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
Фундаментальная информатика и информационные технологии

МОДИФИКАЦИЯ МЕХАНИЗМА ОБРАБОТКИ ОТРИЦАТЕЛЬНЫХ ЛИТЕРАЛОВ В КОМПИЛЯТОРЕ GHC

Курсовая работа

Студента 3 курса
Д. Ш. Мирзоева

Научный руководитель:
Старший преподаватель кафедры информатики и вычислительного
эксперимента Мехмата ЮФУ В. Н. Брагилевский

_____ _____

оценка(рейтинг) подпись руководителя

Ростов-на-Дону
2017

Содержание

Введение

Во многих языках программирования присутствует понятие литерала. Литерал — это предопределённая компилятором константа.

В данной курсовой работе нами рассмотрен язык Хаскель. В его стандарте [1] отсутствуют отрицательные литералы в качестве отдельной лексемы. Для записи отрицательных чисел используется унарная операция минус. Такой подход порождает некоторые особенности, что побудило сообщество разработчиков создать расширение для языка, добавляющее настоящие отрицательные литералы.

Как выяснилось, способ, которым было реализовано данное расширение, не позволял корректно обрабатывать значение “отрицательный ноль”. `-0` компилятор неверно распознавал как обыкновенный, положительный ноль. Такое поведение было воспринято как ошибка. Нами были рассмотрены несколько путей решения данной проблемы. Один из этих способов оказался приемлем для разботчиков компилятора. Изменения подготовленные нами были приняты в основной репозиторий.

1 Литералы в Haskell

1.1 Классификация

В Хаскель можно выделить четыре вида литералов: целочисленные, вещественные, символьные и строковые.

В отличие от многих других языков, логические значения `True` и `False` являются идентификаторами. Они входят в состав стандартного модуля `Prelude` и определены как показано в листинге 1.1.

Листинг 1.1 Определение значений True и False

```
data Bool = True | False
```

Целочисленные литералы представляют значения натуральных чисел, включая ноль. Существует синтаксис для записи в различных системах счисления: десятичной, восьмеричной, шестнадцатеричной. Примеры: `15`, `0o17`, `0O17`, `0xF`, `0XF`.

$$
\begin{array}{lcl}
\mathit{decimal} & \rightarrow & \mathit{digit}\{\mathit{digit}\} \\
\mathit{octal} & \rightarrow & \mathit{octit}\{\mathit{octit}\} \\
\mathit{hexadecimal} & \rightarrow & \mathit{hexit}\{\mathit{hexit}\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{integer} & \rightarrow & \mathit{decimal} \\
 & | & \texttt{0o}\ \mathit{octal} \mid \texttt{0O}\ \mathit{octal} \\
 & | & \texttt{0x}\ \mathit{hexadecimal} \mid \texttt{0X}\ \mathit{hexadecimal}
\end{array}
$$

Рис. 1: Лексическая структура целочисленных литералов[1]

Вещественные литералы представляют рациональные числа. Можно использовать как обычные десятичные дроби, так и научную форму записи с указанием мантиссы и порядка. Примеры: `123.456`, `0.123456e3`.

$$
\begin{array}{lcl}
\mathit{float} & \rightarrow & \mathit{decimal}\ \texttt{.}\ \mathit{decimal}\ [\mathit{exponent}] \\
 & | & \mathit{decimal}\ \mathit{exponent} \\
 & & \\
\mathit{exponent} & \rightarrow & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \mathit{decimal}
\end{array}
$$

Рис. 2: Лексическая структура вещественных литералов[1]

Символьные литералы записываются в одинарных кавычках, а строковые в двойных. Примеры: `'a'`, `"Hello, World!"`.

