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

ОБОБЩЁННОЕ ПРОГРАММИРОВАНИЕ В ЗАДАЧЕ ПОСТРОЕНИЯ ЗИППЕРОВ

Курсовая работа

Студентки 3 курса
А. С. Болотиной

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

_____ _____

оценка (рейтинг) подпись руководителя

Ростов-на-Дону
2017

Содержание

Введение

Идея обобщённого программирования (*generic programming*) состоит в том, чтобы увеличить гибкость языков программирования, расширяя возможности для параметризации программ типами данных. Термин *обобщённое программирование* может иметь различное значение в зависимости от контекста. Говоря об объектно-ориентированных языках, под ним обычно понимают использование параметрического полиморфизма в стиле системы Хиндли—Милнера [1], библиотеки обобщённых алгоритмов и структур данных или метапрограммирование [2]. Тем не менее, когда говорят о функциональном программировании, его определение относится к структурному полиморфизму [1]. Это означает, что функции определяются над структурой типов данных. В данной работе рассматривается *обобщённое программирование типов данных* (*datatype generic programming*) [2], то есть возможность перевода типа в некоторое специальное *представление* (*representation*) его внутренней структуры и написания функций, параметризованных такими представлениями и способных единообразно их обрабатывать.

В работе изучается подход к обобщённому программированию типов данных на языке Haskell, описанный в статье [3] и реализованный в библиотеке generics-sop [4]. Исследуется возможность применения средств подхода в задаче построения *зипперов* — структур данных, используемых для эффективной, чисто функциональной навигации по древовидной структуре.

Раздел 1 включает предварительные сведения об обобщённом программировании типов данных. В разделе 2 содержится описание структуры данных Зиппер, а также задачи, решаемой при помощи этой структуры, излагается идея дифференцирования типов данных и описывается его связь с типом зиппера, приводится набор правил дифференцирования, который является результатом статьи [5], позволяющий механизировать получение зиппера, и рассматривает-

ся более ранний известный подход к реализации обобщённого зиппера, представленный в статье [6]. Этот подход использует средства библиотеки обобщённого программирования multirec [7], недостаток которой заключается в необходимости переводить данный тип в его обобщённое представление вручную или используя расширение языка Template Haskell [8], которое является тяжеловесным средством метапрограммирования. В работе показано, что библиотека generics-sop позволяет генерировать такое представление автоматически, используя встроенные возможности компилятора GHC.

В разделе 3 представлен обзор возможностей библиотеки generics-sop: описываются идеи и особенности нового подхода к обобщённому программированию, обсуждаются преимущества предлагаемой формы структурного представления типов и техники перевода старого представления в новое.

В результате исследования получен механизм, позволяющий автоматизировать процесс построения обобщённого представления зиппера средствами данной библиотеки, он описывается в разделе 4.

1. Введение в обобщённое программирование на языке Haskell

Область применения обобщённого программирования типов данных включает большое количество функций в языке Haskell, которые могут быть определены систематическим образом для различных типов: функции проверки на равенство и сравнения, различные виды преобразований между значениями типов и другими форматами представления данных (JSON, XML и т. д.), функции обхода или навигации по структурам данных, а также доступа к конкретным данным внутри структуры (линзы) и другие.

Чтобы раскрыть идею обобщённого программирования, рассмотрим следующий пример. Предположим, что у нас есть тип дан-

ных A, опишем функцию проверки на равенство для этого типа (листинг 1).

```
eq_A :: A -> A -> Bool
```

Листинг 1: Функция проверки на равенство для типа A

Определение функции $\texttt{eq}_{\texttt{A}}$ несложно дать, имея определение типа A. Алгоритм неформально можно описать так: если тип данных имеет несколько конструкторов, необходимо проверить, что для двух аргументов выбран один и тот же конструктор, и если это так, то сравнить поэлементно на равенство аргументы конструктора.

Например, так определяется функция $\texttt{eq}_{\texttt{Bool}}$ для элементарного типа данных **Bool**, который имеет два конструктора без аргументов (листинг 2).

```
data Bool = True | False

eq_Bool :: Bool -> Bool -> Bool
eq_Bool True  True  = True
eq_Bool False False = True
eq_Bool _     _     = False
```

Листинг 2: Функция проверки на равенство для типа **Bool**

Предположим теперь, что есть класс типов (листинг 3), который

```
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

Листинг 3: Класс типов, представимых в обобщённом виде

связывает тип a, любой экземпляр этого класса, с изоморфным ему типом обобщённого представления Rep a, определяя преобразование между ними с помощью функций from и to.

Теперь, если все типы `Rep a` имеют общую структуру, можно ввести класс типов, определяющий функцию сравнения на равенство `geq`, которая работает для всех типов представления по индукции над их структурой, и определить с её помощью функцию `eq` для любых типов, представимых в обобщённом виде (листинг 4).

