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
Прикладная математика и информатика

Кафедра информатики и вычислительного эксперимента

ГЕНЕРАЦИЯ ЭКЗЕМПЛЯРОВ КЛАССОВ ТИПОВ НА НА ОСНОВЕ ЭКЗЕМПЛЯРОВ ПРОИЗВОДНЫХ КЛАССОВ В ЯЗЫКЕ HASKELL

Выпускная квалификационная работа
на степень бакалавра

Студентки 4 курса
О. Е. Филиппской

Научный руководитель:
асс. каф. ИВЭ А. М. Пеленицын

Ростов-на-Дону
2016

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Функциональные языки имеют множество неоспоримых достоинств, таких как: повышенная надёжность кода, удобство организации модульного тестирования, возможности оптимизации при компиляции, возможности автоматического распараллеливания вычислений. Поэтому в последнее время языки функционального программирования стремительно развиваются. Одним из наиболее популярных функциональных языков является *Haskell* — язык программирования общего назначения, имеющий полную, сильную, статическую систему типов.

Цель работы

Одной из идиом языка программирования *Haskell* являются так называемые *классы типов*. Классы типов в некотором смысле схожи с интерфейсами объектно-ориентированных языках. В частности, классы типов также реализуют наследование, при котором наследник уточняет функционал предка. Однако класс *Monad*, по смыслу наследовавший класс *Applicative*, синтаксически оставался самостоятельным классом.

В марте 2015 года вышла новая версия компилятора: GHC 7.10. В этой версии было реализовано предложение, известное как *The Applicative Monad Proposal* [1]. Суть этого предложения заключается в том, чтобы сделать класс *Monad* подклассом класса *Applicative*, в ре-

зультате чего получается следующая цепочка наследования: *Functor — Applicative — Monad*. Теперь, описывая экземпляры класса *Monad*, необходимо также создавать экземпляры классов *Functor* и *Applicative*. В версия GHC, младших, чем 7.10 *Monad* был самостоятельным классом, поэтому для получения работоспособного кода в этих экземплярах не было необходимости. Таким образом, в результате этих изменений, часть программ, компилирующихся версиями GHC, младше, чем 7.10, перестали компилироваться версией GHC 7.10.

Постановка задачи

Дописывать необходимые экземпляры вручную непрактично, особенно в случае больших объёмов кода. Поэтому требуется создать программное средство, которое восстановит компилируемость этих программ.

Было принято решение создать утилиту с помощью средств самого языка Haskell, а именно при помощи библиотеки *GHC API* — программного интерфейса компилятора GHC. Утилита должна работать следующим образом:

1) получать синтаксическое дерево программы;
2) отфильтровывать узлы, соответствующие экземплярам класса *Monad*;
3) изменять нужным образом экземпляр класса *Monad* и добавлять, если требуется, экземпляры классов *Applicative* и *Functor*.
4) формировать новый файл с текстом программы с помощью функций структурной печати (*pretty-printing*).

ГЛАВА 1

ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

1.1. GHC: создание, развитие, архитектура

Glasgow Haskell Compiler (GHC) — один из наиболее развитых на сегодняшний день компилятор языка Haskell. Является кроссплатформенным компилятором с открытым исходным кодом. С момента своего создания компилятор постоянно совершенствуется. Сначала GHC был частью академического исследовательского проекта, который спонсировался правительством Великобритании в начале 1990-х годов. Создатели GHC преследовали несколько целей:

1) создать широкодоступный, надёжный компилятор языка *Haskell*, генерирующий высокопроизводительный код;
2) обеспечить модульную основу, которую другие исследователи могли бы развивать и расширять;
3) изучать работу программ, чтобы улучшить дальнейшую разработку компиляторов.

GHC можно разделить на три сущности:

Собственно компилятор. По существу представляет собой программу на языке *Haskell*, которая преобразует исходный код в исполняемый машинный код.

Загрузочные библиотеки (Boot Libraries). GHC создан методом *раскрутки компилятора* (англ. *bootstrapping*) [2], т. е. GHC написан

на самом языке Haskell. GHC комплектуется набором библиотек, называемых *загрузочными*, поскольку на них базируется сам компилятор. Наличие этих библиотек позволяет GHC осуществить раскрутку.

Среда выполнения (The Runtime System, RTS). Написана на языках *C* и *C−−*. Обрабатывает все задачи, связанные с запуском скомпилированного кода на языке *Haskell*.

На самом деле, такое разделение в точности соответствует трём подкаталогам дерева исходных кодов GHC: *compiler*, *libraries* и *rts* соответственно.

Остановимся подробнее на самом компиляторе. Компилятор, в свою очередь, также можно разделить на несколько частей.

Диспетчер компиляции (compilation manager). Отвечает за компиляцию исходного кода, состоящего из нескольких и более модулей. Его задача заключается в том,чтобы определять, в каком порядке следует компилировать модули, а также какие модули перекомпилировать не нужно, если с момента последней компиляции в их зависимости не были внесены изменения.

Компилятор языка Haskell (Haskell compiler, Hsc). Управляет компиляцией одного файла исходного кода. Большинство действий над кодом совершается именно здесь.

