ПРОГРАММИРОВАНИЕ, 2015, № 4, с. 13–22

УДК 681.3.06

АССОЦИИРОВАННЫЕ ТИПЫ И РАСПРОСТРАНЕНИЕ ОГРАНИЧЕНИЙ НА ПАРАМЕТРЫ-ТИПЫ ДЛЯ ОБОБЩЕННОГО ПРОГРАММИРОВАНИЯ НА SCALA

© 2015 г. А.М. Пеленицын

Южный федеральный университет

344006 Ростов-на-Дону, ул. Б. Садовая, 105/42

E-mail: apel@sfedu.ru

Поступила в редакцию 05.03.2015

Обобщённое программирование представляет собой принцип создания программных компонент, характеризующихся высокой степенью повторной используемости за счёт отделения реализации алгоритмов от конкретных определений типов данных, с которыми работают эти алгоритмы. В последние годы проведён ряд исследований, направленных на выявление имеющихся и разработку новых механизмов поддержки обобщённого программирования в различных языках программирования. В данной работе анализируются и развиваются подходы к реализации обобщённого программирования с использованием различных элементов языка программирования Scala, разработанного в Федеральной политехнической школе Лозанны специалистом в теории и реализации языков программирования Мартином Одерски.

1. ВВЕДЕНИЕ

Обобщённое программирование представляет собой принцип создания программных компонент, характеризующихся высокой степенью повторной используемости за счёт отделения реализации алгоритмов от конкретных определений типов данных, с которыми работают эти алгоритмы [1].

Различные языки программирования предоставляют разные средства поддержки обобщённого программирования. Можно выделить три подхода к выражению идей обобщённого программирования с помощью современных языков программирования. Первый состоит в использовании имеющихся в языках стандартных средств. Примером этого подхода является применение шаблонов C++, а также обобщённых типов в распространённых объектно-ориентированных языках (Java, C#). Второй подход состоит в проектировании соответствующих расширений для известных языков программирования (например, расширение C# в [2], JavaGI [3]). В рамках третьего подхода разрабатываются новые языки, изначально содержащие нужные механизмы. Главным примером здесь является язык программирования Scala, хотя можно упомянуть и другие, направленные в основном на академические исследования языки построения доказательств, такие как Agda и Coq.

Отдельным примером языка, имеющего развитые средства обобщённого программирования, является язык Haskell. Он изначально проектировался как язык с одним главным компилятором, открытым для экспериментов в области теории типов. Часть результатов такого рода постепенно включается в новые редакции стандарта языка. Однако даже базовые средства Haskell, классы типов и повсеместно используемый параметрический полиморфизм представляют собой довольно развитую инфраструктуру для применения обобщённого программирования. Таким образом, в отношении предложенной выше классификации этот язык равноправно поддерживает первый и второй подходы к реализации и развитию средств обобщённого программирования. Ряд нестандартных, но широко использу-

емых расширений компилятора Glasgow Haskell Compiler (GHC), такие как мультипараметрические классы типов и функциональные зависимости, укрепляют позиции языка в этом отношении.

Исторически первым примером языка, в котором достаточно широко были применены принципы обобщённого программирования, стал язык C++. Так называемые неограниченные шаблоны C++ дают достаточно широкую свободу в использовании параметров-типов (один из главных инструментов обобщённого программирования). Однако использование этих шаблонов имеет ряд известных издержек, в первую очередь это сложность поиска и исправления ошибок (в том числе трудночитаемые сообщения компиляторов об ошибках), а также недоступность раздельной трансляции обобщённых программных модулей (шаблонов классов и функций). Повысить удобство использования обобщённого подхода был призван, в том числе, новый стандарт языка C++11. Однако отказ от включения в него концептов (concepts) по ряду причин сильно ослабил позиции нового стандарта в этом отношении. Тем не менее сегодня, когда речь идёт об испытании тех или иных средств, призванных поддерживать обобщённое программирование, сравнение зачастую проводится либо с известными примерами из области C++, либо с языком Haskell.

Так, статья [2] предлагает расширение языка C# и демонстрирует, как на этом пути снимаются недостатки, заметные при попытке воспроизвести фрагмент известной обобщённой C++-библиотеки Boost Graph Library (BGL) на обычном C#. Само расширение состоит в добавлении так называемых ассоциированных типов и механизма распространения ограничений для параметров-типов.

