, fehlerhaften Ausgaben und nicht terminierende Programme an dem Beispiel des fehlerhaften Sortieralgorithmus aus Abbildung 6.

```
sort :: Ord a => [a] -> [a]                                    1
-- Fehler Nr. 1: Fehlende Gleichung für [] als Argument        2
sort (x:xs) = insert x (sort xs)                               3
insert :: Ord a => a -> [a] -> [a]                             4
insert x [] = [x]                                              5
insert x (y:ys) = if x <= y                                    6
                  -- Fehler Nr. 2: y aus dem Ergebnis fehlt    7
                  then x:ys                                    8
                  -- Fehler Nr. 3: gleicher rekursiver Aufruf  9
                  else y: insert x (y:ys)                      10
main = putStrLn (sort "program")                               11
```

Abbildung 6: Fehlerhafter Sortieralgorithmus - badinsort.hs

3.7.1 Programme mit Laufzeit Fehlern

Führen wir dieses Programm aus, bricht es mit dem Fehler

„`Error: No match in Pattern`“

ab. Unsere erste Wahl, um heraus zu finden, „wo der Fehler herkommt“, ist HAT Trail. Durch Auswahl und Bestätigung von „`No match in pattern.`“ liefert uns HAT Trail die Ausgabe „`sort []`“. Der Fehler wird also durch den Aufruf von `sort []` produziert. Schaut man in den Quellcode, erkennt man recht schnell, dass es für `sort []` kein entsprechedes Matching bzw. Muster gibt. Betrachten wir nun, was HAT Observe uns an Informationen bieten kann. Der :info Aufruf liefert folgende Informationen:

```
hat-observe> :info
7+0 insert    1 main    1 putStrLn    1+7 sort
```

Bereits hier ist deutlich zu erkennen, dass Fehler im Programm vorhanden sind. `insert` und `sort` haben einen zusammengesetzten Zähler. Diese Ausgabe in Form von $M + N$ zeigt an, dass M Aufrufe niemals über das Patter-Matching hinauskamen und nicht ausgewertet wurden. N Aufrufe wurden hingegen ausgewertet und durch den Funktionsrumpf ersetzt.[3] Lassen wir uns also anzeigen, was mit den sieben Aufrufen von `insert` passiert ist.

[3] Die Notation M+N scheint in der aktuellen Version nicht mehr vorhanden zu sein. Es ist aber möglich, dass sie wieder eingeführt wird. Deswegen habe ich sie hier mit aufgeführt.

```
hat-observe> insert
1 insert 'p' _|_ = _|_
2 insert 'r' _|_ = _|_
3 insert 'o' _|_ = _|_
4 insert 'g' _|_ = _|_
5 insert 'a' _|_ = _|_
6 insert 'm' _|_ = _|_
```

Wie schon in Kapitel 3.1 erklärt stellt das Symbol $\perp$ einen undefinierten Ausdruck dar. Dadurch, dass das zweite, und zur Auswertung benötigte, Argument von `insert` undefiniert ist, kann auch auch kein Aufruf von `insert` ausgewertet werden. Betrachten wir nochmal den Quellcode aus Abbildung 6 in Zeile 3, sieht man schnell, dass `insert` ein Aufruf von `sort` ist und demzufolge auch seine Argumente von dort beziehen sollte. Betrachten wir nun also den Aufruf von `sort`.

```
hat-observe> sort
1 sort "program" = _|_
2 sort "rogram" = _|_
3 sort "ogram" = _|_
4 sort "gram" = _|_
5 sort "ram" = _|_
6 sort "am" = _|_
7 sort "m" = _|_
8 sort [] = _|_
```

Aus dem Zusammenhang von `sort` und `insert` in Abbildung 6 Zeile 3 erkennt man, dass das zweite Argument von `insert 'p' _|_` der Aufruf von `sort` mit dem Rest-String ist. Diese Beziehung lässt sich dann bis zu `sort []` zurückverfolgen. Kenntniss über den eigenen Code ist hierbei eine grundsätzliche Vorraussetzung. Zur Vollständigkeit zeige ich noch kurz die Ausgaben von putStrLn und main

```
hat-observe> putStrLn
1 putStrLn _|_ = {IO}
hat-observe> main
1 main = {IO}
```

Die Ausgabe in geschweiften Klammern bedeutet hierbei, dass der Wert nicht genauer angezeigt werden kann und bestätigt nur, dass es sich um eine IO Aktion handelt. Als letztes Werkzeug benutzen wir nun noch HAT Stack und schauen uns den Fehlerstack einmal genauer in Abbildung 7 an. Auch hier ist sofort eindeutig zu sehen, dass unser Fehler bei `sort []` liegt. Dazu wird noch angezeigt, wie die Abarbeitungsfolge aussieht, bis es zu dem eigentlichen Fehler kam.

```
Program terminated with error:
        No match in pattern.
Virtual stack trace:
(badinsort.hs:3)          sort []
(badinsort.hs:3)          sort "m"
(badinsort.hs:3)          sort "am"
(badinsort.hs:3)          sort "ram"
(badinsort.hs:3)          sort "gram"
(badinsort.hs:3)          sort "ogram"
(badinsort.hs:3)          sort "rogram"
(badinsort.hs:10)         sort "program"
(unknown)         main
```

