

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

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

Направление подготовки
02.03.02 — Фундаментальная информатика
и информационные технологии

ЦВЕТОВАЯ ПОДДЕРЖКА ВЫВОДА КОНСОЛИ ИНТЕРПРЕТАТОРА
ЯЗЫКА HASKELL GHCi

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
Н. М. Нежевского

Научный руководитель:
ст. преп. Брагилевский В.Н.

Допущено к защите:

руководитель направления ФИИТ _____ В. С. Пилиди

Ростов-на-Дону
2016

Постановка задачи

1. Анализ возможных подходов к реализации подсветки вывода
2. Изучение программной структуры GHCi
3. Программная реализация цветовой поддержки вывода интерпретатора GHCi

Содержание

Постановка задачи	2
Введение	4
1. Решение на основе внешней обработки вывода	4
1.1. Алгоритм решения	4
1.2. Программная реализация	5
1.3. Сравнение реализаций	9
1.4. Анализ полученных результатов	10
2. Система печати в GHCi	10
3. Необходимые изменения в модуле Outputable	12
3.1. Изменения в модуле Outputable	12
3.2. Примеры с внедренными изменениями	14
Заключение	16
Список литературы	16

Введение

ГНС - один из наиболее развитых на сегодняшний день компиляторов функционального языка Haskell. В состав компилятора входит интерпретирующая интерактивная среда GHCi. GHCi используется для отображения результатов вызовов функций, отладочной информации, вывода типов и сортов функций.

Во многих интерактивных средах разработки поддерживается цветовое выделение ключевых слов, констант и выражений в программном выводе. Такая возможность широко используется для обеспечения наглядности и удобства чтения и анализа программного вывода.

В настоящее время в состав ГНС не входит встроенных средств для окраски выводимого текста. Существуют программы, позволяющие частично решить данную задачу. В данной работе рассматривается программная реализация, обеспечивающая цветовую поддержку консольного вывода в интерпретаторе, а также анализируется программная составляющая ГНС с точки зрения решения задачи цветовой поддержки вывода в консоль.

1. Решение на основе внешней обработки вывода

1.1. Алгоритм решения

Одним из возможных решений является внешняя обработка выводимого интерпретатором GHCi текстового результата. Этот вариант удобен тем, что для его реализации нет необходимости изменять исходный код интерпретатора.

Обработка вывода интерпретатора GHCi должна работать по данному алгоритму:

1. Внешняя программа-обработчик принимает пользовательский ввод;
2. Этот пользовательский ввод передаётся в интерпретатор GHCi;
3. Программа-обработчик перехватывает вывод результата GHCi;
4. Этот результат обрабатывается и возвращается пользователю.

Основной проблемой данного алгоритма является передача команд интерпретатору для обработки. GHC не предоставляет функционала для выполнения команд, подаваемых из внешней программы. Существует возможность строчного выполнения команд путём запуска компилятора GHC в интерактивном режиме со строкой-командой в параметре, однако в случае такого подхода теряется контекст выполнения, такой как значения вычисленных переменных.

1.2. Программная реализация

В рамках решения задачи цветовой поддержки вывода, была создана программная реализация, поддерживающая указанный алгоритм. Эта реализация является обёрткой над GHCi. Она предоставляет пользователю возможность консольного ввода команд, после чего эти команды передаются интерпретатору для вычисления. Результат вычислений обрабатывается во внешней программе, после чего выводится в консоль.

Проблема передачи выражений для вычисления компилятором GHC была решена использованием библиотеки `ghcid`. Данная библиотека позволяет запустить и использовать установленный в системе интерпретатор GHCi в фоновом режиме с полной поддержкой всего функционала. Для обращения к нему она предоставляет собственный API [1].

Листинг 1 — Пример выполнения входной строки с помощью ghcid

```
(ghci, _) <- startGhci "ghci" (Just ".") True  
let executeStatement = exec ghci  
input <- getLine  
result <- executeStatement input
```

В этом примере вводимая команда содержится в переменной `input`. Функция `executeStatement` передаёт введённую команду запущенному в фоне `ghci` и возвращает результат. Результат возвращается как набор строк в монаде `IO` и извлекается в переменную `result`.