Работа [4] посвящена описанию одного из самых незаурядных элементов языка Scala — имплиситам. В ней приводится решение ряда задач разной степени сложности, начиная от моделирования классов типов Haskell, что автоматически вводит Scala в ряд языков, развивающих обобщённый подход, заканчивая довольно сложными примерами вычислений на уровне типов (type-level computations), такими как вычисление типа функции:

```
zipWithN :: (a1 → a2 → … → an) →
            ([a1] → [a2] → … → [an])
```

где n является параметром (на входе — функция от $n-1$ аргумента, а также $n-1$ списков, на выходе — список результатов применений данной функции к соответствующим элементам данных списков). Похожая техника для представления многочленов многих переменных на C++ рассмотрена в [5].

Детальное рассмотрение одного элемента языка, имплиситов, не мешает сделать [4] ряд важных выводов в отношении поддержки обобщённого программирования со стороны Scala в целом. В частности, в статье приведено расширение карты поддержки обобщённого программирования современными языками, первая версия которой была опубликована в одной из основополагающих работ [6]. В последней работе сравнение обобщённых возможностей разных языков программирования, не включая Scala, проводилось на примере реализации упомянутой выше библиотеки BGL.

В [2] проводится анализ возможной реализации фрагмента BGL с использованием расширенного C#. В настоящей работе предложен аналогичный анализ для языка Scala, который представляется более удобным инструментом обобщённого программирования. Выделяются элементы языка Scala, которые позволяют более лаконично реализовать идеи ассоциированных типов и распространения ограничений для решения типичных проблем обобщённого программирования на примере, рассмотренном в [2].

Первым основным результатом работы является применение уточнённых типов Scala (refinement types) для описания ограничений на ассоциированные типы. Далее, показано, как с помощью типов методов, зависящих от цепочек, подход [2] освобождается от необходимости использования параметрического полиморфизма, что в конкретном примере из [2] делает решение более кратким и ясным.

Второй основной результат работы состоит в выявлении и устранении недостатка подхода [2], связанного с отсутствием возможности ретроактивного моделирования, которая представляет собой один из критериев поддержки обобщённого программирования, выделенных в [6]. Общий

подход к решению этой проблемы на основе имплиситов Scala предложен в [4], мы показываем, как адаптировать его в типичной ситуации промышленного библиотечного кода из [2].

Полученные в настоящей работе результаты дополняют описание методов обобщённого программирования на Scala, которое дано в [4]. При этом используется материал, который, с одной стороны более доступен для понимая сути обобщённого программирования, а с другой стороны, более близок к индустриальному программированию, чем в [4], а именно фрагмент BGL. Более реалистичный пример заставляет привлекать элементы языка Scala, которые ранее не использовались (уточнённые типы) или использовались в малой степени (абстрактные типы-члены и типы, зависящие от цепочек) в такого рода задачах. Полученная в настоящей работе конструкция сопоставляется с результатами [2], а также с некоторыми элементами обобщённого программирования в языках C++ и Haskell.

В работе [7] в заключительных замечаниях (п. 6, Связанные работы) отмечаются перспективы Scala в отношении обобщённого программирования. Авторы [7] упоминают о своих планах провести исследование по использованию ассоциированных типов и имплиситов для прояснения этих перспектив, аналогичное тому, как это сделано в настоящей работе. Однако в доступной литературе такие результаты не найдены.

Статья состоит из введения, трёх разделов и заключения. Во втором разделе излагаются результаты [2] в удобном для нас виде: описывается фрагмент BGL и приводится его реализация на C#, в том числе с расширениями из работы [2]. Кроме того, здесь обсуждаются различные недостатки имеющихся решений. В третьем разделе демонстрируются возможности Scala в области обобщённого программирования на примере реализации того же фрагмента BGL, при этом отмечаются преимущества Scala перед расширенным C# из [2]. В разделе 4 отмечается несоответствие подхода [2] некоторым требованиям обобщённого программирования и конструируется решение этой проблемы с использованием идей [4]. Здесь же проводится краткий анализ дополнительных затрат как клиента, так и автора библиотеки, которая использует такой подход, а также делаются замечания относительно других языков программирования в этом отношении. Два основных результата работы, упомянутых выше, изложены, соответственно, во третьем и четвертом разделах статьи.