Конвейер (pipeline). Отвечает за коллаборацию *Hsc* с необходимыми внешними программами. Например, иногда файл исходного кода сначала должен пройти предварительную обработку через препроцессор C прежде чем быть переданным Hsc. Выходной же файл Hsc — это обычно файл на языке ассемблера, который должен затем быть переданным ассемблеру для получения объектного файла.

1.2. GHC 7.8 — GHC 7.10

Как уже было сказано, GHC — один из наиболее развитых компиляторов языка *Haskell* на сегодняшний день. Обновления для GHC выходят регулярно. К сожалению, иногда выход обновлений приводит к определённым проблемам, как, например, потеря обратной совместимости. В данной работе решается проблема обратной совместимости версий GHC 7.10 и версий, младше 7.10.

Так, одним из изменений, появившихся в GHC 7.10, отличающих его от GHC 7.8 стала реализация предложения, известного как *Applicative Monad Proposal* (*AMP*). Суть его состоит в том, чтобы сделать класс *Applicative* надклассом класса *Monad* (и в дополнение класса *MonadPlus*) [1].

Так, в версиях GHC младших, чем 7.10 *Monad* был самостоятельным классом. Однако по смыслу он уточнял функционал класса *Applicative*. Так, идентичными являются функции:

1) `pure` из *Applicative* и `return` из *Monad*: обе принимают «чистое» значение и помещают его в минимальный контекст по умолчанию;

2) $*>$ из *Applicative* $>>$ из *Monad* работают идентично.

Однако *Monad* «уточнял» функционал *Applicative* с помощью функции $>>=$ (*связывание, bind*). Таким образом, было принято решение сделать класс *Monad* подклассом класса *Applicative* также и синтаксически.

О других изменениях, произошедших в GHC 7.10 по сравнению с GHC 7.8 см *Release notes for version 7.10.1* [3].

1.3. GHC API — программный интерфейс компилятора

Функционал GHC может быть использован не только для компиляции программ на языке Haskell. Важным примером использования может послужить анализ и, возможно, преобразование кода на языке Haskell. Другим примером является динамическая загрузка кода, как в GHCi [4]. Для этих целей функционал GHC доступен через библиотеку GHC API.

Рассмотрим основные модули этой библиотеки, функционал которых применялся в данной работе.

Parser. Данный модуль предоставляет функции, которые проводят лексический анализ (*parsing*) кода на языке *Haskell*. Данные функции принимают файл или строку исходного кода и дают на выходе абстрактное синтаксическое дерево (*abstract syntax tree*, *AST*), либо сообщение об ошибке.

HsSyn. Содержит описание структуры верхнего уровня для модуля (*HsModule*). Объект такого типа данных можно получить, если к файлу с исходным кодом применить функцию **parseModule** из модуля *Parser*.

HsDecls. Здесь определён абстрактный синтаксис для глобальных объявлений, таких как объявление типа, класса, экземпляра и т. д.

HsTypes. В данном модуле описан абстрактный синтаксис для типов, определённых пользователем.

Outputable. В данном модуле определены классы и функции структурной печати (*pretty-printing*). В частности, здесь определён класс **Outputable**: если тип данных имеет экземпляр этого класса, значит, к нему можно применять функции структурной печати. Эти

функции возвращают объект типа данных **SDoc**, который в свою очередь легко преобразовать к строке.

DynFlags. Параметры компиляции. Большинство флагов являются динамическими. Это означает, что они могут изменяться от одного процесса компиляции к другому.

SrcLoc. Данный модуль содержит типы, относящиеся к положению различных элементов в файле исходного кода и позволяющие присваивать метки этом положениям.

Более подробно см. документацию по GHC API [5].

1.4. Экземпляр-сирота (orphan instance)

При решении поставленной задачи необходимо будет убедиться, что у экземпляра класса *Monad* отсутствуют уже описанные соответствующие ему экземпляры классов *Applicative* и *Functor*. Однако в многомодульных проектах может возникнуть ситуация, при которой экземпляры разных классов оказались в разных модулях. Если экземпляр класса типов *C* для типа *T* описан в модуле, в котором не описан ни класс *C*, ни класс *T*, то он называется *экземпляр-сирота* (*orphan instance*) [6].

Проверим, в каком случае наличие такого экземпляра возможно в условиях поставленной задачи. Для этого создадим два простейших модуля: *Moduleone.hs* (см. листинг 1.4.1, с. 11) и *Moduletwo.hs* (см. листинг 1.4.2, с. 12) в первом (*Moduleone.hs*) из них опишем свой тип данных *MyData* и экземпляр класса *Monad* для него. В другом (*Moduletwo.hs*) создадим экземпляры классов *Functor* и *Applicative* для *MyData* и подключим в нём *Moduleone* с описанием *MyData*.

