МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет»

Институт математики, механики и компьютерных наук им. И.И. Воровича

Мирзоев Денис Шамширович

ИССЛЕДОВАНИЕ ДЕТАЛЕЙ РЕАЛИЗАЦИИ ОТЛАДЧИКА GHCI

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению 02.03.02 – Фундаментальная информатика и информационные технологии

Научный руководитель –
ст. преп. каф. ИВЭ В.Н. Брагилевский

Ростов-на-Дону – 2018

Содержание

Введение

С каждым днём усиливается роль информатизации в жизни общества. Уже трудно представить образование, культуру, производство и другие области человеческой деятельности вне связи с информационными технологиями.

Современные информационные системы способны производить огромное количество вычислений в кратчайшие сроки. Такая производительность позволяет решать важные прикладные задачи: прогнозирование процессов, различные задачи оптимизации. Каждый день люди пользуются достижениями информатики, изучая прогноз погоды, запрашивая оптимальный маршрут на карте, оплачивая товары и услуги банковской картой, общаясь в сети Интернет, делая заказы из любой точки планеты. Всё это это стало возможным благодаря широкому проникновению информатики в производство и экономику.

Информатизация общества является общемировой тенденцией. Всё больше требуется специалистов в этой сфере, возникают всё более сложные задачи.

Важнейшую роль в процессе информатизации общества играет сеть Интернет. К середине 2015 года число пользователей достигло 3,3 млрд человек. Люди получили доступ к огромному объёму информации и возможность мгновенно ей обмениваться.

Программное обеспечение позволяет использовать возможности компьютера наиболее эффективным образом. Оно помогает специалистам решать задачи быстрее и качественнее. В этом случае специалисты в сфере информационных технологий не являются исключением. Они также используют программное обеспечение в своей работе.

Программное обеспечение принято делить на свободное и проприе-

тарное, то есть несвободное. Существует несколько критериев свободного программного обеспечения. Тем не менее, среди специалистов нет единодушия по этому поводу. К примеру, Ричард Столлман выделяет такие критерии:

- Свобода запускать программу;
- Свобода изучать устройство программы и изменять её поведение, то есть иметь доступ к её исходному коду;
- Свобода распространять копии программы;
- Свобода распространять копии программы с изменениями.

В зависимости от реализации этих свобод программу можно отнести к той или иной степени проприетарности.

Существуют различные лицензии программного обеспечения, которые регламентируют правовые аспекты взаимодействия с программным продуктом. К свободным можно отнести Apache, GNU GPL, BSD, Creative Commons. Проприетарные лицензии обычно создаются индивидуально для каждого проекта.

В данной работе исследуется компилятор GHC. Он доступен под свободной лицензией BSD. Она допускает безграничное распространение для любых целей, но требует сохранения текста лицензии во всех копиях, а также ограничивает использование имён разработчиков, вносящих вклад в этот проект.

В данной работе мы исследуем детали реализации отладчика языка Хаскель, встроенного в интерпретатор GHCi. Его исходный код находится в удалённом git репозитории[1].

Отладка – это процесс устранения ошибок в программе. Для его упрощения разработки используют специальные прогрммы-отладчики.

Они обладают обширным набором средств контроля за выполнением программы.

Хотя все отладчики различаются с точки зрения функциональности, можно выявить общие принципы работы, присущие большинству из них. К примеру, часто бывает удобно остановить работу программы в месте предполагаемой ошибки и посмотреть значения доступных в этот момент переменных. Кроме того, отладчики позволяют выполнять программу “шаг за шагом“, то есть выполнять каждый раз лишь одну строку программы. Это помогает после остановки проследить изменение значений переменных и саму последовательность выполнения инструкций.

Особый интерес заключается в том, что Хаскель является языком программирования с ленивой стратегией вычисления. Это означает, что порядок вычисления не задан явно. Продемонстрируем эту особенность на двух примерах.

Листинг 0.1 Делим два числа на Си

```
1 int divFooBar(void) {
2     int foo, bar;
3     foo = ...;
4     bar = ...;
5     return foo / bar;
6 }
```

В листинге 0.1 приведён пример на языке Си. Последовательность выполнения `divFooBar` будет такой и только такой: сначала вычислится значение `foo`, потом вычислится `bar`, потом вычислится результат.

Листинг 0.2 Делим два числа на Хаскель

```
1 divFooBar =
2     let foo = ...
3         bar = ...
4     in foo `div` bar
```

Код в листинге 0.2 делает то же самое. Разница заключается в ленивости. Значения в Хаскель не вычисляются сразу. Вычисление выражения начинается только в тот момент, когда понадобится его результат. В связи с этим становится трудно сказать, в каком порядке будет выполняться программа. `foo` и `bar` могут быть вычислены в любом порядке.