```
class GEq repA where
  geq :: repA -> repA -> Bool

eq :: (Generic a, GEq (Rep a)) => a -> a -> Bool
eq x y = geq (from x) (from y)
```

Листинг 4: Определение обобщённой функции сравнения на равенство

1.1. Алгебраические типы данных

Алгебраические типы данных (АТД, *algebraic data types*) — основной механизм, использованный для реализации структур данных в функциональных языках программирования. Они определяются как составные типы, которые могут быть представлены в виде типов-сумм из типов-произведений. Тип-произведение соответствует декартову произведению множеств значений типов, а тип-сумма в теории множеств совпадает с дизъюнктным, или размеченным, объединением, то есть множеством, элементами которого являются пары, состоящие из метки (соответствующей конструктору) и сопоставляемого с ней типа-произведения (представляющего аргументы конструктора).

Приведём наиболее простые примеры алгебраических типов данных. Базовыми случаями являются единичный тип, то есть тип, состоящий из одного конструктора без аргументов, нулевой тип — тип, не имеющий конструкторов, и тип-константа, единственный конструктор которого принимает один аргумент. Примеры таких типов представлены в листинге 5. Слева от знака = стоят конструкторы

типов, а справа — конструкторы значений. Примеры элементарных типа-суммы и типа-произведения приведены в листингах 6–7.

```
data Unit = Unit

data Zero

data Const a = Const a
```

Листинг 5: Базовые АТД: единичный тип, нулевой тип и тип-константа

```
data Either a b = Left a | Right b
```

Листинг 6: Тип-сумма

```
data (,) a b = (,) a b
```

Листинг 7: Тип-произведение

Более сложным примером является рекурсивный тип бинарного дерева `Tree`, определяемый с двумя конструкторами: `Leaf` для листа и `Node` для узла, содержащего два корневых узла его поддеревьев (листинг 30). Тип `Tree` представляет собой тип-сумму единичного типа и типа-произведения двух типов-констант.

Вернёмся к примеру с определением обобщённой функции `eq` (см. листинг 4), рассмотренному в начале раздела. Теперь можно, используя структуру алгебраических типов данных, определить обобщённое представление для любого типа, являющегося АТД. Определим, например, экземпляр класса `Generic` (см. листинг 3) для типа **`Bool`** — типа-суммы двух единичных типов.

Для того, чтобы построить обобщённое представление, соответствующее структуре типа **`Bool`**, нам понадобится ввести два типа-комбинатора для суммы и единицы (листинг 9).

```
data Tree = Leaf | Node Tree Tree
```

Листинг 8: Рекурсивный тип бинарного дерева

```
data U          = Unit
data (a :+: b) = L a | R b
```

Листинг 9: Комбинаторы для единичного типа и типа-суммы

Определение типа обобщённого представления `Rep Bool` с функциями `from` и `to` через введённые комбинаторы выглядит, как в листинге 10.

```
instance Generic Bool where
  type Rep Bool = U :+: U
  from True     = L Unit
  from False    = R Unit
  to   (L Unit) = True
  to   (R Unit) = False
```

Листинг 10: Определение обобщённого представления для типа **Bool**

Теперь можно определить экземпляры класса `GEq` из листинга 4 для типов-комбинаторов (листинг 11). В итоге функция `eq` будет работать для типа **Bool** и для любых алгебраических типов, составленных из сумм и единиц и являющихся экземплярами класса `Generic`.

1.2. Функторы

Под термином *функтор* в данной работе понимается отображение, определённое на типах. Заметим, что это соответствует теоретико-категорному понятию функтора как отображения между категориями — областью и кообластью в данном случае является категория **Hask** типов языка Haskell, таким образом функтор, действуя на типы, переводит их в другие типы — объекты этой категории. Фор-