При попытке скомпилировать *Moduletwo.hs* мы получаем сообщение:

```
$ ghc Moduletwo.hs
[1 of 2] Compiling Moduleone
            ( Moduleone.hs, Moduleone.o )

Moduleone.hs:7:10:
    No instance for (Applicative MyData)
      arising from the superclasses of
      an instance declaration
    In the instance declaration for ‘Monad MyData’
```

Это означает, что компиляция *Moduleone.hs* не удалась, поскольку в нём присутствует экземпляр класса *Monad*, но отсутствует требуемый экземпляр класса *Applicative*.

Рассмотрим другой вариант. Теперь в *Moduletwo.hs* опишем тип данных *MyData* и определим для него экземпляры классво *Applicative* и *Functor* (см. листинг 1.4.4, с. 12). В модуле *Moduleone.hs* определим экземпляр класса *Monad* и подключим *Moduletwo.hs* (см. листинг 1.4.3, с. 12). Компиляция в этом случае пройдёт успешно.

Таким образом, в условиях поставленной задачи может возникнуть ситуация, когда экземпляр класса *Monad* описан в одном модуле, а экземпляры классов *Applicative* и *Functor* (а также и сам тип данных) — в другом. Тем не менее, экземпляров-сирот лучше избегать. Поскольку у экземпляров классов типов нет идентификаторов, их нельзя явно исключить: все экземпляры, описанные в некотором модуле импортируются при подключении этого модуля. Поэтому

Листинг 1.4.1. Реализация модуля Moduleone.hs

```
module Moduleone where

import Control.Monad

data MyData a = MyData a
instance Monad MyData
```

Листинг 1.4.2. Реализация модуля Moduletwo.hs

```
module Moduletwo where

import Control.Applicative
import Data.Functor
import Moduleone

instance Functor MyData
instance Applicative MyData
```

Листинг 1.4.3. Реализация модуля Moduleone.hs

```
module Moduleone where

import Control.Monad
import Moduletwo

instance Monad MyData
```

можно описать несколько разных экземпляров одного и того же класса, которые находятся в разных модулях. Однако если в области видимости имеется несколько экземпляров одного и того же класса для одного и того же типа данных, компилятор не может решить, какой из них использовать. Экземпляры, не являющиеся сиротами, позволяют избегать определения множественных экземпляров.

Итак, при решении задачи мы будем рассматривать случай, когда тип данных и экземпляр класса *Monad* для него описаны в одном модуле, т. е. когда экземпляров-сирот возникнуть не может.

Листинг 1.4.4. Реализация модуля Moduletwo.hs

```
module Moduletwo where

import Control.Applicative
import Data.Functor

data MyData a = MyData a

instance Functor MyData
instance Applicative MyData
```

ГЛАВА 2

ОСНОВНЫЕ АЛГОРИТМЫ

2.1. Синтаксическое дерево

Для начала необходимо провести синтаксический анализ (*parsing*) файла с исходным кодом и получить дерево разбора (синтаксическое дерево). Для этого нужно описать функцию, которая принимает имя файла с исходным кодом и возвращает объект типа данных *HsModule* (структура верхнего уровня для модуля). Схематично *HsModule* представлен на рис. 2.1, с. 13.

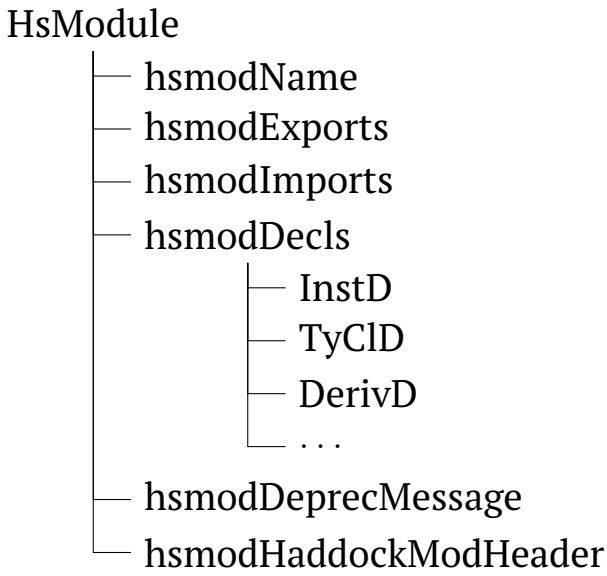

Рисунок 2.1 — Структура верхнего уровня HsModule

2.2. Алгоритм выбора экземпляров класса Monad

Далее получим экземпляры класса *Monad*. Необходимо отфильтровать только те экземпляры, для которых нет соответствующих им экземпляров класса *Applicative* и/или *Functor*.

1. Выберем все объявления. Для этого оставим только ветвь *hsmodDecls* (см. рис. 2.2, с. 15).

2. Выберем только объявления экземпляров (см. замечание 1). Они имеют тип данных *InstDecl*.

3. Оставим только те объекты типа данных *InstDecl* (см. замечание 2), которые были созданы с помощью конструктора *ClsInstDecl* — это и есть экземпляры классов типов.

4. Необходимо выбрать только экземпляры класса *Monad*. Обратимся к конструктору типа данных *ClsInstDecl* (см. рис. 2.4, с. 18). Первое поле этого конструктора `cid_poly_ty` как раз содержит информацию о том, экземпляр какого класса описан.