По этой причине GHCi имеет сложное устройство и не похож на отладчики традиционных языков программирования.

1 Отладка средствами GHC

GHCi – это интерпретатор Хаскель кода. Он является частью GHC, компилятора языка Хаскель. GHCi позволяет вычислять значения и интерпритировать код. Кроме того, в его состав входит отладчик.

1.1 Команды отладчика

Полный список команд GHCi можно найти в руководстве пользователя[5]. Нас интересуют только команды связанные с отладкой.

- `:break` устанавливает точку останова;
- `:force` и `:print` показывают значение выражения. Разница между ними в том, что `:force` принудительно вычисляет его значение, а

`:print` нет.

- `:history` показывает последние выполненные шаги вычисления;
- `:forward` и `:back` позволяют перемещаться по истории вычислений вперед и назад соответственно;
- `:trace` включает историю вычислений;
- `:step` производит один шаг вычисления;
- `:continue` продолжает выполнение вычисления после остановки.

1.2 Пример работы с отладчиком GHCi

Рассмотрим простую программу, осуществляющую сортировку списка целых чисел алгоритмом быстрой сортировки.

Листинг 1.1 main.hs

```
1 qsort [] = []
2 qsort (a:as) = qsort left ++ [a] ++ qsort right
3  where
4   left  = filter (<=a) as
5   right = filter (a<) as
6
7 main = print (qsort [8, 4, 0, 3, 1, 23, 11, 18])
```

Загрузим её в интерпретатор.

Листинг 1.2

```
$ ghci main.hs
Ok, modules loaded: Main.
*Main>
```

Теперь можно добавить точку останова. Сделаем это на второй части определения `qsort`.

Листинг 1.3

```
*Main> :break 2
Breakpoint 0 activated at main.hs:2:16-47
```

Команда `:break 2` устанавливает точку останова на строке 2 последнего загруженного модуля. В нашем случае это `main.hs`. Она выбирает самое левое полное подвыражение на этой строке(`qsort left ++ [a] ++ qsort right`).

Теперь можно запустить программу.

Листинг 1.4

```
*Main> :main
Stopped in Main.qsort, main.hs:2:16-47
_result :: [Integer] = _
a :: Integer = 8
left :: [Integer] = _
right :: [Integer] = _
[main.hs:2:16-47] *Main>
```

Выполнение приостановлено на точке останова. Подсказка ввода изменилась, демонстрируя, что мы остановились и где мы остановились: `main.hs:2:16-47`. Помимо этого выводится список доступных переменных. Здесь можно заметить специальную переменную `_result`, которая связана с результатов текущего выражения. Для отображения значения переменной есть команда `:print`.

Листинг 1.5

```
[main.hs:2:16-47] *Main> :print left
left = (_t1::[Integer])
```

Заметим, что `left` не был вычислен, поэтому в его отображении отсутствует полный список значений. Можно потребовать его вычисления.

Для этого существует команда `:force`.

Листинг 1.6

```
[main.hs:2:16-47] *Main> :force left
left = [4,0,3,1]
```

Стоит отметить, что вычисление подвыражения может не завершаться или завершаться с ошибкой даже в том случае, когда вычисление выражение целиком завершается успешно. Рассмотрим пример.

Листинг 1.7 main.hs

```
1 a = 0
2 b = 1 `div` a
3 foo = if a == 0 then 0 else b
```

Результатом вычисления foo будет ноль, а вовсе не ошибка, как можно подумать, заметив деление на ноль в строке 2. Это происходит благодаря ленивой стратегии вычисления. Значение `b` не вычисляется, потому что его результат не требуется для вычисления `foo`.

1.3 История вычислений

В традиционных отладчиках существует понятие стека вызова функций. После остановки в некотором месте программы он помогает понять, откуда мы туда пришли.

Стек состоит из адресов возврата функций, которые вычисляются в данный момент, в порядке обратном порядку их вызова. По нему можно перемещаться, выясняя значения локальных переменных в этих функциях.

Похожий доступ к стеку можно получить и в Хаскель, но это требует компиляции с опцией `-prof`, которая включает оснащение кода систе-

мой профилирования. Такой режим компиляции несовместим с отладчиком.

Существует альтернатива — история вычислений. Выражения в Хаскель вычисляются по мере необходимости, поэтому важнее стека вызовов может оказаться именно последовательность предпринятых шагов вычисления. Историю таких шагов можно получить с помощью команд `:trace` и `:hist`. Рассмотрим пример.

Листинг 1.8

```
*Main> :list qsort
1  qsort [] = []
2  qsort (a:as) = qsort left ++ [a] ++ qsort right
*Main> :b 1
Breakpoint 0 activated at main.hs:1:12-13
```

Запустим сортировку небольшого списка с трассировкой с помощью команды `:trace`.