```
instance (GEq a, GEq b) => GEq (a :+: b) where
  geq (L x) (L y) = geq x y
  geq (R x) (R y) = geq x y
  geq _     _     = False
instance GEq U where
  geq Unit Unit = True
```

Листинг 11: Определение работы `geq` для типов-сумм и единичных типов

мальное определение функтора в теории категорий также требует выполнения двух уравнений, называемых в Haskell «законами функтора», — сохранения единичного морфизма и композиции. Однако в рамках этой работы используется нестрогое определение функтора и не требуется соблюдение этих свойств.

С программистской точки зрения, функтор — это полиморфный тип, то есть тип, параметризованный другим типом. В языке Haskell введён дополнительный уровень абстракции над типами — виды, или сорта, типов (*kinds*) [9]. Вид `*` соответствует всем типам, значениями которых являются термы. Полностью применённая форма любого типа (если он параметризован), определённого через ключевое слово **`data`**, имеет вид `*`. Если `k` и `l` — виды, то типы вида `l`, параметризованные типами вида `k`, будут иметь вид `k -> l`.

Например, типы **`Int`** и список `[`**`Int`**`]` — вида `*`, а тип `[]` — неприменённая форма параметризованного типа списка, представленного в листинге 12, — вида `* -> *`.

```
data [] a = [] | a : [a]
```

Листинг 12: Полиморфный тип списка

Функторы имеют вид `* -> *`. Тип списка является примером функтора.

Для того, чтобы строить обобщённые представления любых функторов, необходимо ввести новую систему комбинаторов, как в листинге 13. Типы `K a`, `U`, `f :+: g` и `f :×: g` соответствуют типу-кон-

станте, единичному типу и типам суммы и произведения (см. листинги 5–7); дополнительно они параметризованы типом `x` и имеют вид `* -> *`, так как это требуется для функтора. Тип `I` позволяет обобщённо представить параметр функтора.

```
data K a       x = K a
data I         x = I x
data U         x = Unit
data (f :+: g) x = L (f x) | R (g x)
data (f :×: g) x = f x :×: g x
```

Листинг 13: Типы-комбинаторы для обобщённого представления функторов

С помощью новых комбинаторов можно представлять не только функторы, но и вообще любые алгебраические типы данных. Такой подход к построению обобщённого представления структуры типов используется в [6].

Для определения обобщённого представления функторов требуется также новый класс `Generic1` (листинг 14), в котором тип `Rep1 f` — вида `* -> *`. В листинге 15 приводится в качестве примера представление функтора `Pair Int`.

```
class Generic1 f where
  type Rep1 f :: * -> *
  from1 :: f p -> Rep1 f p
  to1   :: Rep1 f p -> f p
```

Листинг 14: Класс обобщённо представимых функторов

```
data Pair a b = Pair a b

instance Generic1 (Pair Int) where
  type Rep1 (Pair Int) = K Int :×: I
  ...
```

Листинг 15: Пример типа обобщённого представления функтора

Необходимость введения отдельного класса для представления функторов видится существенным недостатком, однако при рассмотренном подходе для этого не существует другой возможности. Такой способ принят в старой технологии обобщённого программирования GHC generics [10], реализованной в компиляторе GHC.

2. Зипперы

Многие эффективные алгоритмы, применяемые в императивном программировании, используют деструктивные операции над элементами структур данных. Однако в случае неизменяемой структуры сложность таких алгоритмов становится $\Theta(\log n)$ или $\Theta(n)$ при наивном решении, когда копируется всё дерево.

Структура данных *Зиппер* — эффективное решение задачи чисто функциональной навигации по древовидной структуре, предложенное Ж. Юэ [11]. Задача формулируется как представление древовидной структуры данных вместе с фокусом на текущем узле, который может перемещаться влево, вправо, вниз и вверх по этой структуре.

Решение состоит в построении производной структуры, содержащей фокус, который указывает на текущий узел, и хранящей путь, восходящий от него к корню дерева. Элементы чисто функциональной структуры, представленной таким образом, можно изменять за константное время.

Зиппер является паттерном проектирования [12], который можно использовать при реализации текстового редактора, где точкой фокуса будет текущее положение курсора, файловой системы (фокус — рабочий каталог), компилятора или интерактивного средства доказательства теорем. В качестве примера использования зиппера в промышленном ПО можно привести оконный менеджер xmonad [13].

2.1. Тип зиппера

Суть решения задачи навигации заключена в том, чтобы, выбирая направление для перемещения по дереву, на каждом уровне сохранять окружающий его *контекст*, то есть все соседние для данного узлы. Тогда после каждого шага можно возвращаться вверх к корню, реализовав дополнительно функцию вставки (*plugging in*) узла в контекст. Путь к корню, таким образом, строится как список контекстов для каждого уровня дерева и выбранного направления.

Рассмотрим следующий пример типа терма в абстрактном синтаксическом дереве некоторого языка (листинг 16).