После получения результатов из `GHCI`, набор строк передаётся парсеру. Парсер выделяет из строки элементы, которым необходимо добавить окраску. К данным элементам он добавляет информацию о типе подстроки. Информация о типе используется в дальнейшем для выбора параметров окраски.

```
data SDoc next =  
  SEmpty  
  | SText String (SDoc next)  
  | SToken Token (SDoc next)
```

`SToken` содержит некоторое выражение, которое необходимо окрасить и дополнительную информацию о типе выражения. Этот тип используется в дальнейшем для окраски выделенной подстроки. Если же подстрока не распознаётся, то она заносится в `SText`, и в дальнейшем никак дополнительно не обрабатывается.

```
data Token =  
  Arrow String  
  | Bracket String  
  | Digit String  
  | Typeclass String  
  | Type String  
  | CompileInfo String  
  | Keyword String
```

В листинге представлена функция-парсер, генерирующая из строки выражение типа `SDoc`.

Листинг 2 — Функция парсинга выражения

```
parseExpression :: String -> SDoc next
parseExpression "" = SEmpty
parseExpression str
  | isStartsWithArrow str
    = SToken (Arrow (extractArrow str)) (parseExpression .
      dropArrow $ str)
  | isStartsWithBracket str
    = SToken (Bracket (extractBracket str)) (parseExpression .
      dropBracket $ str)
  | isStartsWithTypeclass str
    = SToken (Typeclass (extractTypeclass str)) (parseExpression
      . dropTypeclass $ str)
  | isStartsWithType str
    = SToken (Type (extractType str)) (parseExpression . dropType
      $ str)
  | isStartsWithInfo str
    = SToken (CompileInfo (extractCInfo str)) (parseExpression .
      dropCInfo $ str)
  | isStartsWithKeyword str
    = SToken (Keyword (extractKeyword str)) (parseExpression .
      dropKeyword $ str)
  | otherwise
    = SText (extractWord str ) (parseExpression . dropWord $ str)
```

После построения выражения типа SDoc, функция-визитор `colorize` осуществляет проход по нему. Подвыражения типа SToken окрашиваются и преобразуются в тип SText с удалением информации о содержащемся типе. Окрашивание происходит добавлением ANSI СИМВОЛОВ.

Листинг 3 — Окрашивание выражения

```
colorize :: SDoc next -> SDoc next
colorize SEmpty          = SEmpty
colorize (SToken token n) = SText (colorizeToken token) (colorize
  n)
colorize (SText str n)    = SText str (colorize n)
```

Функция `colorizeToken` используется для окраски выражения, содержащегося в подвыражении. Он использует информацию, представленную типом `Token` для выбора цвета окраски данной строки.

```
colorizeToken :: Token -> String
colorizeToken (Arrow arrow) = clr SRed arrow
colorizeToken (Bracket bracket) = clr SBlue bracket
...
```

Список доступных цветов описан в типе-перечислении `SColor`.

```
data SColor =
    SRed
  | SBlue
  | SGreen
  | ...
```

Тип-перечисление `SColor` имеет реализацию класса типов `Show`. С его помощью можно получить строковое представление указанного цвета.

```
instance Show SColor where
    show SRed = "\x1b[31m"
    show SGreen = "\x1b[32b"
    ...
```

Функция добавления ANSI символов `clr` получает строковое представление передаваемого цвета и имеет следующий вид.

```
clr :: SColor -> String -> String
clr col str = (show col) ++ str ++ (show getDefaultColor)
```

После данных преобразований, исходное выражение типа `SDoc` содержит последовательность строк с имеющимся цветовым выделением. Функция `concat'` проходит по полученному выражения, преобразуя его в строку.

```
concat' :: SDoc next -> String
concat' SEmpty = ""
concat' (SText subStr subSDoc) = subStr ++ (concat' subSDoc)
```

На скриншоте ниже представлен скриншот с примером работы полученной реализации. Запуск проводился на GHC версии 7.8.4. Бы-