Листинг 1.9

```
*Main> :trace qsort [3, 2, 1]
Stopped in Main.qsort, main.hs:1:12-13
_result :: [a] = _
```

Теперь мы можем посмотреть историю шагов вычисления. Для этого есть команда `:hist`.

Листинг 1.10

```
[main.hs:1:12-13] *Main> :hist
-1  : qsort:left (main.hs:4:12-26)
-2  : qsort (main.hs:2:16-25)
-3  : qsort (main.hs:2:16-47)
-4  : qsort:left (main.hs:4:12-26)
-5  : qsort (main.hs:2:16-25)
-6  : qsort (main.hs:2:16-47)
-7  : qsort:left (main.hs:4:12-26)
-8  : qsort (main.hs:2:16-25)
-9  : qsort (main.hs:2:16-47)
<end of history>
```

Для перемещения по истории используются команды `:forward` и `:backward`. Следом удобно использовать команду `:list`, которая отображает исходный код места, на котором мы остановились и выделяет вычисляемое выражение.

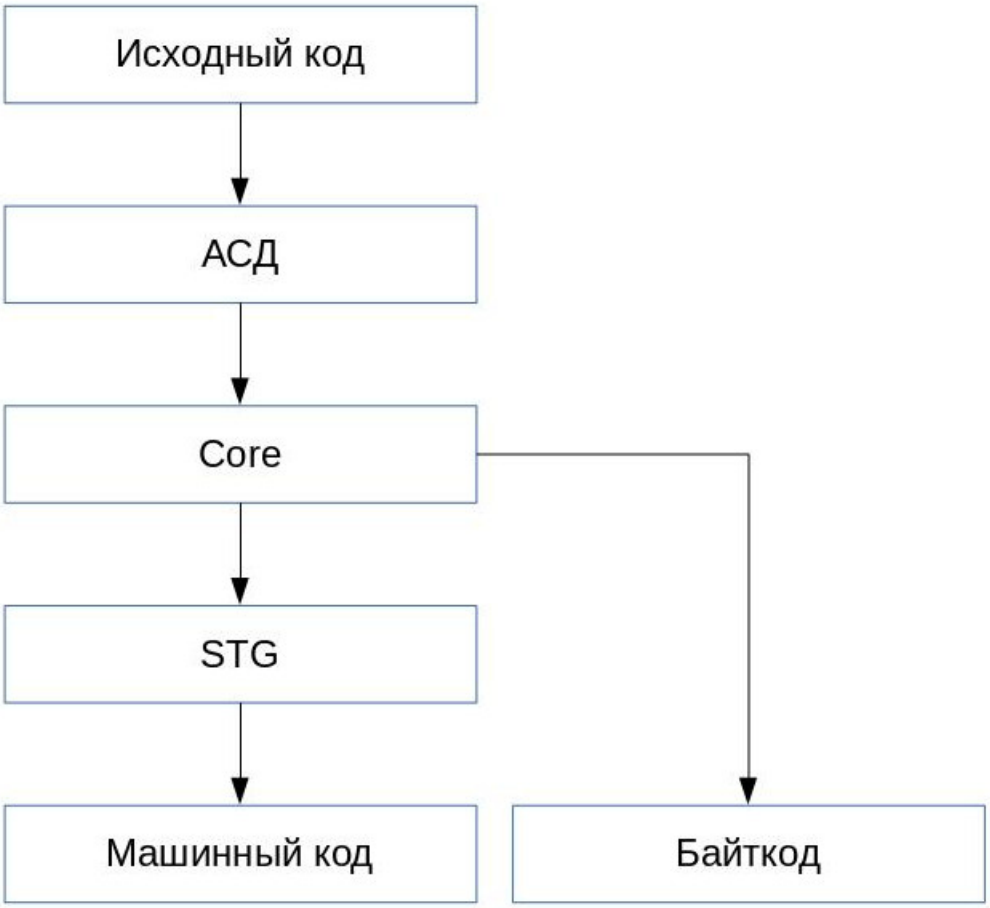
Листинг 1.11

```
[main.hs:1:12-13] *Main> :back
Logged breakpoint at main.hs:4:12-26
_result :: [Integer]
a :: Integer
as :: [Integer]
[-1: main.hs:4:12-26] *Main> :list
3   where
4     left  = >>> filter (<=a) as <<<
5     right = filter (a<) as
[-1: main.hs:4:12-26] *Main> :forward
Stopped at main.hs:1:12-13
_result :: [a]
[main.hs:1:12-13] *Main> :list
1  qsort [] = >>> [] <<<
2  qsort (a:as) = qsort left ++ [a] ++ qsort right
```

2 Особенности процесса компиляции

Подробно процесс компиляции GHC рассматривается в книге The Architecture of Open Source Applications[6]. Мы приведём лишь общую схему.