2. ФРАГМЕНТ BGL И ЕГО РЕАЛИЗАЦИЯ НА C# С РАСШИРЕНИЯМИ

В качестве примера для иллюстрации обобщённых возможностей предлагаемого расширения C# в [2] использована следующая функция из состава BGL (C++):

```
template ⟨class Graph⟩
typename Graph::vertex_type
first_neighbor(Graph g, typename
Graph::vertex_type v)
{return target(current(out_edges(v,
g)));}
```

Приведённая функция по заданному экземпляру графа и некоторой вершине, принадлежащей этому графу, возвращает одну из вершин, смежных данной (“первую” в некотором упорядочении). Один очевидный технический недостаток этого кода связан с необходимостью использовать ключевое слово `typename` для указания того, что зависимый от параметра шаблона Graph идентификатор `vertex_type` является именем типа, а не обычным членом класса. Другой, более фундаментальный недостаток “неограниченных” шаблонов языка C++: заголовок и тело данной функции содержат довольно большое количество предположений о параметре-типе `Graph`, которые никак не отражены в коде, а лишь описаны в документации (эти предположения будут явно описаны далее). Пользователь такой функции должен искать и разбираться с документацией к каждой из таких функции либо пробовать запускать компиляцию кода, использующего эти функции, а также готовиться к разбору некоторого количества малопонятных ошибок компилятора, по которым можно восстановить те же ограничения. Однако компиляция программ на C++ занимает довольно большое время, которое превосходит время компиляции программ на более современных языках. Сообщения об ошибках при использовании шаблонных функций разбирать особенно трудно, и несмотря на то, что к решению этой проблемы было привлечено немало сил разработчиков компиляторов, результаты их работы здесь всё ещё сложно признать достигшими своей цели.

```
G_Vertex first_neighbor⟨G, G_Vertex, G_Edge, G_OutEdgeIterator⟩
              (G g, G_Vertex v)
         where G : IncidenceGraph⟨G_Vertex, G_Edge, G_OutEdgeIterator⟩,
               G_Edge : GraphEdge⟨G_Vertex⟩,
               G_OutEdgeIterator : IEnumerable⟨G_Edge⟩{
return g.out_edges(v).Current.target();
}
```

Рис. 1.

```
interface IncidenceGraph⟨Vertex, Edge, OutEdgeIterator⟩
        where Edge : GraphEdge⟨Vertex⟩,
              OutEdgeIterator : IEnumerable⟨Edge⟩{
  OutEdgeIterator out_edges(Vertex v);
  int out_degree(Vertex v);
}
```

Рис. 2.

Путь, по которому последнее десятилетие шло сообщество C++ для решения указанной проблемы, — это разработка достаточно формализованной документации к обобщённым библиотекам. Центральным понятием на этом пути стало понятие концепта (concept) — совокупности ограничений, которым должны удовлетворять параметры-типы в обобщённых функциях и классах. Путь этот достаточно ограничен, и потому было принято несколько попыток явного введения концептов в язык C++ (достаточно полный обзор работ в этом направлении с сопутствующими ссылками дан в [8]). В итоге концепты не были добавлены в новый стандарт языка C++11, что можно рассматривать как неудачу в процессе усиления поддержки обобщённого программирования на C++ [11].

Альтернативный к неограниченным шаблонам C++ механизм выражения идей обобщённого программирования можно найти в современных объектно-ориентированных языках программирования, таких как Java и C#. Отличие подходов C++ и Java/C# хорошо изучено в рамках теории типов: C++ поддерживает параметрический ("неограниченный") полиморфизм, а Java/C# — ограниченный полиморфизм, формализованный в рамках Системы $F_{<:}$ — объединения классической системы F с подтипированием [9]. Отличие этих двух подходов в самом простейшем случае, который, однако, хорошо описывает положение дел в C++ и Java/C#, можно понять, используя аналогию с двумя правилами: "разрешено всё, что не запрещено" и "запрещено всё, что не разрешено", соответственно. Во втором случае возникает необходимость писать большое количество явных параметров-типов и описывать связи между ними. Выясняется, что базовые средства Java/C# плохо справляются с такой задачей. На рис. 1 показано, как будет выглядеть описанная выше функция на обычном (без расширений) языке C#.

В определении функции `first_neighbor` содержится довольно много предположений о свойствах типов параметров. В коде на C++, как было отмечено выше, они не отражены явно, в отличие от данного фрагмента на языке C#. Второе отличие первого фрагмента кода от второго чисто стилистическое: вызовы свободных функций заменены на вызовы методов.