$$
\begin{array}{lcl}
\mathit{char} & \rightarrow & \texttt{'}\ (\mathit{graphic}_{\langle \texttt{'} \mid \backslash \rangle} \mid \mathit{space} \mid \mathit{escape}_{\langle \backslash\, \& \rangle})\ \texttt{'} \\
\mathit{string} & \rightarrow & \texttt{"}\ \{\mathit{graphic}_{\langle \texttt{"} \mid \backslash \rangle} \mid \mathit{space} \mid \mathit{escape} \mid \mathit{gap}\}\ \texttt{"} \\
\mathit{escape} & \rightarrow & \backslash\ (\ \mathit{charesc} \mid \mathit{ascii} \mid \mathit{decimal} \mid \texttt{o}\ \mathit{octal} \mid \texttt{x}\ \mathit{hexadecimal}\ ) \\
\mathit{charesc} & \rightarrow & \texttt{a} \mid \texttt{b} \mid \texttt{f} \mid \texttt{n} \mid \texttt{r} \mid \texttt{t} \mid \texttt{v} \mid \backslash \mid \texttt{"} \mid \texttt{'} \mid \texttt{\&} \\
\mathit{ascii} & \rightarrow & \texttt{\^{}}\ \mathit{cntrl} \mid \texttt{NUL} \mid \texttt{SOH} \mid \texttt{STX} \mid \texttt{ETX} \mid \texttt{EOT} \mid \texttt{ENQ} \mid \texttt{ACK} \\
 & | & \texttt{BEL} \mid \texttt{BS} \mid \texttt{HT} \mid \texttt{LF} \mid \texttt{VT} \mid \texttt{FF} \mid \texttt{CR} \mid \texttt{SO} \mid \texttt{SI} \mid \texttt{DLE} \\
 & | & \texttt{DC1} \mid \texttt{DC2} \mid \texttt{DC3} \mid \texttt{DC4} \mid \texttt{NAK} \mid \texttt{SYN} \mid \texttt{ETB} \mid \texttt{CAN} \\
 & | & \texttt{EM} \mid \texttt{SUB} \mid \texttt{ESC} \mid \texttt{FS} \mid \texttt{GS} \mid \texttt{RS} \mid \texttt{US} \mid \texttt{SP} \mid \texttt{DEL} \\
\mathit{cntrl} & \rightarrow & \mathit{ascLarge} \mid \texttt{@} \mid \texttt{[} \mid \backslash \mid \texttt{]} \mid \texttt{\^{}} \mid \texttt{\_} \\
\mathit{gap} & \rightarrow & \backslash\ \mathit{whitechar}\ \{\mathit{whitechar}\}\ \backslash
\end{array}
$$

Рис. 3: Лексическая структура символьных и строковых литералов[1]

1.2 Отрицательные числа

Как можно заметить из приведённых фрагментов стандарта, знак не является частью лексемы обозначающей число. Тем не менее запись `-10` является синтаксически правильной. Минус в данном случае является отдельной лексемой, обозначающей операцию унарный минус. Во время компиляции он заменяестя вызовом функции `negate`, определённой в классе `Num` стандартного модуля `Prelude`.

Вместе с другими особенностями синтаксиса и системы типов это рождает некоторые трудности.

1.3 Особенность 1: вызов функции записывается без скобок

Функции играют в языке Хаскель важную роль. Для такого частого действия, как вызов функции, разработчики решили сделать максимально простой синтаксис. Чтобы вызвать фукнцию `f` с аргументом `x`, достаточно просто написть `f x`.

Листинг 1.2 демонстрирует пробему, связанную с отсутствием необходимости писать скобки вокруг аргументов. Выражение `print -1` воспринимается компилятором, как вызов функции `print` с двумя аргументами: `-` и `1`. Такое действие невозможно, о чём нам любезно сообщает компилятор. Вероятно, программист хотел напечатать число минус один. Для того чтобы сделать это, ему необходимо обернуть `-1` в скобки, то есть написать `print (-1)`.

Листинг 1.2 Ошибка при отсутствии скобок вокруг унарного минуса

```
Prelude> print (-1) -- OK
Prelude> print -1   -- ERROR

<interactive>:2:1:
    Non type-variable argument in the constraint:
                                  Num (a -> IO ())
    (Use FlexibleContexts to permit this)
    When checking that ‘it’ has the inferred type
      it :: forall a. (Num (a -> IO ()), Show a)
            => a -> IO ()
```

Стоит сказать, что многие программисты на Хаскель стараются не писать лишних скобок. Для них необходимость оборачивать отрицательные числа в скобки может может быть раздражающей и даже мешающей, ухудшающей читаемость кода.

1.4 Особенность 2: тип значения представляемого литералом определяется контекстом

Система типов языка Хаскель позволяет литералам иметь разный тип в разных контекстах. Сами по себе целочисленные и вещественные ли-

тералы не имеют конкретного предопределённого типа. Он определяется во время компиляции в каждом месте использования таким образом, чтобы удовлетворять наложенным ограничениям. К примеру, если функция имеет параметр типа `Int8`, и мы передадим ей в качестве аргумента число `42`, то компилятор определит его тип как `Int8`.