Рис. 1: Схема процесса компиляции



На рисунке 2 приведена схема процесса компиляции. Рассмотрим её более подробно. В прямоугольниках обозначена форма представления программы на текущем этапе.

2.1 Машинный код

При компиляции отдельного файла на выходе получается исполняемый или объектный файл, то есть машинный код.

На вход компилятору поступает файл с исходным кодом. Он рассматривается как последовательность символов.

Эту последовательность символов обрабатывает лексический анализатор Alex и превращает её в последовательность лексем. Последовательность лексем обрабатывает синтаксический анализатор Happy и превращает её в абстрактное синтаксическое дерево. Далее происходит проверка типов.

Следующий этап — удаление синтаксического сахара. Хаскель транслируется в промежуточный язык Core. Он имеет гораздо более простой синтаксис. На этом шаге происходят многочисленные оптимизации. Использование компактного промежуточного языка позволяет упростить код связанный с оптимизацией и сделать его в некоторой мере независимым от синтаксиса самого языка.

Последний шаг — кодогенерация. Прежде всего Core транслируется в язык STG. STG код транслируется в C--. Это низкоуровневый императивный язык, который используется в качестве переносимого ассемблера. Он может быть скомпилирован напрямую в машинный код, а может быть транслирован в ещё одно промежуточное представление: llvm или C. Это обеспечивает более высокий уровень оптимизации.

2.1.1 Немного об STG

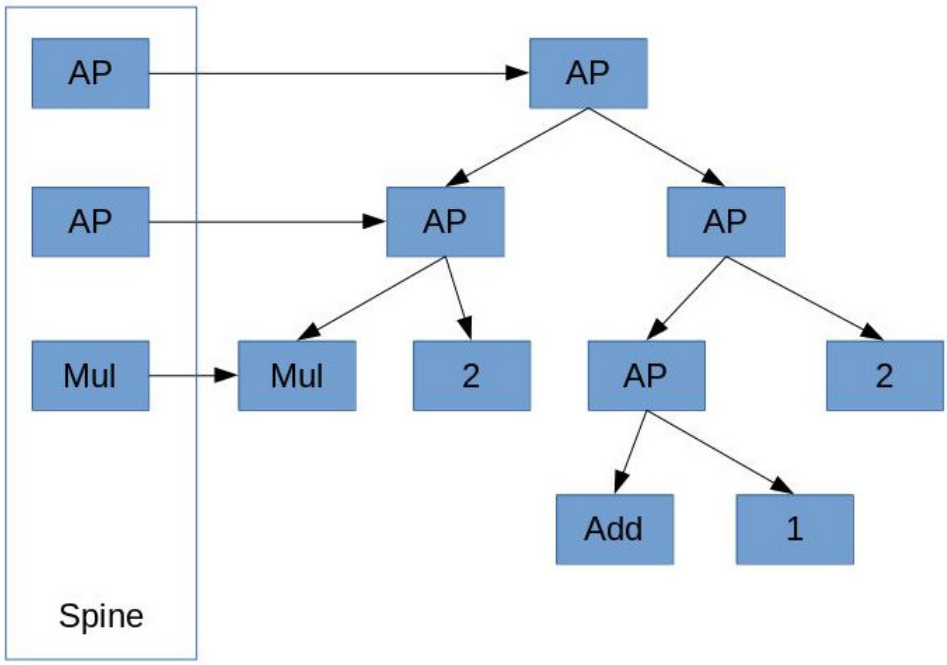
STG это абстрактная машина для свёртки графов(Simon L Peyton Jones [2]). Упрощённое описание STG можно найти в Учебнике по Haskell[3].

STG была разработана специально для компиляции функциональных

языков в низкоуровневое машинное представление. Её язык оперирует понятиями, которые легко отобразить на существующие процессоры. Среди них стек, куча, регистры.

STG расшифровывается как Spineless Tagless G-machine.

- **Spineless** означает, что у этой машины отсутствует spine. Spine – это стек указазателей на самые левые узлы дерева.



- Структура объектов, которыми оперирует STG, похожа. Все они имеют первым полем указатель на код, который нужно выполнить, чтобы вычислить их значение. Поэтому для не нужно знать точный тип вычисляемого объекта.

  В структуре объектов отсутствует поле, обозначающее тип, то есть tag. Отсюда **tagless** в названии.

- **G-machine** это сокращение от Graph reduction machine, то есть машина графовых свёрток. Здесь граф — это выражение на функциональном языке, а свёртка — это процесс его вычисления.

2.2 Байткод

Интерпретатор использует другой выходной формат. На выходе от компилятора он требует Byte Code Object(BCO). В отличие от стандартной кодогенерации, входом которой служит код C--, входом генератора байткода служит Core.