5. С помощью вышеописанных действий, получим также экземпляры классов *Functor* и *Applicative*.

6. Наконец, отфильтруем только те экземпляры класса *Monad*, для которых нет соответствующих им экземпляров классов *Applicative* и/или *Functor* (то есть те, для которых нет экземпляров классов *Applicative* и *Functor*, созданных для тех же самых типов данных).

Замечание 1. Каждое объявление описывается типом данных *HsDecl*. *HsDecl* в свою очередь имеет несколько конструкторов:

1) `TyClD (TyClDecl id)` для классов и типов, определённых пользователем;

2) `InstD (InstDecl id)` для объявлений экземпляров;

3) `ValD (HsBind id)` для функций, констант и т.п.;

4) и др.

Замечание 2. У типа данных *InstDecl* также имеется несколько конструкторов:

1) `ClsInstD {cid_inst :: ClsInstDecl name}`

2) `DataFamInstD {dfid_inst :: DataFamInstDecl name}`

3) `TyFamInstD {tfid_inst :: TyFamInstDecl name}`

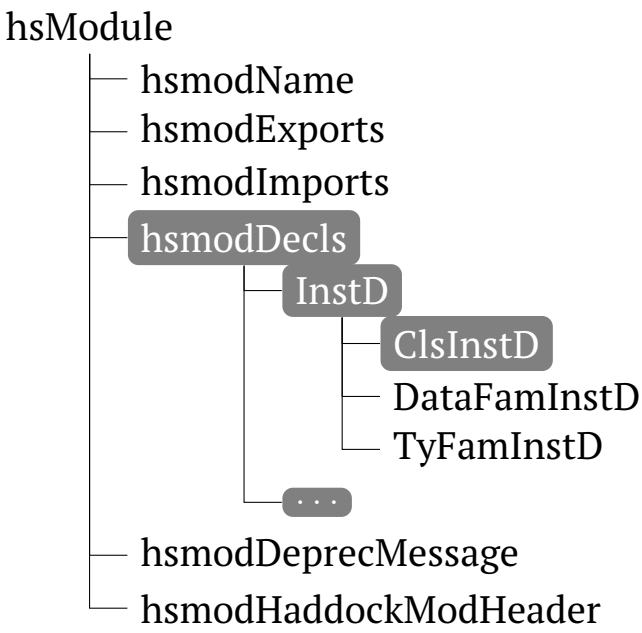

Рисунок 2.2 — Экземпляры классов в синтаксическом дереве

2.3. Алгоритм генерации экземпляров классов Applicative и Functor

2.3.1. Подключение модулей

Сначала убедимся, что подключены стандартные модули, содержащие реализацию:

1) класса Functor;
2) класса Applicative.

1. Выберем все подключения (*imports*). Для этого возьмём ветвь *hsmodImports*. Каждое подключение описывается типом данных *ImportDecl* (см. рис. 2.3, с. 16).
2. Рассмотрим конструктор типа данных *ImportDecl*. Нас интересует поле *ideclName* — это имя импортируемого модуля. Проверим, есть ли среди них *Data.Functor* либо *Control.Applicative*.
3. Если интересующие нас имена модулей не найдены, то их импорты необходимо добавить. Сформируем соответствующие объекты типа данных *ImportDecl* и добавим их к уже имеющимся.

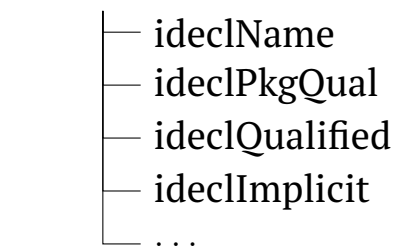

Рисунок 2.3 — Структура данных для импорта

2.3.2. Непосредственно генерация экземпляров

Необходимые экземпляры будем генерировать в соответствии с рекомендациями на *GHC Developer Wiki* [7].

Экземпляр класса Applicative. Для каждого экземпляра класса *Monad* создадим объект типа *ClsInstDecl* по следующему правилу.

1. Поскольку функции `pure` из *Applicative* и `return` из *Monad* идентичны (см главу 1.2), то определение функции `return` перенесём в определение для `pure`. Функцию `return` определим в терминах `pure`.

2. Аналогичным образом поступим с функцией `(*>)` из класса *Applicative* и `(>>)` из *Monad*.

3. Заменим «старые» экземпляры класса *Monad* экземплярами, полученными в предыдущих пунктах. Добавим экземпляры класса *Applicative* к уже имеющимся.

Экземпляр класса Functor. Для каждого экземпляра класса *Monad* создадим объект типа *ClsInstDecl* по следующему правилу.

1. Функцию `fmap` из класса `Functor` определим в терминах `liftM` (эта функция описана в стандартном модуле *Control.Monad*).

2. Добавим очередной экземпляр в ветвь к уже имеющимся.

Наконец, присоединим описания экземпляров классов к остальным объявлениям (*hsmodDecls*) модуля.

2.4. Алгоритм формирование выходного файла

Наконец, необходимо сформировать новое синтаксическое дерево, по которому затем сгенерировать уже работоспособный код.