Abbildung 7: HAT Stack Darstellung

3.7.2 Nicht terminierende Programme

Korrigiert man nun den Fehler in Abbildung 6 durch Hinzufügen von `sort [] = []`, so erhalten wir ein nicht terminierendes Programm mit einem unendlichen String „aaaa...“. Hat man den Verdacht, dass ein Programm nicht terminiert, so bricht man es mittels Strg-C ab. Je füher abgebrochen wird, desto besser. Betrachten wir nun das Tracing mit HAT Observe

```
hat-observe> :info
16346 insert    1 main    8 sort    16345 <=    1 putStrLn
```

Die immens hohe Anzahl an Aufrufen von `insert` deutet auf einen Fehler in der Implementierung von `insert` hin, der für die nicht-Terminierung des Progammes verantwortlich ist. Betrachten wir die Funktion `insert` nun etwas genauer.

```
hat-observe> insert
1 insert 'p' ('a':_) = "aaaaaaaaaa..."
2 insert 'r' ('a':_) = 'a':_
3 insert 'o' ('a':_) = 'a':_
4 insert 'g' ('a':_) = 'a':_
5 insert 'r' "a" = 'a':_
6 insert 'a' "m" = "a"
7 insert 'm' [] = "m"
8 insert 'r' "a" = _
9 insert 'g' ('a':_) = _
10 insert 'o' ('a':_) = _
11 insert 'r' ('a':_) = _
```

Schon bei der ersten Zeile finden wir den Fehler. Egal wie die Restliste von a aussieht, der Aufruf von `insert` endet in dem unendlichen String. Ersetzen wir die Restliste mit [], so erhalten wir `insert 'p' ''a'' = ''aaaaa...''`, und somit die Information, dass

`insert` falsch implementiert wurde. Wir ersetzen dann den Code aus Abbildung 6 in Zeile 10 durch „`else y : insert x ys`“ und haben den Fehler behoben.

3.7.3 Programme mit fehlerhaftem Output

Rufen wir das Programm nun auf, erhalten wir als Ausgabe „agop“, also haben wir 4 Buchstaben verloren und eine fehlerhafte Ausgabe. In diesem Fall können wir sowohl mit HAT Trail, HAT Observe, HAT Detect als auch HAT Explore den Fehler ausfindig machen. Schauen wir uns zuerst HAT Observe an.

```
hat-observe> insert _ _ = "agop"
1 insert 'p' "agor" = "agop"
```

Wir wenden hier das in Kapitel 3.3.1 vorgestellte Pattermatching an. und suchen die `insert` Aufrufe, die die Endausgabe „agop“ liefern. Das Ergebnis zeigt uns, dass wir das 'r' verlieren, wenn wir das 'p' einfügen. Um das noch genauer zu betrachten beschränken wir das Muster noch weiter und suchen nach allen Aufrufen, die das 'p' einfügen.

```
hat-observe> insert 'p' _ in insert
1 insert 'p' "gor" = "gop"
2 insert 'p' "or" = "op"
3 insert 'p' "r" = "p"
```

Es ist leicht zu erkennen, dass der Fehler in Zeile 3 liegt. Nehmen wir uns nun den Fehler mit HAT Trail vor.

```
Output: ------------------------------------------------------
agop\n

Trail: ------- badinsort.hs line: 7 col: 19 ----------
<- putStrLn "agop"
<- insert 'p' "agor" | if False
```

Auch hier ist zu erkennen, dass wir beim `insert` von 'p' das 'r' aus "agor" verlieren. Insgesamt scheint in diesem Fall aber das Tracing mit HAT Observe genauer zu sein. Betrachten wir nun das Programm mit HAT Detect.

```
        hat-detect 2.04 (:? for help, :q to quit)

1  sort "program" = "agop"
hat-detect> n
2  sort "rogram" = "agor"
hat-detect> n
3  sort "ogram" = "ago"
hat-detect> n
4  sort "gram" = "ag"
hat-detect> n
```

```
5  sort "ram" = "ar"
hat-detect> n
6  sort "am" = "a"
hat-detect> n
7  sort "m" = "m"
hat-detect> y
8  insert 'a' "m" = "a"
hat-detect> n
Bug found! Faulty reduction:
    insert 'a' "m" = "a"
Done.
```