ло продемонстрировано цветное выделение текстовых и численных констант, а также встроенных в GHC типов и классов типов.

```
GHCi, version 7.8.4: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
ghci>> [1..10]
[1,2,3,4,5,6,7,8,9,10]
ghci>> let fun = \x -> if x > 0 then "pos" else (show x)
ghci>> :t fun
fun :: (Show a, Ord a, Num a) => a -> [Char]
ghci>> map fun [-3..2]
["-3", "-2", "-1", "0", "pos", "pos"]
ghci>> :load some_module
[1 of 1] Compiling Main                  ( some_module.hs, interpreted )
Ok, modules loaded: Main.
ghci>> :t some_fun
some_fun :: Int -> Int
ghci>> some_fun

<interactive>:25:1:
  No instance for (Show (Int -> Int)) arising from a use of 'print'
  In a stmt of an interactive GHCi command: print it
ghci>> some_fun 3
4
ghci>> █
```

1.3. Сравнение реализаций

В рамках сравнения реализаций, рассмотрим скрипт `ghci-color`, используемый для задачи окрашивания вывода [2].

Скрипт `ghci-color` также использует внешнюю обработку текстового вывода. Это достигается использованием потокового текстового редактора `sed`. `Sed` получает входной поток, преобразует каждую строку строку, согласно записанным в скрипт правилам, а затем выводит результат в выходной поток. Правила в скрипте основываются на регулярных выражениях. К подстроке, распознающейся регулярным выражением, добавляются ANSI символы.

Несмотря на схожий принцип действия, данные реализации имеют различия. Для использования `ghci-color` необходимо установ-

ливать текстовый редактор `sed`. Кроме того, в текущей реализации `sed` отсутствует подсветка типов и классов типов. Масштабирование скрипта с правилами затруднено тем, что для каждого нового элемента необходимо добавлять новое правило на основе регулярных выражений.

1.4. Анализ полученных результатов

Полученная реализация решает поставленную задачу по окраске вывода интерпретатора `GHCI`, однако обладает рядом недостатков:

1. Программа может обеспечить корректную окраску только для стандартных имён, так как не обладает информацией об определенных пользователем типах, классах типов и переменных
2. В некоторых ОС необходимы дополнительные действия по обеспечению корректной работы подсветки на основе ANSI символов
3. Со стороны пользователя требуются дополнительные действия для запуска данной программы, и её дальнейшего использования

Таким образом, полученная программа решает поставленную задачу по окраске вывода `GHCI`. Однако, с учётом указанных выше недостатков, решение можно считать частичным.

2. Система печати в `GHCI`

Система вывода компилятора `GHSC` основана на принципе структурной печати. Структурная печать используется для программного комбинирования, форматирования и вывода текста [3].

Основном примитивом в модели структурной печати является тип `Doc`, находящийся в модуле `Pretty`.

```
data Doc
= Empty
| NilAbove Doc
| Nest FastInt Doc
| Union Doc Doc
| ...
```

Тип `Doc` представляет набор текстовых элементов. Для данных текстовых элементов определены различные операции конкатенации, форматирования и выравнивания.

Тип `SDoc` является для типа `Doc`, он находится в модуле `Outputable`. Тип `SDoc` содержит дополнительную информацию для обработки текстового элемента, содержащегося в `Doc`.

```
type SDoc = PprStyle -> Doc
```

```
data PprStyle
= PprUser PrintUnqualified Depth
| PprCode CodeStyle
| PprDump
| PprDebug
```

`PprStyle` содержит перечисление вариантов вывода текста. Он используется, чтобы указать цель вывода текста: пользовательский вывод, отладка или печать самой программы.

В языке Haskell класс типов содержит сигнатуры функций, применяемых к значениям некого типа. [4] Класс типов `Outputable` предоставляет интерфейс для типов, поддерживающих функцию структурной печати. Для большинства встроенных типов в Haskell имеется поддержка `Outputable`.

```
class Outputable a where
    ppr :: a -> SDoc
```

Класс типов `Outputable` имеет одну функцию `ppr`, которая переводит выражение исходного типа в соответствующее ему выражение типа `SDoc`. Например, функция `ppr` для типа `Int` имеет следующий вид:

```
instance Outputable Int where  
  ppr n = int n
```

```
int n = \ _ -> (text $ show n)
```

Для конечного преобразования элементов типа Dос в строки и их вывода, используется функция рендеринга текста, входящая в состав модуль Pretty.

Листинг 4 — Типовая аннотация к функции рендеринга текста

```
render :: Handle -> SDoc -> IO ()
```

Текущая реализация класса типов Outputable и типа SDoc не поддерживает указание цвета данного элемента, поэтому для решения задачи по добавлению окраски вывода необходимо изменить их реализации.