Приведём пример предположения о типе параметра `g`: этот тип должен позволять вызывать метод `out_edges`. В чистых объектно-ориентированных языках традиционным способом фиксации этого требования является требование реализации некоторого специального интерфейса. В данном случае он назван `IncidenceGraph`. Метод возвращает итератор по коллекции инцидентных данной вершине дуг, этот факт отражён аналогично: вводится дополнительный параметр-тип `G_OutEdgeIterator`, который должен реализовывать стандартный интерфейс `IEnumerable`. В этом интерфейсе имеется свойство `Current`, которое возвращает ссылку на текущую ("первую") вершину, что используется в теле функции, и так далее.

Первая проблема, возникающая в таком коде, — это большое число явных параметров-типов. Вторая проблема станет видна, если рас-

```
interface GraphEdge {
   type Vertex;              // (1)
   Vertex source();
   Vertex target();
}
interface IncidenceGraph {
   type Vertex;              //  (1)
   type Edge : GraphEdge;    //  (2)
   type OutEdgeIterator : IEnumerable<Edge>; // (2)
   require Vertex == Edge::Vertex;           // (3)
   OutEdgeIterator out_edges(Vertex v);
   int out_degree(Vertex v);

}
G::vertex_type first_neighbor<G>(G g, G::Vertex v) where G : IncidenceGraph {
   return g.out_edges(v).Current.target();
}
```

Рис. 3.

смотреть определение одного из использованных в предыдущем коде интерфейсов. На самом деле, это обобщённые интерфейсы, которые зависят от параметров-типов, и для их корректного описания требуется упоминать практически те же самые ограничения, что в определении функции выше (см. рис. 2).

Разумеется, хотелось бы описывать ограничения для параметров `Edge` и `OutEdgeIterator` только однажды, в определении типа `IncidenceGraph`, и избавить программиста от необходимости повторять их при каждом определении новой функции наподобие `first_neighbour`.

Обе указанные проблемы решаются в [2] с помощью проектирования расширения языка C#: введения ассоциированных типов и автоматического распространения ограничений, соответственно. Механизм ассоциированных типов позволяет объявлять внутри интерфейсов абстрактные типы по аналогии с традиционными абстрактными методами. Такие абстрактные типы-члены, как и методы интерфейсов, должны быть конкретизированы в классе, реализующем интерфейс. В этом отношении название абстрактные типы-члены, принятое в Scala, является более строгим, чем "ассоциированные типы", используемые в [2].

При добавлении ассоциированных типов в интерфейсы возникает необходимость в описании ограничений на них по аналогии с тем, как это делается для параметров-типов. Традиционно первый вид ограничений такого рода в объектных языках — это ограничения вида: тип T должен быть подтипом некоторого другого типа. Оказывается удобным добавить ещё один простейший вид ограничений — равенство параметров-типов или ассоциированных типов. Например, параметр-тип (или ассоциированный тип) для вершины графа `Vertex`, используемый в интерфейсах дуги графа `GraphEdge` и самого графа `IncidenceGraph`, должны совпадать, если они используются при определении одной функции.

Распространение ограничений — это механизм, позволяющий компилятору использовать ограничения на типы, определённые в разных интерфейсах, для доказательства того, что код, использующий эти интерфейсы, удовлетворяет системе типов.

В результате реализации этих идей в статье [2] получен следующий код для исходной функции на расширенном C# (см. рис. 3).

Метками (1) обозначены места, где ассоциированные типы использованы вместо типов-параметров интерфейсов, (2) — ограничения вида "T является подтипом S", (3) — ограничения вида "T совпадает с S".

3. ВЛОЖЕННЫЕ ОБЪЯВЛЕНИЯ ТИПОВ И ОБОБЩЕННЫЕ ОГРАНИЧЕНИЯ В SCALA

В языке Scala изначально присутствовали вложенные объявления типов, так называемые абстрактные типы-члены; имеется также несколько различных способов описания ограничения на

```
trait GraphEdge {
  type Vertex         // (1)
  def source: Vertex
  def target: Vertex
}
trait IncidenceGraph {
  type Vertex         // (1)
  type Edge <: GraphEdge { type Vertex = IncidenceGraph.this.Vertex} // (3)
  type OutEdgeIterator <: Iterator[Edge]                             // (2)

  def out_edges(v: Vertex): OutEdgeIterator
  def out_degree(v: Vertex): Int
}
def first_neighbor(g: IncidenceGraph)(v: g.Vertex): g.Vertex = {
  g.out_edges(v).next.target
}
```

Рис. 4.