```
data Term = Var String
          | Lambda String Term
          | App Term Term
          | If Term Term Term
```

Листинг 16: Тип терма

Предположим, что у нас имеется построенное дерево с узлами этого типа, как показано в листинге 17. Значение `term` является

```
term :: Term
term = If (Var "false")
          (App (Lambda "x" (Var "x"))
               (Var "38"))
          (Var "13")
```

Листинг 17: Пример дерева разбора терма

корнем этого дерева. Чтобы спуститься вниз от корня, сфокусировавшись, например, на подтерме со значением `Var "x"`, для восстановления пути к корню требуется на первом шаге — опустившись до значения `App (Lambda "x" (Var "x")) (Var "38")` — сохранить окружающий контекст этого элемента, то есть левый и правый соседние узлы, затем сохранить, добавив в список, соседний узел подтерма

`Lambda "x" (Var "x")` — значение `Var "13"` — и, наконец, на последнем шаге — константу `"x"`, которая является контекстом для выбранного узла.

Тип зиппера для терма и составляющий его тип контекста, соответствующие реализации изложенной идеи, приведены в листингах 18–19. Индексы конструкторов контекста указывают на выбранное направление, то есть порядковый номер дочернего узла, пройденного на текущем уровне в пути от корня к фокусу.

```
type TermZipper = (Term, [TermContext])
```

Листинг 18: Тип зиппера для терма

```
data TermContext = Lambda₁ String
                 | App₁ Term
                 | App₂ Term
                 | If₁ Term Term
                 | If₂ Term Term
                 | If₃ Term Term
```

Листинг 19: Тип контекста для терма

2.2. Дифференцирование типов данных

Получение типа контекста, как показывает К. Макбрайд в статье [5], — чисто механический процесс, который можно выполнять, применяя к типу структуры данных, для которой требуется построить зиппер, набор математических преобразований, приведённый ниже. В записи данных формул 1 и 0 обозначают рассмотренные в разделе 1 единичный и нулевой типы, а $S + T$ и $S \times T$ соответственно тип-сумму

и тип-произведение (см. листинги 5–7).

$$\begin{aligned}
\partial_x x &\mapsto 1 && (1)\\
\partial_x C &\mapsto 0 && (2)\\
\partial_x (S + T) &\mapsto \partial_x S + \partial_x T && (3)\\
\partial_x (S \times T) &\mapsto S \times \partial_x T + \partial_x F \times G && (4)\\
\partial_x (S|_{y=T}) &\mapsto \partial_x S|_{y=T} + \partial_y S|_{y=T} \times \partial_x T && (5)
\end{aligned}$$

Эти операции, которые можно использовать как список инструкций для автоматического построения типа контекста, похожи внешне на известные правила дифференцирования из математического анализа с тем отличием, что они определяются на типах. Для типа зиппера они означают, что

(1) тип x содержит один x в тривиальном контексте,

(2) константы не содержат x,

(3) в $S + T$ можно найти x либо в S, либо в T,

(4) в $S \times T$ можно найти x в S, пропуская T, либо в T, пропуская S,

(5) подстановка выражения T вместо переменной y в формулу S.

Предыдущий пример вывода типа контекста для терма с использованием приведённых правил записывается следующим образом.

$$\begin{aligned}
\mathbf{term} &= C_{\mathbf{string}} + C_{\mathbf{string}} \times \mathbf{term} + \mathbf{term}^2 + \mathbf{term}^3\\
\mathbf{term}' &= C_{\mathbf{string}} + 2 \times \mathbf{term} + 3 \times \mathbf{term}^2
\end{aligned}$$

Применение этих правил можно запрограммировать на уровне типов. В данной работе поставлена задача их использования с техникой обобщённого программирования для автоматизации получения типа контекста зиппера. Далее рассматривается ранее имеющийся подход к этой проблеме, а найденное решение, основывающееся на новых средствах, приводится в разделе 4.

2.3. Библиотека multirec и обобщённые зипперы

В библиотеке multirec реализован подход к обобщённому программированию типов данных, описанный в [6]. Проблема доступа к рекурсивным элементам структуры данных, включающая задачи свёртки и обхода по значениям типа, имеющего рекурсивную структуру, и в том числе навигации с помощью зиппера, решается для *регулярных типов*[1], подмножества алгебраических типов данных, которые имеют представление в виде наименьшей неподвижной точки некоторого полиномиального выражения над типами.

Тип `Term`, как показывает следующая запись, является регулярным типом.

$$\begin{aligned} \mathbf{term} &= C_{\mathbf{string}} + C_{\mathbf{string}} \times \mathbf{term} + \mathbf{term}^2 + \mathbf{term}^3 \\ &= \mu x.\, C_{\mathbf{string}} + C_{\mathbf{string}} \times x + x^2 + x^3 \end{aligned}$$

Выражения такого вида могут быть представлены как функторы, параметризованные типом `x`, и далее они называются *полиномиальными функторами*.

В [6] пользуются определением регулярного типа, чтобы ввести класс для обобщённого представления всех регулярных типов. Пусть для регулярного типа a существует такой полиномиальный функтор PF_a, что

$$a \cong \mu y.\, PF_a.$$

Тогда, используя свойство наименьшей неподвижной точки $\mu y.\, F = F\,(\mu y.\, F)$, получим изоморфизм типов

$$a \cong \mu y.\, PF_a \cong PF_a\,(\mu y.\, PF_a) \cong PF_a\,(a).$$

[1] В статье выполняются построения для более сложного класса типов — *взаимно рекурсивных*, однако в этой работе рассматриваются только регулярные типы как их частный случай.

Класс регулярных типов, таким образом, определяется, как показано в листинге 20.