Листинг 2.1 compiler/llvm/LlvmCodeGen.hs

```
44 llvmCodeGen :: DynFlags -> Handle -> UniqSupply
45                 -> Stream.Stream IO RawCmmGroup ()
46                 -> IO ()
```

Листинг 2.2 compiler/nativeGen/AsmCodeGen.hs

```
108 nativeCodeGen :: DynFlags -> Module -> ModLocation
      ↪ -> Handle -> UniqSupply
109                -> Stream IO RawCmmGroup ()
110                -> IO UniqSupply
```

Листинг 2.3 compiler/ghci/ByteCodeGen.hs

```
81 byteCodeGen :: HscEnv
82              -> Module
83              -> CoreProgram
84              -> [TyCon]
85              -> Maybe ModBreaks
86              -> IO CompiledByteCode
```

BCO – это один из объектов, которыми оперирует STG. Он представляет из себя массив инструкций байткода. Интерпретацией этих объектов занимается среда времени выполнения. Интерпретатор во время своей работы взаимодействует со средой времени выполнения, для исполнения своих команд.

Листинг 2.4 rts/Interpreter.c

```
295 Capability *
296 interpretBCO (Capability* cap)
297 {
```

3 Архитектура отладчика GHCi

3.1 Основные локации

- **`ghc/GHCi`**

  В папке ghc/GHCi находится код реализующий взаимодействие с пользователем: распознавание и выполнение функций соответствующих команд.

- **`libraries/ghci`**

  Содержит:

  - Типы для межпроцессной обработки значений
  - Протокол взаимодействия
  - Реализацию сообщений
  - Реализацию расширения Template Haskell для отладчика

- **`iserv`**

  Процесс-сервер отладчика. Код довольно простой. Большая часть реализации содержится в libraries/ghci

- **`compiler/ghci`**

  Модуль compiler/ghci предоставляет интерфейс сервера, который используется остальным кодом.

- **`rts/Interpreter.c`**

  Содержит код системы времени выполнения, отвечающий за исполнение байткода. Функция `interpretBCO`.

- **`include/rts/Bytecodes.h`**

  Содержит номера байткод инструкций.

Листинг 3.1 include/rts/Bytecodes.h

```
26 #define bci_STKCHECK            1
27 #define bci_PUSH_L              2
28 #define bci_PUSH_LL             3
29 #define bci_PUSH_LLL            4
30 #define bci_PUSH8                       5
31 #define bci_PUSH16                      6
32 ...
```

3.2 Выполнение

Рассмотрим что происходит, когда мы запускаем интерпретатор. Во первых отметим, что ghci не является отдельной от компилятора утилитой. Это скрипт, который запускает компилятор, то есть ghc, с параметром --interactive.

Листинг 3.2

```
user@desktop:~$ which ghci
/usr/bin/ghci
user@desktop:~$ cat /usr/bin/ghci
#!/bin/sh
exec "/usr/bin/ghc-8.0.1" --interactive "$@"
```

Теперь обратимся к коду. Функция `main`, с которой начинается выполнение ghc, находится в файле ghc/Main.hs. Там происходит анализ пара-

метров командной строки. Если среди параметров есть `--interactive`, то вызывается функция `interactiveUI`.

Главной монадой в ghc является монада Ghc. Она позволяет осуществлять ввод-вывод и обеспечивает доступ к состоянию компилятора: флаги настроек, граф модулей и другие.

Листинг 3.3 compiler/main/GhcMonad.hs

```
newtype Ghc a = Ghc { unGhc :: Session -> IO a }
```

Модули компилятора, которые требуют больше возможностей расширяют эту монаду. Интерпретатор пользуется монадой GHCi. Она помимо возможностей монады Ghc обеспечивает доступ к состоянию интерпретатора: загруженные модули, подсказка ввода, количество введённых строк и другие.

Листинг 3.4 ghc/UI/Monad.hs

```
newtype GHCi a
    = GHCi { unGHCi :: IORef GHCiState -> Ghc a }
```

`interactiveUI` настраивает стандартные буферы для интерактивной работы и запускает `runGHCi`. Её результат передаётся аргументом для `startGHCi`, потому что значение результата имеет тип `GHCi ()`, а `interactiveUI` возвращает тип `Ghc ()`.

Листинг 3.5 ghc/GHCi/UI.hs

```
400 interactiveUI
401    :: GhciSettings
402    -> [(FilePath, Maybe Phase)]
403    -> Maybe [String]
404    -> Ghc ()
405 interactiveUI config srcs maybe_exprs = do

...

454      startGHCi (runGHCi srcs maybe_exprs)
455          GHCiState{ progname = default_progname,
456                     args     = default_args, ... }
```

Для взаимодействия с пользователем ghc использует библиотеку haskeline. Она похожа на библиотеку realine для си. Ввод строк происходит в монаде InputT. `runGHCi` запускает read-eval-print loop, который работает в этой монаде.