эти типы и типы-параметры интерфейсов и методов. Сами интерфейсы в Scala несколько видоизменены: используется более общее понятие характеристики (trait). Характеристики можно использовать как обычные интерфейсы, знакомые по языкам Java и C#. Однако характеристики могут содержать и реализацию методов. Отметим, что в данной статье это свойство не используется, однако способ, которым в Scala обходятся известные проблемы множественного наследования реализации, можно прочесть в любой книге по Scala, к примеру в [10].

Компилятор Scala обладает мощным механизмом анализа типов, и потому распространение ограничений, которое авторы [2] рассматривали отдельно, здесь не требует дополнительного обсуждения: если ограничения описаны в характеристике, они будут учтены при использовании объектов, реализующих данную характеристику. В результате может быть получено решение, ничем не уступающее предложенному в [2] (см. рис. 4).

Типы-члены класса или характеристики в Scala могут быть конкретными:

```
type Vertex = Int
```

— в этом случае они играют роль простых псевдонимов (подобно typedef в C++); другая возможность — это абстрактные типы-члены, примеры которых обозначены (1) в предыдущем листинге; они могут быть конкретизированы в наследниках характеристики.

Абстрактный тип может быть ограничен (см. (2)). Например, объявление

```
type A <: B // (4)
```

гарантирует, что для конкретизации типа `A` в наследниках обязательно будет использован один из подтипов типа `B`.

Наиболее явным свидетельством развитости системы типов Scala является тот факт, что ограничения на типы-члены могут быть выражены без введения специального синтаксиса, как это было с ключевым словом `require` в [2]. Рассмотрим подробнее (3). Всё это выражение целиком удовлетворяет виду (4), где в качестве `B` использован так называемый уточнённый тип (refinement type):

`GraphEdge { type Vertex = IncidenceGraph.this.Vertex }` // (5)

Этот тип является надтипом любого типа, который наследует `GraphEdge` и определяет свой тип-член `Vertex` так же, как определён `Vertex` в текущем объекте (`this`), реализующем характеристику `IncidenceGraph`.

Кажется, что конструкция `IncidenceGraph.this.Vertex` переусложнена по сравнению с обращением к ассоциированным типам, которое использовалось в [2]. В Scala имеются так называемые проекции типов, которые выглядят более привычно: `IncidenceGraph#Vertex`. Данная запись подразумевает все типы-члены `Vertex`, определённые внутри любого типа, наследующего IncidenceGraph. Легко понять, что такая широкая трактовка не подходит для данной задачи: если в одном типе графа вершина определяется целым числом, а в другом — строкой,

то, к примеру, метод `first_neighbor`, получив на вход первый граф, должен возвращать целочисленное, но никак не строковое значение.

Возможность описывать ограничения на ассоциированные типы указана в [6] среди основных критериев поддержки обобщённого программирования в языках программирования. Насколько нам известно, использование уточнённых типов с этой целью в Scala не предлагалось ранее.

Итак, запись (5) описывает в точности группу типов с нужным для `Edge` из (3) свойством. Вернёмся вновь к (3). Изначальный смысл <: состоял в определении отношения подтипирования. Поскольку (5) (правая часть <: в (3)) уже описывает нужную группу типов, то <: логично было бы заменить на = . Однако в этом случае в (3) описывался бы уже не абстрактный, а конкретный тип-член, и тип `Edge` превратился бы в обычный псевдоним для записи длинной конструкции (5). Потому в (3) следует использовать именно <:, а не = .

В заголовке метода `first_neighbor` совсем не потребовалось использовать тип-параметр, как это требовалось в [2]: там идентификатор `G` был необходим, чтобы получить доступ к вложенному типу `G::Vertex`. В Scala, чтобы получить нужный вложенный тип нужного наследника `IncidenceGraph`, следует использовать переменную (параметр g в данном случае), а не `G#Vertex`, по указанной выше причине со слишком широкой трактовкой проекций типов, таких как `G#Vertex`. Если тип параметра, в данном случае параметра `v` — вершины графа, зависит от другого параметра, в данном случае `g` — графа, то их следует помещать в разные списки аргументов, то есть разные пары круглых скобок — таково требование языка, так называемые типы методов, зависящие от цепочек (path-dependent method type).