```
class Regular a where
  type PF a :: * -> *
  from      :: a -> PF a a
  to        :: PF a a -> a
```

Листинг 20: Класс регулярных типов

Для обобщённого представления полиномиального функтора используются показанные в разделе 1 типы-комбинаторы `K a`, `U`, `I`, `f :+: g` и `f :×: g` (см. листинг 13).

Функции на уровне типов, вычисляющие тип контекста как производную заданного типа, в данном подходе описываются нижеприведённым образом (листинг 21).

```
data family Context (f :: * -> *) a

data instance Context (K b)     a
data instance Context U         a
data instance Context I         a = CI
data instance Context (f :+: g) a = CL (Context f a)
                                  | CR (Context g a)
data instance Context (f :×: g) a = C1 (Context f a) (g a)
                                  | C2 (f a) (Context g a)
```

Листинг 21: Вычисление типа контекста

В библиотеке также введён класс `Zipper`, предоставляющий ряд функций для навигации по структуре, а определение типа фокуса (*location*) выглядит, как в листинге 22.

```
data Loc a where
  Loc :: (Regular a, Zipper (PF a) a)
         => a -> [Context (PF a) a] -> Loc a
```

Листинг 22: Тип фокуса

Недостаток реализации multirec заключается в необходимости вручную переводить тип в регулярное представление или обращаясь к метапрограммному средству Template Haskell, предоставляющему на этапе компиляции возможности изменения синтаксического дерева программы. Представленный в следующем разделе подход [3] поддерживает более удобные и выразительные средства обобщённого программирования и позволяет генерировать код обобщённого представления типа автоматически.

3. Обобщённое программирование с generics-sop

Реализация библиотеки generics-sop основывается на результатах исследования [3] и использует новые расширения языка Haskell, появившиеся в последние годы, что позволяет писать данными средствами более удобный и выразительный обобщённый код.

Данная библиотека предлагает новый стиль обобщённого представления типов данных в виде списочных структур, отличающийся от представления в большинстве используемых в данное время библиотек обобщённого программирования типов данных, подобного рассмотренному в предыдущих разделах, — как комбинации бинарных сумм и произведений с использованием множества частных структур — комбинаторов типов. В новом подходе каждый тип представляется как единая N-арная сумма, где каждый компонент суммы — единое N-арное произведение.

Важным преимуществом нового представления является отделение метаданных от основного структурного представления типа, что делает конструкцию обобщённой структуры менее громоздкой, не содержащей лишней информации и более удобной в использовании, позволяющей писать чистый выразительный код.

3.1. *N*-арные суммы и произведения

Результатом исследования [3] является использование таких современных расширений системы типов Haskell, как DataKinds, ConstraintKinds, PolyKinds и другие, для формирования и обработки списков типов, кодирующих конечные суммы и произведения.

Расширение DataKinds, впервые появившееся в статье [14], позволяет продвигать типы на более высокий уровень — видов типов, и таким образом даёт возможность определять новые виды, соответствующие типу данных. При этом конструкторы значений становятся конструкторами новых типов. Однако такие продвинутые (*promoted*) типы являются ненаселёнными, то есть не имеют конструкторов значений, и для того, чтобы строить термы с использованием этих типов, необходимо вводить структуры данных, предоставляющие для них конструкторы.

В листингах 23–24 приводятся определения типов N-арного произведения и N-арной суммы. Эти типы параметризованы типом полиморфного вида `[k]` (полиморфизм на уровне видов типов включается расширением PolyKinds), который получен продвижением типа списка.