Листинг 3.6 ghc/GHCi/UI.hs

```
892 -- | The main read-eval-print loop
893 runCommands :: InputT GHCi (Maybe String) -> InputT
      ↪ GHCi ()
894 runCommands gCmd = runCommands' handler Nothing gCmd
      ↪  >> return ()
```

`runCommands'` вызывает `runOneCommand`, которая вызывает `doCommand`. Если ввод начинается с двоеточия `doCommand` распознаёт команду и запускает её обработчик. В противном случае считается, что пользователь ввёл выражение.

Команда Если введённая строка начинается с восклицательного знака, то в этом случае её остальная часть рассматривается как команда шел скрипта.

Листинг 3.7 ghc/GHCi/UI.hs

```
1204 specialCommand :: String -> InputT GHCi Bool
1205 specialCommand ('!':str) = lift $ shellEscape (
       ↪ dropWhile isSpace str)

...

1221 shellEscape :: String -> GHCi Bool
1222 shellEscape str = liftIO (system str >> return False
       ↪ )
```

Если строка не начинается с восклицательного знака, то из неё выделяется первое слово. По нему происходит поиск в списке команд и макросов.

Листинг 3.8 ghc/GHCi/UI.hs

```
1206 specialCommand str = do
1207   let (cmd,rest) = break isSpace str
1208   maybe_cmd <- lift $ lookupCommand cmd
1209
1210   case maybe_cmd of
1211     GotCommand cmd -> (cmdAction cmd) (dropWhile
         ↪ isSpace rest)
```

Активные макросы и команды доступны в монаде `GHCi`. Список макросов изначально пуст. Пользователь может определить свои макросы с помощью команды `:def`. Список команд доступных в интерпретаторе по умолчанию находится в переменной `ghciCommands`, которая определена в этом же файле. В этом списке помимо названий команд находятся соответствующие им функции.

Код Если пользователь ввёл код, работает функция `runStmt`.

Листинг 3.9 ghc/GHCi/UI.hs

```
1080 runStmt :: String -> SingleStep -> GHCi (Maybe GHC.
       ↪ ExecResult)
1081 runStmt stmt step = do
1082   dflags <- GHC.getInteractiveDynFlags
1083   if | GHC.isStmt dflags stmt    -> run_stmt
1084      | GHC.isImport dflags stmt  -> run_import
1085      | otherwise                 -> run_decl
```

Код бывает трёх видов: это может быть просьба загрузить модуль(`import`), объявление переменной или функции(`declaration`), выражение(`statement`).

Рассмотрим подробно случаи объявления и выражения.

Объявление В случае объявления работа продолжается в монаде `GHC`. Вызывается функция `runDeclsWithLocation`.

Листинг 3.10 compiler/main/InteractiveEval.hs

```
199 runDeclsWithLocation :: GhcMonad m => String -> Int
      ↪ -> String -> m [Name]
200 runDeclsWithLocation source linenumber expr =
201 ...
202     (tyThings, ic) <- liftIO $ hscDeclsWithLocation
          ↪ hsc_env expr source linenumber
203 ...
```

`hscDeclsWithLocation` выполняет все стадии компиляции: лексический и синтаксический анализ, разрешение имён, проверка типов, удаление синтаксического сахара, оптимизация и кодогенерация, а также линковка. Она возвращает новый интерактивный контекст и новые доступные идентификаторы: функции, переменные, геттеры структур, методы классов.

Выражение В случае выражения работает функция `execStmt`. Она сначала компилирует, а затем выполняет введённый пользователем код.

Листинг 3.11 compiler/main/InteractiveEval.hs

```
-- | Run a statement in the current
     interactive context.
execStmt
  :: GhcMonad m
  => String -- ^ a statement (bind or expression)
  -> ExecOptions
  -> m ExecResult
execStmt stmt ExecOptions{..} = do
...
    -- compile to value (IO [HValue]), don't run
    r <- liftIO $
        hscStmtWithLocation hsc_env' stmt
                            execSourceFile
                            execLineNumber
...
    evalStmt hsc_env' (isStep execSingleStep) (
      ↪ execWrap hval)
```

В конце концов управление переходит в функцию `sandboxIO`.