3. Необходимые изменения в модуле Outputable

Для получения списка необходимых изменений в системе вывода GHC, была разработана модель вывода, действующая аналогично системе вывода в GHC. Эта модель поддерживала ограниченное количество типов и операций с элементами. На основе этой модели были проанализированы и получены изменения, которые необходимы в GHC для обеспечения подсветки вывода.

Для внедрения данных изменений в компилятор GHC, необходимо изменение большого количество стороненно кода, поэтому предлагаемые изменения рассматриваются в рамках созданной модели.

3.1. Изменения в модуле `Outputable`

В текущей реализации тип `SDoc` содержит информацию только о формате вывода. Тип-перечисление `PprStyle` в `SDoc` необходимо заменить на тип `SDocStyle`, который будет содержать информацию о формате вывода и о цвете данного текста. Это добавление указывать текст данного элемента в момент его создания.

Листинг 5 — Дополненная реализация типа `SDoc`

```
type SDoc = SDocStyle -> Doc
data SDocStyle = {
    pprStyle :: PprStyle,
    pprColor :: PprColor
}
type PprColor =
    CDefault
  | CRed
  | CBlue
  | ...
```

Примером работы с обёрткой при изменении цвета, является функция `colorize`. Ввиду неизменяемости данных в языке Haskell, функция `colorize` изменяет информацию о цвете путем создания нового элемента типа `SDoc`, содержащего новый цвет.

Листинг 6 — Пример функции окраски

```
colorize :: PprColor -> SDoc -> SDoc
colorize newPprColor sDoc = (newSDocStyle -> docContent)
  where
    docContent = sDoc createEmptyStyle
    sDocStyle = pprStyle sDoc
    newSDocStyle = SDocStyle {
        pprStyle = pprStyle sDocStyle,
        pprColor = newPprColor
    }
```

Для поддержки цветового выделения всеми типами, необходим новый класс типов `ColorOutputable`. Чтобы поддерживать

ColorOutputable, тип должен иметь поддержку Outputable. Для полной поддержки введённых изменений, необходимо полностью заменить Outputable на ColorOutputable во всех участках кода, где создаются элементы типа SDoc. Также ColorOutputable должен поддерживать все операции комбинирования, используемые в структурной печати.

Листинг 7 — Класс типов ColorOutputable

```
class Outputable a => ColorOutputable a where  
  cprpr :: a -> SDoc
```

Для получения соответствующего элемента типа SDoc на основании выражения исходного типа, функция cprpr использует функцию prpr. После чего к выражению добавляется информация о цвете. Если данный элемент не требует окрашивания, то его значением цвета указывается CDefault. При дальнейшей генерации выводимого текста, этот элемент дополнительно не обрабатывается и выводится в стандартном цвете.

Листинг 8 — Пример реализации ColorOutputable для типа Int

```
instance ColorOutputable Int where  
  cprpr n = (colorize CBlue) . prpr $ n
```

Полученная информация о цвете используется при получении строкового представления из элементов типа SDoc. Функция colorizeDoc получает окрашенный вариант элемента типа Doc.

```
colorizeDoc :: String -> String -> Doc -> Doc  
colorizeDoc newColorStr defColorStr doc = (ptext newColorStr) <+>  
  doc <+> (ptext defColorStr)
```

3.2. Примеры с внедренными изменениями

В качестве примеров рассмотрим несколько функций, аналоги которых были реализованы в модели.

Функция `mkExpectedActualMsg`, входящая в модуль `TcErrors` генерирует сообщение о несоответствии типов. Этот поддокумент состоит из ожидаемого типа и полученного типа.