Понятие цепочки (path) играет большую роль в системе типов Scala. Цепочка — это запись вида A.B.C.D. В такой последовательности разрешено лишь четыре типа идентификаторов в качестве A, B,...: пакет, объект, значение val, this/super и их варианты с указанием параметра-типа. В случае выполнения такого требования на элементы цепочки говорят, что цепочка определяет стабильный тип, то есть конкретный тип, известный на этапе компиляции. Это отличает цепочки от проекций типов наподобие `G#Vertex`, которую можно было бы использовать в заголовке метода `first_neighbor`:

```
def first_neighbor[G <: IncidenceGraph](g:
G, v: G#Vertex):G#Vertex =
      g.out_edges(v).next.target
```

Такой вариант более близок по форме к версии в [2], он позволил не выделять различия между проекциями и цепочками (в терминологии Scala) авторам [2]. Однако, как упоминалось выше, в Scala проекция гарантирует менее строгий контроль типов, чем цепочка. Кроме того, такой вариант метода не скомпилируется, если не изменить объявление `out_edges`: в качестве параметра этого метода следует также указать проекцию. В этом случае получится менее строгий контроль типов за счёт более громоздкой записи, что вряд ли можно как-то оправдать.

В таких широко распространённых языках программирования, как C++, Java, C#, нет механизмов, аналогичных абстрактным типам-членам, потому в них не возникает раздельных понятий, наподобие цепочек и проекций типов — в контексте таких языков обычно используется единое понятие квалифицированных типов.

4. ИМПЛИСИТЫ КАК СРЕДСТВО УКАЗАНИЯ ОГРАНИЧЕНИЙ НА ПАРАМЕТРЫ-ТИПЫ

Решение, приведённое в разделе 3, достаточно близко воспроизводит подход [2], что становится возможным благодаря использованию таких элементов Scala, как абстрактные типы-члены, уточнённые типы и типы методов, зависящие от цепочек. Это решение почти целиком полагается на различные формы подтипирования и не использует параметры-типы, что характерно для объектного, а не обобщённого подхода. Вследствие этого оно не удовлетворяет нескольким важным качествам обобщённого кода. В первую очередь — возможности ретроактивного моделирования. В данном случае это означает неспособность метода `first_neighbor` работать с типами, которые не обладают характеристикой `IncidenceGraph`, то есть, в более привычной терминологии, не реализуют нужный ин-

```
trait IncidenceGraph[G] { graph ⇒
    type Vertex
    type Edge
    val edgeMod: GraphEdge[Edge] {type Vertex = graph.Vertex}
    type OutEdgeIterator <: Iterator[Edge]
    def out_edges(g: G, v: Vertex): OutEdgeIterator
}
```

Рис. 5.

терфейс. Это является прямым следствием использования в большей мере объектного подхода. Данная особенность вызовет проблемы у клиента библиотеки графовых компонент, содержащей `first_neighbor`, если клиент имеет собственный тип графа, который не обладает характеристикой `IncidenceGraph` и который нельзя менять, например, из соображений обратной совместимости. Такая проблема решается в обобщённом подходе, если используемый язык содержит достаточно развитые средства поддержки данного подхода. В [4] показано, как такого рода проблемы можно решать в Scala. Рассмотрим решение данной проблемы в конкретном примере из [2], пользуясь идеями [4].

В обобщённом программировании ограничения, которым должен удовлетворять переданный тип, формулируются без помощи отношения подтипирования, связывающего этот тип. Вместо этого ограничения собираются в отдельную сущность, которую принято называть концептом или классом типа. Первое название больше характерно для сообщества C++, второе — Haskell, мы будем использовать первое. Чтобы указать способ, которым конкретный тип удовлетворяет концепту, нужно определить модель концепта для данного типа. Заметим, что это можно сделать намного позже того, как конкретный тип был определён и зафиксирован. Так выглядит ретроактивное моделирование.

Чтобы понять, каким образом обобщённый подход реализуется в Scala, нужно ответить на ряд вопросов. Прежде всего, как формулируется концепт. Здесь ответ прост: концептом является параметризованная характеристика. Вот, к примеру, концепт для дуги графа.