Листинг 3.12 libraries/ghci/GHCi/Run.hs

```
sandboxIO :: EvalOpts -> IO a -> IO (EvalStatus a)
sandboxIO opts io = do
  -- We are running in uninterruptibleMask
  breakMVar <- newEmptyMVar
  statusMVar <- newEmptyMVar
  withBreakAction opts breakMVar statusMVar $ do
    measureAlloc $ tryEval $ rethrow opts
                 $ clearCCS io
```

`MVar` это изменяемая переменная, одной из функций которой служит синхронизация потоков. Она может находиться в двух состояниях: пустая или хранит значение. При помещении в неё значения, если она не

пустая, поток будет заблокирован, пока другой поток не опустошит её. Также блокировка происходит при изъятии, если в переменной нет значения.

sandboxIO создаёт новый поток, запускает в нём введённый код и начинает ждать, когда в переменной statusMVar появится статус выполнения. Статус это либо завершение, либо остановка на точке останова.

В момент, когда затрагивается точка останова, поток, выполняющий код пользователя, заполняет изменяемую переменную statusMVar значением EvalBreak и пытается достать значение из переменной breakMVar. Первое позволяет разблокировать поток ghci, второе даёт возможность потоку ghci продолжить выполнения с места остановки.

3.3 Роль системы времени выполнения

Во время удаления синтаксического сахара в АСД внедряются специальные узлы HsTick. Это необходимо для установки соответствия между кодом на последующих этапах и исходным кодом программы. Они используются для проверки покрытия кода тестами и в отладчике для реализации точек остановки. Все выражения, подвыражения и сами объявления оборачиваются в этот конструктор.

При компиляции HsTick добавляется специальная инструкция байткода BRK_FUN. Код интерпретации данной инструкции находится в файле rts/Interpreter.c в функции interpretBCO. Когда выполнение доходит до этой инструкции, происходит следующее. Интерпретатор проверяет активна ли точка останова соответствующая этой инструкции. Информация об этом хранится в специальном массиве бит внутри BCO. У инструкции BRK_FUN есть параметр – индекс в этом массиве. Установка соответствующего бита активирует точку останова.

Если точка неактивна, то выполнение продолжается. В противном случае выполняется код, который использует изменяемые переменные из `sandboxIO`.

3.4 Ошибка в работе отладчика

В рамках исследования реализации отладчика мы обратили внимание на неполадку в его работе(#8316, [4]). Она заключается в том, что отладчик аварийно завершается при попытке вычислить значение определённой переменной после остановки.

Для демонстрации ошибки рассмотрим файл Test.hs:

Листинг 3.13 Test.hs

```
1 foo :: [Int]
2 foo = [1..]
```

Загрузим его в интерпретатор

Листинг 3.14

```
ghci Test.hs
```

Поставим точку останова на `foo`, запустим его вычисление и напечатаем с помощью команды `:print`.

Листинг 3.15

```
*Main> :break foo
Breakpoint 0 activated at main.hs:2:7-11
*Main> foo
Stopped in Main.foo, main.hs:2:7-11
_result :: [Int] = _
[main.hs:2:7-11] *Main> :print foo
foo = (_t1::[Int])
```

Теперь попробуем вычислить значение `_t1`.

Листинг 3.16

```
[main.hs:2:7-11] *Main> _t1
```

Здесь возникает ошибка и интерпретатор аварийно завершается.

Листинг 3.17 Ошибка

```
<interactive>: internal error: TSO object entered!
    (GHC version 8.5.20180302 for
      ↪ x86_64_unknown_linux)
    Please report this as a GHC bug:  http://www.
      ↪ haskell.org/ghc/reportabug
[1]    5445 abort (core dumped)  ghci Test.hs
```

3.4.1 Место падения

Изучение причин возникновения данной ошибки привело нас к функции `cvObtainTerm` из файла `compiler/ghci/RtClosureInspect.hs`. Во время выполнения информация о типах удаляется, но отладчику она нужна. Эта функция производит анализ значения в памяти с целью восстановления его типа.

В приведённом примере происходит следующее. В момент остановки программа предоставляет отладчику информацию о доступных именах и их адресах, но она не может предоставить их тип. Отладчик запускает функцию `cvObtainTerm`, чтобы определить типы полученных имён. Это происходит на основе байткода, который вычисляет значение, соответствующее имени, и контекста из уже известных типов. Необходимо понимать, что полную информацию о типе таким образом восстановить невозможно.

По всей видимости, анализ типа переменной `_t1` проходит неправильно. `_t1` связывается с вычисляемым в данный момент значением. На время вычисления указатель на вычисляющий код заменяется на так называемую чёрную дыру. Механизм чёрных дыр был описан в статье Tail recursion without space leaks, Richard Jones[7]. Он решает проблему утечки памяти.