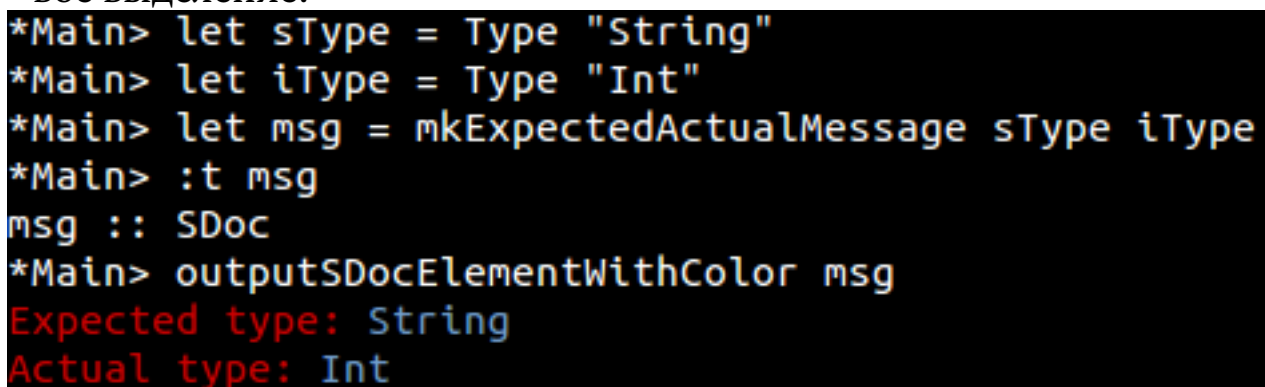
```
mkExpectedActualMsg :: Type -> Type -> SDoc
mkExpectedActualMsg exp_ty act_ty
  = vcat [ ptext (text "Expected type:") <+> ppr exp_ty
          , ptext (text "  Actual type:") <+> ppr act_ty ]
```

В данной функции возможно добавить цветное выделение текстовых констант и типов. Для окрашивания типов используется функция `srpr`, входящая в класс типов `ColorOutputable`. Тип `String` не имеет реализации для `Outputable`, поэтому для него нельзя реализовать `ColorOutputable`. В данном примере окраска строковых констант выполняется вручную.

Листинг 9 — Реализация функции с изменениями

```
mkExpectedActualMsg :: Type -> Type -> SDoc
mkExpectedActualMsg exp_ty act_ty
  = vcat [ colorize cRed $ ptext (text "Expected type:") <+> cprpr
          exp_ty
          , colorize cRed $ ptext (text "  Actual type:") <+> cprpr
          act_ty ]
```

На приведённом скриншоте иллюстрируется полученное цветное выделение.



```
*Main> let sType = Type "String"
*Main> let iType = Type "Int"
*Main> let msg = mkExpectedActualMessage sType iType
*Main> :t msg
msg :: SDoc
*Main> outputsDocElementWithColor msg
Expected type: String
Actual type: Int
```

Функция `srcParseErr` используется в парсере компилятора GHC. Она генерирует сообщение об ошибке в пользовательском вводе.

```
srcParseErr options buf len = if null token
```

```

    then ptext (text "parse error (possibly incorrect
        indentation or mismatched brackets)")
    else ptext (text "parse error on input" <+> quotes (text
        token))
where token = ...

```

В данном примере добавляется цветное выделение к сообщению об ошибке и неправильно введённому значению.

```

srcParseErr options buf len = if null token
    then parseErrorMsg <+> ptext (text "(possibly incorrect
        indentation or mismatched brackets)")
    else parseErrorMsg <+> ptext (text " on input" <+>
        quotes (text token))
where
    parseErrorMsg = colorize cRed $ ptext (text "parse error")
    token = ...

```

Данные примеры иллюстрируют, что представленных выше изменений достаточно для цветовой поддержки вывода консоли интерпретатора GHCi.

Заключение

В результате работы была создана программная реализация, решающая поставленную задачу. Эта реализация имеет ряд указанных недостатков.

Была проанализирована реализация вывода, используемая в GHC. На основе этой реализации был создан модуль, действующий аналогично системе вывода GHC. По данной модели были получены изменения, которые необходимо внести в систему печати GHC для того, чтоб реализовать цветовую поддержку вывода.

Список литературы

1. Библиотека ghcid. – URL: <https://github.com/ndmitchell/ghcid> (дата обр. 05.05.2016)
2. Скрипт ghci-color. – URL: <https://github.com/rhysd/ghci-color> (дата обр. 06.05.2016)
3. Документация по компилятору GHC – URL: <https://downloads.haskell.org/ghc/7.10.2/docs/html/> (дата обр. 30.04.2016)
4. Bryan O’Sullivan – Real World Haskell. – URL: <http://book.realworldhaskell.org/> (дата обр 25.04.2016)