1. Заменим ветви *hsmodImports* и *hsmodDecls* (рис. 2.1, с. 13) ветвями, полученными в предыдущих пунктах.

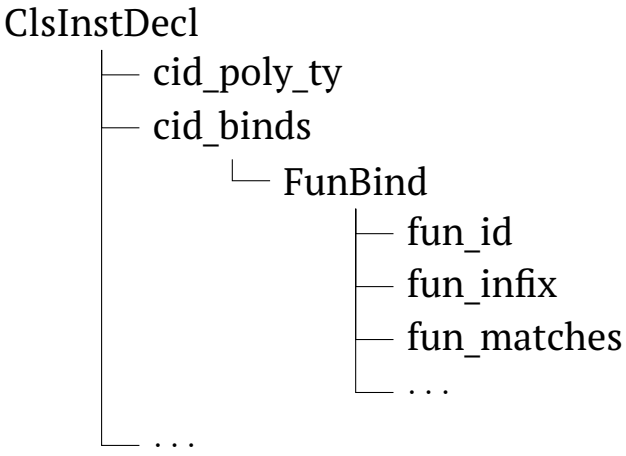

Рисунок 2.4 — Структура данных для экземпляра класса

2. Сгенерируем код из синтаксического дерева при помощи функций структурной печати.

3. Новый код перезапишем в поданный утилите файл. При этом, если исходный код уже был работоспособным, то выведем соответствующее сообщение и изменять файл не будем.

2.5. Другие способы решения

Для решения данной задачи можно предложить и другие способы.

1. Установить расширение для компилятора *Deriving Functor*, которое позволяет генерировать экземпляр класса Functor автоматически с помощью ключевого слова `deriving`. Однако проблема с экземплярами класса *Applicative* остаётся открытой, кроме того, код программ всё равно придётся править вручную, что непрактично в случае больших объёмов исходного кода.

2. Написать скрипт, который анализирует текст программы на языке Haskell, используя средства для работы со строками. Данный скрипт должен находить ключевые слова **`instance`** `Monad` ,

а затем дописывать в нужном месте необходимые определения экземпляров других классов. Такой способ также имеет недостатки, поскольку слишком сложно учесть все особенности двумерного синтаксиса языка *Haskell* для того, чтобы на выходе получить работающий код. Кроме этого, сложность представляет проверка наличия уже описанных экземпляров, особенно в случае объёмных проектов, содержащих большое число модулей.

ГЛАВА 3

ОБЗОР ПРОГРАММНОЙ РЕАЛИЗАЦИИ

Программный код разбит на несколько модулей.

MyParser.hs: содержит функции, проводящие синтаксический разбор модулей и строк.

AppendImport.hs: в данном модуле описаны функции, решающие задачу импорта дополнительных модулей.

AppendInstance.hs: здесь определены функции, формирующие необходимые экземпляры классов типов.

Manipulation.hs: содержит некоторые вспомогательные функции.

FixingMonad.hs: здесь находится код основной программы (*main*). С кодом программы можно ознакомиться в репозитории с исходными кодами [8].

3.1. Получение синтаксического дерева

Для получения синтаксического дерева опишем функцию `parseHaskell`(см. листинг 3.1.1, с. 21), принимающую имя файла и возвращающую объект типа данных *HsModule*. Воспользуемся функциями синтаксического разбора из модуля *Parser* (см. главу 1.3), а именно функцией *parseModule*. Инициализируем состояние парсера (*PState*) при помощи функции `mkPState`. Её аргументы:

1) `dynFl`: динамические флаги (см. главу 1.3);

2) `sbuf`: строка, подвергнутая синтаксическому разбору (в данном случае модуль с исходным кодом целиком);

3) `srcloc`: расположение в файле (поскольку файл подвергается синтаксическому разбору целиком, то расположение — это сам файл).

Листинг 3.1.1. Функция parseHaskell

```
parseHaskell :: FilePath -> IO (Maybe (HsModule RdrName))
parseHaskell file = do
    initStaticOpts
    dynFl <- GHC.runGhc (Just libdir) GHC.getSessionDynFlags
    sbuf <- hGetStringBuffer file
    let srcloc = mkRealSrcLoc (mkFastString file) 1 1
    return $ case unP Parser.parseModule (mkPState dynFl sbuf
       srcloc) of
        POk _ (L _ mdl) -> Just mdl
        PFailed _ _     -> Nothing
```

3.2. Фильтр экземпляров класса Monad

Следуя алгоритму, описанному в главе 2.2, опишем следующие функции.

1. Функция `getHsDecls` (см. листинг 3.2.1, с. 22) выбирает только ветвь с объявлениями.

2. Функция `getInstDecls` (см. листинг 3.2.2, с. 23) фильтрует только те объявления, которые являются объявлениями экземпляров. Предикат `isInstDecl` проверяет, действительно ли очередное объявление описывает экземпляр. Вспомогательная функция `getOneInstDecl` «достаёт» отдельно взятое объявление из конструктора.