Листинг 1.3 Полиморфные литералы

```
Prelude> :t 42
42 :: Num a => a
Prelude> :t 3.14
3.14 :: Fractional a => a
```

Вместе с отсутствием отрицательных литералов такая особенность может порождать ошибки или как минимум замедлять работу кода. Такую ситуацию демонстрирует листинг 1.4. Рассмотрим более подробно, что происходит в процессе компиляции этого фрагмента. Для выражения `-128` указан тип `Int8`. Таким должен быть результат функции `negate`, применённой к литералу `128`. Её тип — `Num a => a -> a`, то есть её параметр и результат должны быть одного типа, поэтому `128` тоже будет иметь тип `Int8`. Теперь обратим внимание на значения, представимые в этом типе: `-128..127`. Число `128` не входит в это множество, поэтому произойдёт переполнение в значение `-128`. Во время выполнения к нему будет применена функция `negate`, результатом которой было бы число `128`, если бы оно было представимо в этом типе. На самом деле произойдёт ещё одно переполнение в значение `-128`.

Листинг 1.4 Переполнения на граничных значениях

```
Prelude Data.Int> print (-128::Int8)
<interactive>:3:9: Warning:
    Literal 128 is out of the Int8 range -128..127
    If you are trying to write a large negative
    literal, use NegativeLiterals
-128
```

Такое количество переполнений безусловно нежелательно и может привести к потере в производительности. Конечно, в данном случае мы

получили желаемый результат: напечатано число -128, но в случае с другими типами, для которых реализован класс Num, результат может быть совершенно неожиданным.

2 Расширение NegativeLiterals

2.1 Описание

Для решения проблем связанных с обозначенными особенностями было создано расширение NegativeLiterals. Оно добавляет в язык отрицательные литералы. Чтобы использовать это расширение, необходимо либо добавить флаг компиляции -XNegativeLiterals, либо добавить в начало файла с кодом программы строку
{-# LANGUAGE NegativeLiterals #-}

Листинг 2.1 Демонстрация NegativeLiterals

```
$ ghci -XNegativeLiterals
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> print -1
-1
Prelude> import Data.Int
Prelude Data.Int> print (-128::Int8)
-128
```

2.2 Реализация

Расширение модифицирует работу лексического анализатора. В его описание добавляется несколько новых правил.

Листинг 2.2 Правила работы лексического анализатора, связанные с расширением NegativeLiterals

```
@negative @decimal -- Регулярное выражение
    / { ifExtension negativeLiteralsEnabled } -- Условие
    { tok_num negative 1 1 decimal } -- Генератор лексемы
@negative 0[bB] @binary
    / { ifExtension negativeLiteralsEnabled `alexAndPred`
        ifExtension binaryLiteralsEnabled }
    { tok_num negative 3 3 binary }
@negative 0[oO] @octal
    / { ifExtension negativeLiteralsEnabled }
    { tok_num negative 3 3 octal }
@negative 0[xX] @hexadecimal
    / { ifExtension negativeLiteralsEnabled }
    { tok_num negative 3 3 hexadecimal }
...
@negative @floating_point
    / { ifExtension negativeLiteralsEnabled }
    { strtoken tok_float }
```

Лексический анализатор компилятора GHC генерируется с помощью инструмента Alex. В листинге 2.2 приведены несполько правил, использующих его синтаксис.

Каждое правило состоит из нескольких частей. Первая часть — это регулярное выражение, описывающее лексему. Здесь могут быть использованы макро определения, то есть предварительно определённые регулярные выражения. В листинге 2.2 такими макро определениями являются `@negative`, `@decimal`, `@binary` и другие. Вторая часть — это контекст, в котором это правило может применяться. В приведённом фрагменте проверяется включено ли расширение NegativeLiterals. Третья часть — это код на языке Хаскель, который создаёт объект лексемы на основе последовательности символов, ей соответствующей.

3 Проблема обработки отрицательного нуля в расширении NegativeLiterals

3.1 Описание ошибки

NegativeLiterals решает озвученные проблемы, но, как выяснилось, приносит новые. Недавно была обнаружена ошибка, связанная с данным расширением. [2]

Ошибка заключалась в нераспознавании отрицательного нуля. Без включённого расширения код `print (-0::Double)` верно печатал `-0.0`, но после включения на экран выводился положительный ноль. Листинг 3.1 это демонстрирует.