Листинг 3.18 compiler/ghci/RtClosureInspect.hs

```
644 -- Type & Term reconstruction
645 ------------------------------
646 cvObtainTerm :: HscEnv -> Int
647               -> Bool -> RttiType
648               -> HValue -> IO Term
649 cvObtainTerm
650     hsc_env max_depth force
651     old_ty hval = runTR hsc_env $ do
652 ...
653 -- Blackholes are indirections iff the payload
654 -- is not TSO or BLOCKING_QUEUE.
655 -- So we treat them like indirections; if the
656 -- payload is TSO or BLOCKING_QUEUE,
657 -- we'll end up showing '_' which is
658 -- what we want.
659   Blackhole -> do
660     traceTR (text "Following a BLACKHOLE")
661     appArr (go max_depth my_ty old_ty) (ptrs clos) 0
662 -- We always follow indirections
663   Indirection i -> do
664     traceTR (text "Following an indirection" )
665     go max_depth my_ty old_ty $! (ptrs clos ! 0)
```

Комментарий находящийся рядом с нужной веткой гласит, что чёрные дыры можно обрабатывать таким же образом, как и перенаправления, если у них внутри не находится `TSO`(Thread State Object) или `BLOCKING_QUEUE`. Общее описание этих и других объектов можно найти в комментариях проекта[8].

Проверка показала что у чёрной дыры `_t1` внутри `TSO`. При попытке вычислить значение происходит переход по адресу `TSO`, но там отсутствуют инструкции процессора, что приводит к аварийному завершению программы.

3.5 Неправильный вывод подсказки ввода

Мы рассмотрели ещё одну проблему в работе отладчика[9].

ghci выводит специальную строку, когда ожидает ввода от пользователя. Изначально она состоит из списка загруженных модулей и символа больше. Пользователь может изменить содержание этой строки командой `:set prompt <новая подсказка>`.

Листинг 3.19 Подсказка ввода по умолчанию

```
Prelude>
```

Листинг 3.20 Команда :set prompt

```
Prelude> :set prompt INPUT#
INPUT#
```

После остановки на точке останова подсказка ввода меняется. Она дополняется названием файла и номером строки, где остановилось выполнение.

Листинг 3.21 Подсказка ввода во время остановки

```
Stopped in Main.list, main.hs:1:8-12
_result :: [Integer] = _
[main.hs:1:8-12] *Main>
```

При модификации исходного кода ghci была допущена ошибка, котора привела к тому, что место остановки печаталось дважды.

Рис. 2: Исправление

```
793 793 generatePromptFunctionFromString :: String -> PromptFunction
794     generatePromptFunctionFromString promptS = \_ _ -> do
795         (context, modules_names, line) <- getInfoForPrompt
796
797         let
    794 generatePromptFunctionFromString promptS modules_names line =
    795     processString promptS
    796   where
798 797     processString :: String -> GHCi SDoc
799 798     processString ('%':'s':xs) =
800 799         liftM2 (<>) (return modules_list) (processString xs)
801 800         where
802               modules_list = context <> modules_bit
803               modules_bit = hsep $ map text modules_names
    801       modules_list = hsep $ map text modules_names
```

Функция `generatePromptFunctionFromString` была написана неправильно. Как видно она включала контекст в список модулей, поэтому контекст печатался дважды. Мы исправили это и отправили исправления основным разработчикам, используя сервис Phabricator[10]. Их проверили визуально и помощью запуска тестов, после чего изменения были включены в git репозиторий ghc.

4 Заключение

Мы рассмотрели внутреннее устройство отладчика GHCi. Были продемонстрированы его возможности как отладчика функционального языка программирования. Были рассмотрены две ошибки, одна была исправлена. В ходе изучения ошибок в его работе были исследованы принципы взаимодействия отладчика и отлаживаемой программы. Был подробно изучен процесс компиляции и его особенности в контексте отладки. Также мы подробно описали реализацию механизма точек останова.

Список литературы

[1] Git репозиторий с исходным кодом проекта GHC, *git://git.haskell.org/ghc.git*

[2] Simon L Peyton Jones, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, 1992

[3] Антон Холомьёв, *Учебник по Haskell*, 2012

[4] GHC Trac issue #8316, GHCi debugger segfaults when trying force a certain variable *https://ghc.haskell.org/trac/ghc/ticket/8316*

[5] Руководство пользователя GHC, *https://downloads.haskell.org/~ghc/7.6.2/docs/html/users_guide*

[6] Simon Marlow and Simon Peyton-Jones, *The Architecture of Open Source Applications: The Glasgow Haskell Compiler*, 2012

[7] Richard Jones, *Tail recursion without space leaks*, 1991

[8] GHC Commentary: The Layout of Heap Objects, *https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects*

[9] GHC Trac issue #14973, Location in GHCi debugger prompt printed twice when default prompt is used, *https://ghc.haskell.org/trac/ghc/ticket/14973*

[10] Differential D4661, *https://phabricator.haskell.org/D4661*