Nach 8 Benutzereingaben über die Korrektheit der Gleichungen liefert HAT Detect uns einen Fehler im Code von `insert`. Wir sehen also, dass HAT Observe und HAT Detect den Fehler bis zum `insert` etwas genauer zurückverfolgen konnten als HAT Trail. Aber auch HAT Explore kann hierbei sehr nützlich sein. Öffnen wir den Explorer mittels hat-explore, navigieren wir mit den Pfeiltasten durch die Funktionsaufrufe und markieren inkorrekte Auswertungen mittels 'w'. Abbildung 8 zeigt dann nach einer gewissen Anzahl an Abfragen den Fehler im Quelltext mittels der roten Markierung. Und auch hier ist der Fehler bei `insert` gefunden.

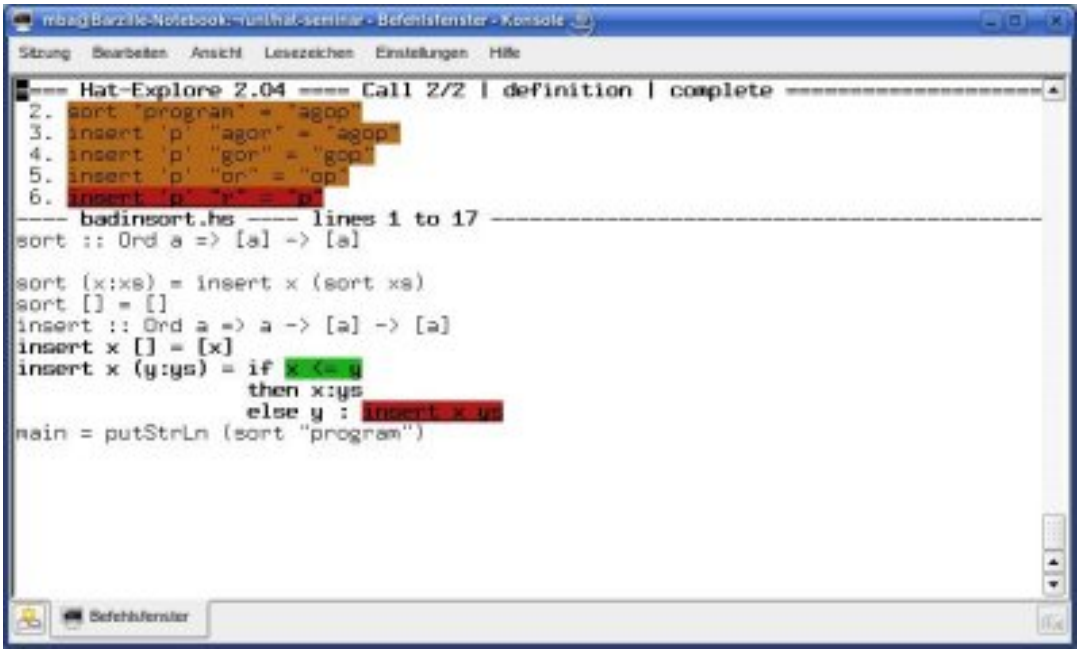

Abbildung 8: HAT Explore Darstellung

4 Zusammenfassung

Die vielfältigen Möglichkeiten, die der Haskell Tracer anbietet, sind eindeutig positiv zu bewerten. Selbst für Neulinge auf dem Gebiet des Tracens bzw Debuggens sind die verschiedenen Werkzeuge schnell und einfach zu verstehen und zu bedienen. Zudem bietet HAT die Möglichkeit, verschiedene Fehlerquellen zu finden. Von Ausgabefehlern über Laufzeitfehler bis hin zu nicht-terminierenden Programmen ist dieser Tracer einsatzbereit. Als Manko muss man allerdings die Geschwindigkeit und den Speicherbedarf der getraceten Dateien anführen. Selbst das relativ kurze Programm aus Abbildung 6 liefert im getraceten Zustand bereits einen Speicherplatzbedarf, der um einen Faktor 5 größer ist als der des Quelltextes.

4.1 Probleme

HAT funktioniert zur Zeit nicht auf allen 64 Bit Systemen. Laut der Mailingliste handelt es sich entweder um Probleme im eigenen Code bezüglich Knotenreferenzen (C Ints vs. 4Byte Typen) oder um Probleme bezüglich des Systems (big-endian Architektur vs. little-endian). Denn auf 64 Bit MacOS und PowerPC Systemen scheint HAT zu funktionieren.

Literatur

[1] www.haskel.org/hat

[2] Hat Tutorial: Part 1, The ART Team, 2002

[3] Hat - The Haskell Tracer: Version 2.04 Users' Manual, The ART Team, 2005

[4] Hat-Explore: Version 2.04 Users' Manual, Olaf Chitil, 2005