3. Функция `getClsInstDecl` (см листинг 3.2.3, с. 23) отбирает только определения экземпляров классов при помощи предиката `isClsInstDecl`, который устанавливает, является ли данный экземпляр экземпляром класса типов.

4. Функция `getInstsMonad` (см листинг 3.2.4, с. 24) фильтрует экземпляры класса *Monad* при помощи соответствующего предиката `isInstanceMonad`.

5. Получение экземпляров *Functor* и *Applicative* происходит абсолютно аналогично.

6. Функция `filterAloneInstMon` (см. листинг 3.2.5, с. 25) принимает список экземпляров класса *Monad* и *Applicative* либо *Functor*. С помощью предиката `hasntApprInst` проверяется, есть ли у текущего экземпляра класса *Monad* соответствующий ему экземпляр класса *Applicative* (либо *Functor*).

Листинг 3.2.1. Выбор всех объявлений, описанные в модуле

```
getHsDecls :: IO (Maybe [LHsDecl RdrName])
                            -> IO (Maybe [HsDecl RdrName])
getHsDecls md = do
    mp <- md
    case mp of
        Nothing  -> return Nothing
        Just mdl -> return $ Just (map unLoc $ hsmodDecls mdl)
```

3.3. Генерация экземпляров классов Applicative и Functor

3.3.1. Импорт модулей

Если интересующие нас модули оказались не подключены, то необходимо сформировать соответствующие импорты (объекты типа

Листинг 3.2.2. Выбор всех объявлений экземпляров из модуля

```
isInstDecl :: HsDecl RdrName -> Bool
isInstDecl (InstD _) = True
isInstDecl _         = False

getOneInstDecl :: HsDecl RdrName -> InstDecl RdrName
getOneInstDecl (InstD (a)) = a

getInstDecls :: IO (Maybe [HsDecl RdrName])
                  -> IO (Maybe [InstDecl RdrName])
getInstDecls md = do
    mp <- md
    case mp of
        Nothing  -> return Nothing
        Just mdl -> return $ Just (map getOneInstDecl
                                   $ filter isInstDecl mdl)
```

Листинг 3.2.3. Выбор всех объявлений экземпляров классов типов

```
isClsInstDecl :: InstDecl RdrName -> Bool
isClsInstDecl (ClsInstD _ ) = True
isClsInstDecl _             = False

getClsInstDecl :: IO (Maybe [InstDecl RdrName])
                    -> IO (Maybe [ClsInstDecl RdrName])
getClsInstDecl md = do
    mp <- md
    case mp of
        Nothing  -> return Nothing
        Just mdl -> return $ Just (map cid_inst
                                     $ filter isClsInstDecl mdl)
```

данных *ImportDecl*). Определим функцию `mkImport`(см. листинг 3.3.1, с. 26), которая принимает строку (название модуля) и возвращает объект типа данных *ImportDecl*. Объект создаётся путём заполнения полей соответствующего конструктора. Поскольку мы формируем простейший импорт модуля вида «**`import`** `ModuleName`», то все поля (кроме имени модуля) заполним константами, выражающими отсутствие значения.

Листинг 3.2.4. Выбор всех объявлений экземпляров класса Monad

```
isInstanceMonad :: ClsInstDecl RdrName -> Bool
isInstanceMonad instD =
  ((hsTypeToString . whatClassIsInstance) instD) == "Monad"

getInstsMonad :: IO (Maybe [ClsInstDecl RdrName])
                    -> IO (Maybe [ClsInstDecl RdrName])
getInstsMonad md = do
    mp <- md
    case mp of
        Nothing  -> return Nothing
        Just mdl -> return $ Just (filter isInstanceMonad mdl)
```

3.3.2. Генерация экземпляров

Для начала опишем общую функцию `mkInstance` (см. листинг 3.3.2, с. 26), которая принимает заголовок экземпляра, функции, определённые в экземпляре, и возвращает объект типа данных *ClsInstDecl*. Затем определим функции, которые по правилам, описанным в главе 2.3.2, составляют список `cid_binds`.

Экземпляр класса Applicative. Для экземпляра класса *Applicative* определим функцию `mkCIDBindsForInstAppl` (см. листинг 3.3.3, с. 27), которая работает следующим образом. Она принимает два аргумента: определения функций `return` и `>>` из класса *Monad*. Если реализация одной или обеих функций в экземпляре класса *Monad* отсутствует, то соответствующая фукнция в классе *Applicative* также не появится.

Экземпляр класса Functor. Для данного экземпляра зададим функцию `mkCIDBindsForInstFunc` (см. листинг 3.3.4, с. 27). Он не зависит от экземпляра класса *Monad*, поэтому `cid_binds` формируются всегда одним и тем же образом.