Листинг 3.1 Пример ошибки

```
$ ghc-8.0.1 -e '-0.0'
-0.0
$ ghc-8.0.1 -XNegativeLiterals -e '-0.0'
0.0
```

rwbarton, обнаруживший данную ошибку, пишет:

> Это логично, но неожиданно. Логика такова, что с NegativeLiterals литерал `-0.0` обозначает `fromRational ((-0)%0)`, что равняется `fromRational 0`, что для `Double` равно положительному нулю, а не отрицательному.
>
> Но это выглядит неудачно. Возможно, было бы лучше, если `0.0` (или `-` применённый к любому другому литералу математически равному нулю) был бы рассахарен в `negate 0.0` даже с NegativeLiterals, так как, похоже, нет никакой другой причины писать `-0.0`.

3.2 Первая идея

Учитывая, что расширение затрагивало только лексический анализатор, мы решили начать с него. Поведение компилятора с выключенным

расширением было правильным, поэтому мы попробовали модифицировать его правила таким образом, чтобы NegativeLiterals не влияло на обработку отрицательного нуля.

Листинг 3.2 Новые макро определения

```
$ascnzdigit = 1-9
@nz_decimal = $decdigit* $ascnzdigit $decdigit*
@nz_floating_point = @nz_decimal \. @decimal @exponent?
                   | @decimal \. @nz_decimal @exponent?
                   | @nz_decimal @exponent
```

Мы добавили два новых макро определения: для ненулевых целых (@nz_decimal) и для ненулвых вещественных чисел(@nz_floating_point).

Листинг 3.3 Обновлённые правила

```
-  @negative @decimal
    / { ifExtension negativeLiteralsEnabled }
    { tok_num negative 1 1 decimal }
+  @negative @nz_decimal
    / { ifExtension negativeLiteralsEnabled }
    { tok_num negative 1 1 decimal }
...
-  @negative @floating_point
    / { ifExtension negativeLiteralsEnabled }
    { strtoken tok_float }
+  @negative @nz_floating_point
    / { ifExtension negativeLiteralsEnabled }
    { strtoken tok_float }
```

С их помощью мы модифицировали правила расширения NegativeLiterals как показано в листинге 3.3.

На первый взгляд такое решение работало, что подтверждали простейшие проверки вроде той, что указана в описании ошибки. К сожалению, запуск тестов выявил недостатки данного подхода. Листинг 3.4 демонстрирует, обнаруженную ошибку.

Листинг 3.4 Ошибка

```
Prelude> print -1
-1
Prelude> print -0

<interactive>:2:1:
    Non type-variable argument in the constraint:
                                  Num (a -> IO ())
    (Use FlexibleContexts to permit this)
    When checking that ‘it’ has the inferred type
      it :: forall a. (Num (a -> IO ()), Show a)
            => a -> IO ()
```

Мы вернули обычное поведение, но забыли, что оно нас не устраивало. В первом случае минус является частью литерала, а во втором — отдельной лексемой. Несмотря на свою простоту такой подход не позволил нам решить проблему, поэтому необходимо было найти другой способ.

3.3 Вторая идея

Решено было отложить обработку отрицательного нуля до более позднего этапа компиляции. Идея заключалась в том, чтобы для отрицательного нуля генерировать вызов `negate`. Для этого понадобилось отдельно от значения литерала сохранять его знак. Это связано с тем, что значения хранятся в типах `Integer` и `Rational`, в которых нет отдельного значения для отрицательного нуля.

Листинг 3.5 Определения лексем

```
data Token =
...
-   | ITinteger  SourceText Integer
+   | ITinteger  IntegralLit
    | ITrational FractionalLit
...
```

Листинг 3.6 Хранимая в литерале информация

```
+data IntegralLit
+  = IL { il_text :: SourceText
+       , il_neg :: Bool -- See Note [Negative zero]
+       , il_value :: Integer
+       }
+  deriving (Data, Show)
data FractionalLit
-  = FL { fl_text :: String     -- How the value was written
-                               -- in the source
+  = FL { fl_text :: SourceText -- How the value was written
+                               -- in the source
+       , fl_neg :: Bool        -- See Note [Negative zero]
       , fl_value :: Rational  -- Numeric value of the literal
       }
   deriving (Data, Show)
```

Мы добавили поле `neg` типа `Bool` в структуры, хранящие информацию о литерале. Для отрицательных литералов его значение `True`, а для положительных `False`.