На такую характеристику можно смотреть как на предикат: тип E удовлетворяет предикату “тип дуги”, если определён тип вершины дуги и два метода — получение вершины начала и прибытия дуги.

```
trait GraphEdge[E] {
  type Vertex
  def source(e: E): Vertex
  def target(e: E): Vertex
}
```

Далее, для конкретного типа и данного концепта можно создать модель. В Scala это можно сделать с помощью так называемых объектов.

```
  type BasicEdge = (Int, Int)

  object basicEdgeMod extends
GraphEdge[BasicEdge] {
      type Vertex = Int
      def source(e: BasicEdge) = e._1
      def target(e: BasicEdge) = e._2
  }
```

Объект `basicEdgeMod` показывает, что тип `BasicEdge` удовлетворяет концепту `GraphEdge`. Объекты в Scala можно представлять как реализацию понятия синглтона (объекта-одиночки) на уровне языка: они описывают тип и сразу же — единственный экземпляр этого типа. Объекты представляются удобным средством описания моделей концептов в Scala. Впервые это наблюдение сделано в [4].

Концепт для всего графа показан на рис. 5. Этот концепт, кроме прочего, требует наличия модели дуги графа для типа `Edge` — в потомках придётся явно определить значение поля `edgeMod`. Идентификатор `graph` здесь является полным аналогом `IncidenceGraph.this`, который использовался в разделе 3 настоящей статьи, предложение `graph =>`, вводящее этот идентификатор, используется, таким образом, исключительно для сокращения записи.

Если имеется конкретный тип графа, `AdjacencyList`, то модель концепта `IncidenceGraph` для него должна, в общем, выглядеть следующим образом (см. рис. 6). Реа-

```
implicit object adjListModel extends IncidenceGraph[AdjacencyList] {
  type Vertex = Int
  type Edge = BasicEdge
  val edgeMod = basicEdgeMod
  type OutEdgeIterator = Iterator[Edge]
  def out_edges(g: AdjacencyList, v: Vertex): OutEdgeIterator = ???
}
```

Рис. 6.

```
def first_neighbor[G, V](g: G, v: V)
   (implicit igMod: IncidenceGraph[G] {type Vertex = V}): igMod.Vertex =
                igMod.edgeMod.target(igMod.out_edges(g, v).next)
```

Рис. 7.

лизация метода `out_edges` здесь не приводится, так как она зависит от способа реализации и интерфейса класса `AdjacencyList`. Слово implicit перед определением объекта означает, что данный объект будет рассматриваться среди кандидатов для передачи в качестве неявного параметра методов, которые такие параметры имеют. Например, `first_neighbor` (см. рис. 7).

Здесь уже от типов графа и вершины не требуется, чтобы они реализовывали некоторые интерфейсы. Вместо этого требуется передача ещё одного параметра, который показывает, что для типа `G` имеется модель концепта Графа, `IncidenceGraph[G]`. Реальный вызов такого метода может выглядеть так:

```
first_neighbor(randomAdjacencyList, 1),
```

где `randomAdjacencyList` возвращает случайный граф. Компилятор сам найдёт определение `adjListModel` и подставит его в качестве неявного параметра `igMod`, если оно помещено в нужной области видимости (подробные правила такого размещения можно найти в литературе, например [10], гл. 21).

Стоит заметить, что код вызова библиотечной функции не изменится по сравнению с тем, как он должен был выглядеть при использовании подхода, рассмотренного в разделе 3. При этом несколько усложнился заголовок функции, и намного менее ясной стала её реализация. То есть основные расходы на дополнительную обобщённость понесёт автор библиотеки, а не её клиент, что можно признать допустимым. По сравнению с вариантом из раздела 3 клиенту придётся определять модели концептов используемой библиотеки для собственных типов. Если клиент использует типы, входящие в эту же библиотеку, то достаточно применять модели, поставляемые с этой библиотекой, и в этом случае клиент вовсе не увидит разницы.

Можно ли снизить дополнительные затраты клиента в части определения моделей для своих типов? В языках, давно поддерживающих обобщённое программирование, есть несколько элементов, служащих этой цели. Например, в Haskell имеется способ так называемого порождения инстанций классов типов (полный аналог моделей концептов) для простейших встроенных классов типов (концептов). В сообществе C++ всё ещё продолжается дискуссия о том, как должны выглядеть концепты и какими свойствами обладать. В том числе обсуждается, насколько развитым должен быть механизм автоматического порождения моделей (concept maps).