Листинг 3.2.5. Выбор всех объявлений экземпляров класса Monad, у которых отсутствуют соответствующие им экземпляры классов Applicative и/или Functor

```
hasntApprInst :: [ClsInstDecl RdrName] -> ClsInstDecl RdrName
                                       -> Bool
hasntApprInst xs x = not $ elem (showSDocUnsafe
                        $ ppr $ takeUserData x)
                  (map (showSDocUnsafe . ppr . takeUserData) xs)

filterAloneInstMon :: Maybe [ClsInstDecl RdrName]
                        -> Maybe [ClsInstDecl RdrName]
                        -> Maybe [ClsInstDecl RdrName]
filterAloneInstMon mpap md =
    case md of
        Nothing  -> Nothing
        Just mdl -> Just (filter
                       (hasntApprInst (unMaybeList mpap)) mdl)
```

3.4. Формирование выходного файла

Сформировав новое синтаксическое дерево, применим к нему функцию структурной печати `ppr`, которая находится в модуле *Outputable* библиотеки GHC API (см. главу 1.3). Данная функция принимает объект, тип данных которого имеет экземпляр класса *Outputable* и возвращает объект типа данных *SDoc*. Последний, в свою очередь, преобразовывается к строке с помощью специальных функций, например, *showSDoc*, *showSDocUnsafe* и других.

Опишем основную функцию (`main`) (см. листинг 3.4.1, с. 27). Она работает следующим образом: принимает имя файла в качестве аргумента командной строки, проводит его синтаксический анализ, применяет к нему функцию `appendAllInstances` (является композицией вышеописанных функций), затем печатает в результат в этот же файл.

Листинг 3.3.1. Формирование узла AST GHC с импортом модуля

```
mkImport :: String -> LImportDecl RdrName
mkImport module =
             noLoc $ ImportDecl { ideclSourceSrc = Nothing
                                , ideclName      = noLoc
                                $ mkModuleName module
                                , ideclPkgQual   = Nothing
                                , ideclSource    = False
                                , ideclSafe      = False
                                , ideclQualified = False
                                , ideclImplicit  = False
                                , ideclAs        = Nothing
                                , ideclHiding    = Nothing }
```

Листинг 3.3.2. Формирование узла AST GHC с экземпляром

```
mkInstance :: LHsType RdrName -> LHsBinds RdrName
                                  -> ClsInstDecl RdrName
mkInstance insthead funs = ClsInstDecl {
                             cid_poly_ty       = insthead
                           , cid_binds         = funs
                           , cid_sigs          = []
                           , cid_tyfam_insts   = []
                           , cid_datafam_insts = []
                           , cid_overlap_mode  = Nothing
                             }
```

Листинг 3.3.3. Формирование поля cid_binds для Applicative

```
mkCIDBindsForInstAppl :: [HsBindLR RdrName RdrName]
                -> [HsBindLR RdrName RdrName] -> LHsBinds RdrName
mkCIDBindsForInstAppl [] [] = listToBag
          [ noLoc (mkFunBind (mkFunId "(<*>)")
                      ([mkLMatch [] $ mkGRHSs "ap"])) ]

mkCIDBindsForInstAppl [] mongr = listToBag
          [ noLoc (mkFunBind (mkFunId "(<*>)")
                      ([mkLMatch [] $ mkGRHSs "ap"]))
          , noLoc (mkFunBindForAppl "*>" (head mongr) True) ]

mkCIDBindsForInstAppl monret [] = listToBag
          [ noLoc (mkFunBindForAppl "pure" (head monret) False)
          , noLoc (mkFunBind (mkFunId "(<*>)")
                       ([mkLMatch [] $ mkGRHSs "ap"])) ]

mkCIDBindsForInstAppl monret mongr = listToBag
          [ noLoc (mkFunBindForAppl "pure" (head monret) False)
          , noLoc (mkFunBind (mkFunId "(<*>)")
                       ([mkLMatch [] $ mkGRHSs "ap"]))
          , noLoc (mkFunBindForAppl "*>" (head mongr) True) ]
```

Листинг 3.3.4. Формирование поля cid_binds для Functor

```
mkCIDBindsForInstFunc :: LHsBinds RdrName
mkCIDBindsForInstFunc = listToBag
    [ noLoc (mkFunBind (mkFunId "fmap")
                        ([mkLMatch [] $ mkGRHSs "liftM"])) ]
```

Листинг 3.4.1. Основная функция (main)

```
main :: IO ()
main = do
    dynFl <- GHC.runGhc (Just libdir) GHC.getSessionDynFlags
    [file] <- getArgs
    mp <- appendAllInstances $ parseHaskell file
    case mp of
        Nothing  -> putStrLn "Parse failed"
        Just mdl -> writeFile "a.hs"
                $ showSDocUnsafe $ ppr mdl
```

ГЛАВА 4

РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

Подадим на вход утилите тестовый файл *Test.hs* и проанализируем результаты. Данный файл содержит несколько примеров экземпляра класса *Monad*.

Рассмотрим первый пример: «пустой» экземпляр класса *Monad*, т. е. содержащий только заголовок.

Исходные данные:

```
data MyData a = MyData a

instance Monad MyData
```

Результат:

```
data MyData a = MyData a

instance Applicative MyData
   where
  (<*>) = ap
instance Functor MyData where
  fmap = liftM
instance Monad MyData where
  return = pure
  (>>) = (*>)
```

Результаты соответствуют ожидаемым: программой были добавлены экземпляры классов *Functor* и *Applicative*. Реализация функций `pure` и `*>` отсутствует, поскольку в экземпляре монады соответствующие идентично работающие функции реализованы не были.