На этапе разрешения имён эта информация может быть использована для того, чтобы обнаружить отрицательный ноль. Идеальным местом для этого оказалась функция `rnOverLit`, которая производит разрешение имён перегруженных литералов, к которым как раз относятся целые и вещественные литералы. В листинге 3.7 показаны осуществлённые изменения. Теперь она помимо литерала возвращает также функцию `negate`, если она применена к отрицательному нулю. Это позволяет в любом месте вызова этой функции легко вставить вызов функции `negate`. Может показаться более выгодным изменить сигнатуру функции `rnOverLit` так, чтобы она возвращала не обязательно литерал, а, к примеру, выражение — уже применённую функцию `negate`. К сожалению, существующая кодовая база не позволяла легко произвести подобное изменение.

Листинг 3.7 rnOverLit

```
+isNegativeZeroOverLit :: HsOverLit t -> Bool
+isNegativeZeroOverLit lit
+ = case ol_val lit of
+        HsIntegral i   -> 0 == il_value i && il_neg i
+        HsFractional f -> 0 == fl_value f && fl_neg f
+        _              -> False
+
-rnOverLit :: HsOverLit t -> RnM (HsOverLit Name, FreeVars)
+rnOverLit :: HsOverLit t ->
+             RnM ((HsOverLit Name, Maybe (HsExpr Name)), FreeVars)
 rnOverLit origLit
   = do  { opt_NumDecimals <- xoptM LangExt.NumDecimals
         ; let { lit@(OverLit {ol_val=val})
             | opt_NumDecimals = origLit {ol_val =
                   generalizeOverLitVal (ol_val origLit)}
             | otherwise       = origLit
           }
         ; let std_name = hsOverLitName val
-        ; (SyntaxExpr { syn_expr = from_thing_name }, fvs)
+        ; (SyntaxExpr { syn_expr = from_thing_name }, fvs1)
             <- lookupSyntaxName std_name
         ; let rebindable = case from_thing_name of
                                HsVar (L _ v) -> v /= std_name
                                _             -> panic "rnOverLit"
-        ; return (lit { ol_witness = from_thing_name
-                      , ol_rebindable = rebindable
-                      , ol_type = placeHolderType }, fvs) }
+        ; let lit' = lit { ol_witness = from_thing_name
+                         , ol_rebindable = rebindable
+                         , ol_type = placeHolderType }
+        ; if isNegativeZeroOverLit lit'
+          then do { (SyntaxExpr { syn_expr = negate_name }, fvs2)
+                      <- lookupSyntaxName negateName
+                  ; return ((lit' { ol_val = negateOverLitVal val }
+                            , Just negate_name)
+                           , fvs1 `plusFV` fvs2) }
+          else return ((lit', Nothing), fvs1) }
```

Основным местом использования `rnOverLit` служит функция `rnExpr`. В листинге 3.8 показано, как мы генерируем применение функции `negate` для отрицательного нуля.

Листинг 3.8 rnExpr

```
rnExpr (HsOverLit lit)
-  = do { (lit', fvs) <- rnOverLit lit
-       ; return (HsOverLit lit', fvs) }
+  = do { ((lit', mb_neg), fvs) <- rnOverLit lit
+                                  -- See note: [Negative zero]
+       ; case mb_neg of
+           Nothing -> return (HsOverLit lit', fvs)
+           Just neg -> return ( HsApp (noLoc neg)
+                                      (noLoc (HsOverLit lit'))
+                              , fvs ) }
```

Сложные изменения в коде требуют развернутого комментария. Для того чтобы другой программист, работающий с этим кодом, понял почему он написан именно так, мы написали такой комментарий и добавили ссылки на него в самых важных местах:

```
{-
Note [Negative zero]
~~~~~~~~~~~~~~~~~~~~
There were problems with negative zero in conjunction
with Negative Literals extension. Numeric literal
value is contained in Integer and Rational types
inside IntegralLit and FractionalLit. These types
cannot represent negative zero value. So we had to
add explicit field 'neg' which would hold information
about literal sign. Here in rnOverLit we use it to
detect negative zeroes and in this case return not
only literal itself but also negateName so that users
can apply it explicitly. In this case it stays
negative zero.  Trac \#13211
-}
```

Каждая ошибка, обнаруженная в компиляторе, нуждается в тесте. Мы написали тест, который способен обнаружить решаемую нами проблему. Он приведён в листинге 3.9.

Листинг 3.9 Тест обработки отрицательного нуля

```
-- | Test for @NegativeLiterals@ extension (see GHC #13211)