```
data NP (f :: k -> *) (xs :: [k]) where
  Nil  :: NP f '[]
  (:*) :: f x -> NP f xs -> NP f (x ': xs)
```

Листинг 23: Тип N-арного произведения

```
data NS (f :: k -> *) (xs :: [k]) where
  Z ::    f x  -> NS f (x ': xs)
  S :: NS f xs -> NS f (x ': xs)
```

Листинг 24: Тип N-арной суммы

Конструкторы `S`, `Z`, `:*` и `Nil` создают значения этих типов, при добавлении элемента в сумму или произведение его тип заносится

в список типов (здесь `'[]` и `':` — продвинутые версии конструкторов значений списка — для отличения их от обычных конструкторов списка в расширении DataKinds к ним добавляется знак `'`). Введённые структуры позволяют конструировать на уровне термов гетерогенные списки — списки, состоящие из значений разных типов.

В листинге 25 показан пример гетерогенного списка, построенного с помощью конструкторов типа `NP`. N-арная сумма (листинг 26) — это выбор из списка.

```
hlist :: NP I '[String, Int, Char]
hlist = I "ab" :* I 3 :* I 'x' :* Nil
```

Листинг 25: Пример гетерогенного списка — N-арного произведения

```
type HChoice = NS I '[Char, Bool, Int, Bool]

c0, c2 :: HChoice
c0 = Z (I 'a')
c2 = S (S (Z (I 13)))
```

Листинг 26: N-арная сумма — выбор из списка

3.2. Класс Generic и автоматическая генерация обобщённого представления

Используя типы `NS` и `NP`, можно выразить структуру алгебраического типа данных в виде N-арной суммы произведений. К примеру, обобщённое представление типа `Term` из раздела 2 (см. листинг 16) выглядит, как в листинге 27.

N-арные произведения (внутренние списки типов) представляют наборы аргументов конструкторов, а выбор из суммы соответствует выбору конкретного конструктора.

```
type RepTerm = NS (NP I)
  '[ '[String]
   , '[String, Term]
   , '[Term, Term]
   , '[Term, Term, Term]
   ]
```

Листинг 27: Пример обобщённого представления типа

```
type Rep a = NS (NP I) (Code a)

class Generic (a :: *) where
  type Code a :: [[*]]
  from :: a -> Rep a
  to   :: Rep a -> a
```

Листинг 28: Класс обобщённо представимых типов

Класс `Generic` для всех типов, имеющих обобщённое представление такого вида, можно определить, как в листинге 28.

Функции `from` и `to` задают изоморфизм типов `a` и `Rep a` при условии, что их композиция является тождественным отображением, а `Code a` — список списков типов, кодирующий представление в виде N-арной суммы произведений. Такой код позволяет устанавливать соответствие между типом данных и его обобщённым представлением, тем не менее это всё ещё необходимо делать вручную.

Реализация класса `Generic` в библиотеке generics-sop более сложная — она автоматизирует процесс построения представления типа, используя собственный класс `Generic` компилятора GHC и переводя старое представление на основе комбинаторов в новое — в виде N-арной суммы произведений. Код полного определения класса `Generic` из библиотеки generics-sop представлен в листинге 29.

Функции `gfrom` и `gto` определены в библиотеке generics-sop — они реализуют технику, известную как *generic generic programming* [15]. Это техника перевода одних форм обобщённого представления ти-