Теперь рассмотрим пример, содержащий экземпляр класса *Monad* с реализованными в нём функциями `retrun` и `>>=`. Кроме то-

го, в исходном файле уже присутствовал экземпляр класса *Functor*, в котором была определена функция `fmap`.

Исходные данные:

```
newtype Prob a = Prob {
        getProb :: [(a,
           Rational)]
        } deriving Show

instance Functor Prob where
    fmap f (Prob xs) = Prob
            $ map (\(x,p)
                -> (f x,p))
                   xs

flatten :: Prob (Prob a) ->
   Prob a
flatten (Prob xs) =
        Prob $ concat
             $ map multAll xs
    where
        multAll (Prob innerxs
           ,p) =
          map (\(x,r)
             -> (x,p*r))
                innerxs

instance Monad Prob where
  return x = Prob [(x,1%1)]
  m >>= f
    = flatten (fmap f m)
```

Результат:

```
newtype Prob a = Prob {
  getProb :: [(a, Rational)]
  } deriving (Show)

instance Functor Prob where
  fmap f (Prob xs) = Prob
        $ map (\ (x, p)
            -> (f x, p)) xs
flatten :: Prob (Prob a) ->
   Prob a
flatten (Prob xs)
  = Prob $ concat
         $ map multAll xs
  where
      multAll (Prob innerxs,
         p)
        = map (\ (x, r)
            -> (x, p * r))
               innerxs

instance Applicative Prob
   where
  pure x = Prob [(x, 1 % 1)]
  (<*>) = ap

instance Monad Prob where
  return = pure
  (>>) = (*>)
  m >>= f
    = flatten (fmap f m)
```

Результаты соответствуют ожидаемым:

1) «лишний» экземпляр класса *Functor* не появился;
2) экземпляр класса *Applicative* был сгенерирован в соответствии с требованиями (см. главу 2.3);
3) экземпляр класс *Monad* был изменён нужным образом;
4) изменения не затронули код, не относящийся к проблеме.

ЗАКЛЮЧЕНИЕ

В ходе работы над задачей было получено решение с применением библиотеки GHC API — программного интерфейса компилятора. Проанализировав результаты, можно прийти к следующим выводам.

Достоинства

С помощью данного способа решения решается проблема обратной совместимости версий GHC младше, чем 7.10 и GHC 7.10. Поскольку были использованы стандартные функции библиотеки, которые учитывают все особенности двумерного синтаксиса языка *Haskell*, на выходе получается код, который гарантированно скомпилируется.

Недостатки

Однако у данного подхода имеются и недостатки.

1. Если в исходном файле присутствовали комментарии, то в выходном файле они не сохраняются, поскольку игнорируются при синтаксическом анализе. Однако данная утилита была создана в помощь программистам, которые хотят воспользоваться какими-либо сторонними библиотеками, то есть отсутствие комментариев в этом случае не так важно.

2. Гораздо более существенным недостатком является то, что конструкции **`case`** `...` **`of`** на выходе получаются синтаксически неверными, а именно, между вариантами отсутствует точка с запятой (см. ниже). Это связано с наличием ошибки в исходном коде самого компилятора: функции структурной печати (*pretty printing*), описанные в GHC API, используют вариант конструкции с фигурными скобками, который предполагает наличие точек с запятой, однако они не добавляются. Тем не менее, эта проблема в данный момент решается: нами инициирован процесс добавления соответствующих изменений в общедоступный код компилятора GHC.

```
f mp = case mp of {
           Nothing -> 1 -- missing ;
           Just _ -> 2 }
```

СПИСОК ЛИТЕРАТУРЫ

1. Functor-Applicative-Monad Proposal. — URL: https://wiki.haskell.org/Functor-Applicative-Monad_Proposal (дата обр. 18.06.2016).
2. Wikipedia. The Free Encyclopedia. Glasgow Haskell Compiler. — URL: https://en.wikipedia.org/wiki/Glasgow_Haskell_Compiler (дата обр. 18.06.2016).
3. Release notes for version 7.10.1. — URL: https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/release-7-10-1.html (дата обр. 26.06.2016).
4. GHC/As a library. — URL: https://wiki.haskell.org/GHC/As_a_library (дата обр. 18.06.2016).
5. ghc-8.0.1: The GHC API. — URL: http://downloads.haskell.org/~ghc/latest/docs/html/libraries/ghc-8.0.1/index.html (дата обр. 18.06.2016).
6. Haskell Wiki. Orphan instance. — URL: https://wiki.haskell.org/Orphan_instance (дата обр. 22.06.2016).
7. GHC 7.10.x Migration Guide. — URL: https://ghc.haskell.org/trac/ghc/wiki/Migration/7.10 (дата обр. 22.06.2016).
8. Репозиторий с исходными кодами. — URL: https://github.com/Valoisa/Generate-ancestor-instances (дата обр. 22.06.2016).