{-# LANGUAGE NegativeLiterals #-}

floatZero0 = 0 :: Float
floatZero1 = 0.0 :: Float

floatNegZero0 = -0 :: Float
floatNegZero1 = -0.0 :: Float

doubleZero0 = 0 :: Double
doubleZero1 = 0.0 :: Double

doubleNegZero0 = -0 :: Double
doubleNegZero1 = -0.0 :: Double

main = do
    print (isNegativeZero floatZero0)
    print (isNegativeZero floatZero1)
    print (isNegativeZero floatNegZero0)
    print (isNegativeZero floatNegZero1)
    print (isNegativeZero doubleZero0)
    print (isNegativeZero doubleZero1)
    print (isNegativeZero doubleNegZero0)
    print (isNegativeZero doubleNegZero1)
```

После исправления мелких недочётов и форматирования отладочного вывода все тесты были пройдены.

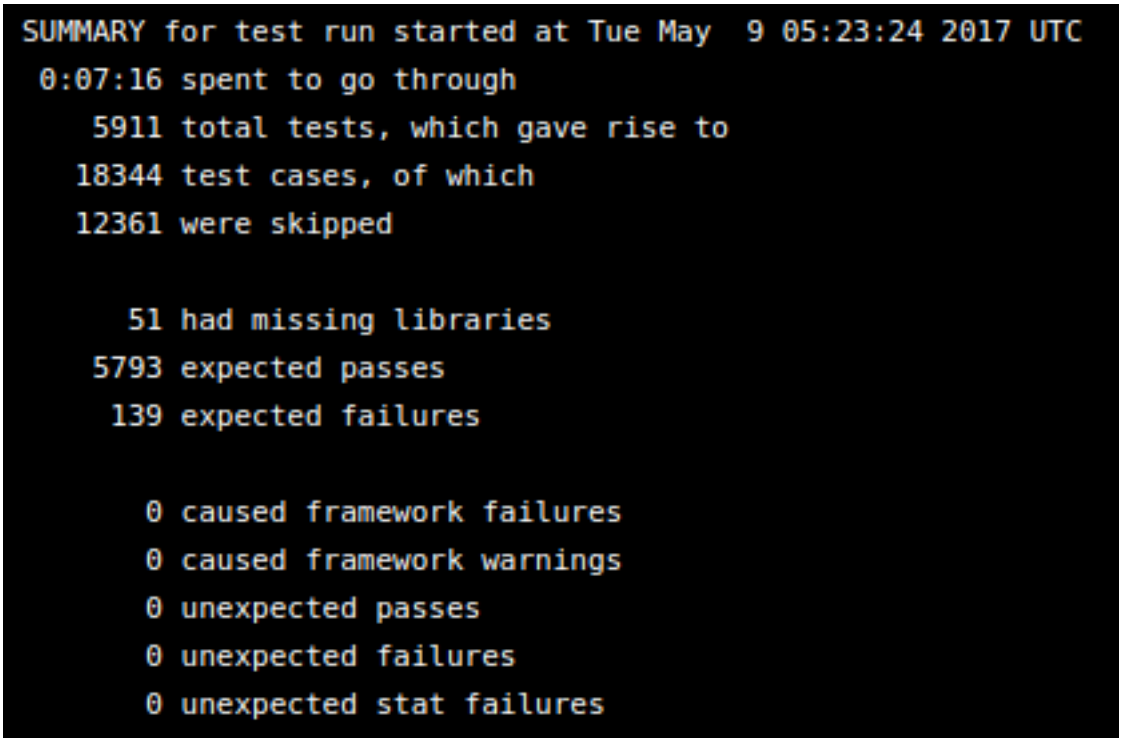

Рис. 4: Результаты тестов

Для упрощения анализа изменений, вносимых в компилятор, и быстрого прогона тестов разработчики GHC используют систему Phabricator. Мы опубликовали изменения с помощью этой системы. [3] Бен Гамари, один из рецензентов, проанализировал код и согласился принять данные изменения в основной репозиторий. Это произошло 9 мая [4], что поставило точку в решении проблемы обработки отрицательного нуля.

Список литературы

[1] Simon Marlow(editor), *Haskell 2010 Language Report*

[2] NegativeLiterals -0.0 :: Double, *https://ghc.haskell.org/trac/ghc/ticket/13211*

[3] Haskell Phabricator D3543, Make XNegativeLiterals treat -0.0 as negative 0, *https://phabricator.haskell.org/D3543*

[4] Commit to GHC repository, *https://git.haskell.org/ghc.git/commitdiff/0279b*