Приведём пример, когда полезно использовать автоматическое порождение моделей. Предположим, что имеется такой же, как в разделе 3, тип графа, содержащий все необходимые объявления вложенных типов и метод `out_edges`. Как соединить его с методом `first_neighbor`, определённым в данном разделе? Необходимо создать модель, в которой `out_edges` будет выглядеть следующим образом.

```
def out_edges(g: AdjacencyList, v: Vertex):
  OutEdgeIterator = g.out_edges(v)
```

Такое определение `out_edges` можно было бы легко сгенерировать средствами компилятора. Нужно ли предоставлять такую возможность? Такой вопрос поставлен, в частности, в работе

Б. Страуструпа [11] (раздел Technical Issues), где автор обсуждает возможность включения концептов в стандарт C++ в будущем, после выхода стандарта 2011 года, из которого концепты были удалены. Страуструп полагает, что такая возможность должна быть предоставлена. В Scala такая возможность (на сегодняшний день) недоступна.

5. ЗАКЛЮЧЕНИЕ

В данной работе выделены элементы языка Scala, с помощью которых можно определить необходимые требования к обобщённым интерфейсам фрагмента Boost Graph Library. Реализованы два подхода к выражению таких требований: один подход транслирует идеи [2] в окружение языка Scala и во многом основан на подтипировании, а второй более близок к предложениям, сделанным в [4], и использует явное выделение таких сущностей обобщённого программирования, как концепты и их модели.

Идея использования характеристик Scala для определения концептов может столкнуться с ограничениями, свойственными подтипированию, и последний пример раздела 4 — это только один простейший случай такого рода проблем. В то же время, язык предоставляет такой интересный механизм, как структурные типы, позволяющие описывать список нужных членов в типе и не требующих наследования от какой-либо заранее определённой характеристики. Возможно, структурные типы могли бы стать более гибкой основой обобщённого программирования на Scala, однако этот вопрос требует дополнительного исследования.

Автор статьи благодарен участникам научного семинара по основаниям языков программирования под руководством доцента, к.ф.-м.н. С.С. Михалковича за подробное обсуждение использовавшихся в работе статей, а также, в особенности, участникам семинара Ю.В. Беляковой и В.Н. Брагилевскому за консультации и ценные советы по решению поднятых в данной статье проблем.

СПИСОК ЛИТЕРАТУРЫ

1. *Musser D.A., Stepanov A.A.* Generic Programming // Proceeding of International Symposium on Symbolic and Algebraic Computation. V. 358 of Lecture Notes in Computer Science. Rome. Italy. 1988. P. 13–25.

2. *Jarvi J., Willcock J., Lumsdaine A.* Associated types and constraint propagation for mainstream object-oriented generics // OOPSLA '05 Proc. of the 20th annual ACM SIGPLAN conf. on Object-oriented programming, systems, languages, and applications. ACM New York, NY, USA. 2005. P. 1–19.

3. *Wehr S., Lammel R., Thiemann P.* JavaGI: Generalized Interfaces for Java // Proc. of the European Conf. on Object-Oriented Programming, ed. E. Ernst. LNCS, vol. 4609. Berlin, Germany. Springer-Verlag. 2007. P. 347–372.

4. *Oliveira B.C.d.S., Moors A., Odersky M.* Type classes as objects and implicits // OOPSLA '10 Proc. of the ACM int. conf. on Object oriented programming systems languages and applications. ACM New York, NY, USA. 2010. P. 341–360.

5. *Пеленицын А.М.* Методы обобщённого и метапрограммирования в программной реализации декодера алгебро-геометрических кодов // Прикл. информатика, 2012, № 2 (38). С. 60–70.

6. *Garcia R., Jarvi J., Lumsdaine A., Siek J., Willcock J.* An extended comparative study of language support for generic programming // J Funct. Program., 17(2), 2007. P. 145–205.

7. *Gregor D., Jarvi J., Siek J., Stroustrup B., Dos Reis G., Lumsdaine A.* Concepts: Linguistic Support for Generic Programming in C++ // ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06), Portland, Oregon. 2006. P. 291–310.

8. *Sutton A., Stroustrup B.* Design of Concept Libraries for C++ // Proc. SLE 2011 (International Conference on Software Language Engineering). 2011. P. 97–118.

9. *Cardelli L., Wegner P.* On Understanding Types, Data Abstraction, and Polymorphism // ACM Computing Surveys. New York, NY, USA: ACM. 1985. Т. 17. № 4. P. 471–523.

10. *Хорстманн К.* Scala для нетерпеливых / М.: ДМК Пресс, 2013. 408 с.

11. *Stroustrup B.* The C++0x "Remove Concepts" Decision // Электронное издание Dr.Dobb's Journal. 2009. http://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111