```
newtype SOP (f :: (k -> *)) (xss :: [[k]])
  = SOP (NS (NP f) xss)

type Rep a = SOP I (Code a)

class (All SListI (Code a)) => Generic (a :: *) where
  type Code a :: [[*]]
  type Code a = GCode a

  from         :: a -> Rep a
  default from :: (GFrom a, GHC.Generic a,
                   Rep a ~ SOP I (GCode a))
                   => a -> Rep a
  from = gfrom

  to         :: Rep a -> a
  default to :: (GTo a, GHC.Generic a,
                 Rep a ~ SOP I (GCode a))
                 => Rep a -> a
  to = gto
```

Листинг 29: Полное определение класса `Generic`

пов данных в другие, что обеспечивает максимальную гибкость при использовании различных средств обобщённого программирования, применение которых можно комбинировать в одной программе, и это также является одним из преимуществ нового подхода.

Ниже приводится пример определения типа данных (листинг 30) и автоматически генерируемого для него кода обобщённого представления. Строка **`deriving`** `GHC.Generic` средствами компилятора создаёт для типа экземпляр класса `GHC.Generic`, а строка **`instance`** `Generic (BinTree a)` преобразуется в следующий код (листинг 31).

```
data BinTree a = Leaf a | Node (BinTree a) (BinTree a)
  deriving GHC.Generic

instance Generic (BinTree a)
```

Листинг 30: Определение АТД бинарного дерева

```
instance Generic (BinTree a) where
  type Code (BinTree a) = '[ '[a]
                           , '[BinTree a, BinTree a]
                           ]

  from :: BinTree a -> Rep (BinTree a)
  from (Leaf x)   = SOP (Z (I x :* Nil))
  from (Node l r) = SOP (S (Z (I l :* I r :* Nil)))

  to :: Rep (BinTree a) -> BinTree a
  to (SOP (Z (I x :* Nil)))            = Leaf x
  to (SOP (S (Z (I l :* I r :* Nil)))) = Node l r
```

Листинг 31: Автоматически генерируемый код обобщённого представления

4. Реализация обобщённого зиппера

В текущем разделе описывается разработанный механизм автоматизированной генерации обобщённого представления контекста зиппера, основанный на структурном представлении типа в виде N-арной суммы произведений, с использованием правил дифференцирования типов данных, рассмотренных в разделе 2.

Данное решение демонстрирует выразительность современных средств языка Haskell и нового подхода к обобщённому программированию типов данных и программированию на уровне типов применительно к задаче построения обобщённого зиппера.

4.1. Введение алгебраических операций над типами

Для дальнейшей работы с представлениями типов в форме вложенных списочных структур введём следующие функции на уровне типов, которые являются алгебраическими операциями, если введённые конструкции на типах мыслить как суммы и произведения, фактически же они соответствуют продвинутым версиям функций работы со списками:

- *сложение N-арных сумм произведений* — это конкатенация списков из списков типов;
- *умножение типа на N-арную сумму произведений* — добавление типа в начало каждого внутреннего списка суммы;
- *умножение N-арного произведения на сумму произведений* — конкатенация списка типов с каждым внутренним списком суммы.

Все функции определяются рекурсивно на списках уровня типов, их реализация приведена в листингах 32–34.

```
type family (.++) (xs :: [[*]]) (ys :: [[*]]) :: [[*]]

type instance (x ': xs) .++ ys = x ': (xs .++ ys)
type instance '[]       .++ ys = ys
```

Листинг 32: Сложение N-арных сумм произведений

```
type family (.*) (x :: *) (ys :: [[*]]) :: [[*]]

type instance x .* (ys ': yss) = (x ': ys) ': (x .* yss)
type instance x .* '[]         = '[]
```

Листинг 33: Умножение типа на N-арную сумму произведений

```
type family (.**) (xs :: [*]) (ys :: [[*]]) :: [[*]]

type instance (x ': xs) .** yss = x .* (xs .** yss)
type instance '[]       .** yss = yss
```

Листинг 34: Умножение N-арного произведения на сумму произведений

Например, вызов `'[Int, Bool] .** '[ '[Bool], '[Bool, Char]]` вычисляет тип `'[ '[Int, Bool, Bool], '[Int, Bool, Bool, Char]]`.

Введение приоритета операций (листинг 35) позволяет далее не ставить скобки в выражениях с их участием.

```
infixr 6 .++
infixr 7 .*
infixr 7 .**
```

Листинг 35: Введение приоритета операций

4.2. Получение типа контекста

Следующим этапом для построения обобщённого представления типа контекста является определение функции дифференцирования N-арного произведения типов (листинг 36).

```
type family DiffProd (a :: *) (xs :: [*]) :: [[*]] where
  DiffProd a '[] = '[]
  DiffProd a '[a] = '[ '[]]
  DiffProd a '[x] = '[]
  DiffProd a (x ': xs)
    = xs .** DiffProd a '[x] .++ x .* DiffProd a xs
```

Листинг 36: Дифференцирование N-арного произведения

Функция, вычисляющая тип контекста, реализованная с использованием всех введённых ранее операций, представлена в листинге 37.

```
type family ToContext (a :: *) (code :: [[*]]) :: [[*]]

type instance ToContext a (xs ': xss)
  = DiffProd a xs .++ ToContext a xss
type instance ToContext a '[] = '[]
```

Листинг 37: Генерация типа контекста

К примеру, для типа `Tree` (листинг 38) и вызова

```
type RepTreeContext = NS (NP I) (ToContext Tree (Code Tree))
```

автоматически генерируется следующий код (листинг 39).

```
data Tree = Leaf  Int
          | TNode Tree Tree Int Bool Tree
          | BNode Bool Tree Tree
  deriving GHC.Generic

instance Generic Tree
```

Листинг 38: АТД: дерево

```
type RepTreeContext = NS (NP I)
  ('[ '[Tree, Int, Bool, Tree]
    , '[Tree, Int, Bool, Tree]
    , '[Tree, Tree, Int, Bool]
    , '[Bool, Tree]
    , '[Bool, Tree] ])
```

Листинг 39: Автоматически генерируемый код типа контекста

Заключение

Результатом данной работы является полученная техника построения типа контекста обобщённого зиппера, которая служит примером возможности применения средств обобщённого программирования к различным задачам эффективного обхода структур данных

и их обработки. Выразительность приведённого кода демонстрирует преимущества применения нового подхода к обобщённому программированию типов данных к поставленной задаче.

В ходе исследования изучены имеющиеся подходы к обобщённому программированию типов данных, проведён их сравнительный анализ, выявлены достоинства и недостатки различных библиотек обобщённого программирования.

Список литературы

1. *Löh A.* Exploring Generic Haskell // PhD thesis, Utrecht University. — 2004.
2. *Gibbons J.* Datatype-Generic Programming // Spring School on Datatype-Generic Programming. — 2007.
3. *Vries E. de*, *Löh A.* True Sums of Products // The ACM SIGPLAN Workshop on Generic Programming. — 2014.
4. generics-sop, библиотека обобщённого программирования, основанная на суммах произведений. — URL: `http://hackage.haskell.org/package/generics-sop`.
5. *McBride C.* The Derivative of a Regular Type is its Type of One-Hole Contexts. — 2001. — URL: `http://strictlypositive.org/diff.pdf`.
6. Generic Programming with fixed points for mutually recursive datatypes / A. Rodriguez [и др.] // The ACM SIGPLAN International Conference on Functional Programming. — 2009.
7. multirec, библиотека обобщённого программирования, основанная на неподвижных точках. — URL: `http://hackage.haskell.org/package/multirec`.

8. Модуль библиотеки multirec, содержащий метапрограммный код для генерации регулярного представления. — URL: http://hackage.haskell.org/package/multirec-0.7.7/docs/Generics-MultiRec-TH.html.
9. *Löh A.* Applying Type-Level and Generic Programming in Haskell // Summer School on Generic and Effectful Programming. — 2015.
10. GHC generics, ранний подход к обобщённому программированию типов данных. — URL: http://hackage.haskell.org/package/base-4.7.0.2/docs/GHC-Generics.html.
11. *Huet G.* The Zipper // Journal of Functional Programming. — 1997.
12. *Adams M. D.* Scrap Your Zippers: A Generic Zipper for Heterogeneous Types // The ACM SIGPLAN Workshop on Generic Programming. — 2010.
13. xmonad, оконный менеджер, использующий зиппер. — URL: http://xmonad.org.
14. Giving Haskell a Promotion / S. Weirich [и др.] // The ACM SIGPLAN Workshop on Types in Language Design and Implementation. — 2012.
15. *Magalhães J. P.*, *Löh A.* Generic Generic Programming // International Symposium on Practical Aspects of Declarative Languages. — 2014.