Министерство образования и науки Российской Федерации

Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Труды научной школы И. Б. Симоненко

Выпуск второй

Ростов-на-Дону
Издательство Южного федерального университета
2015

УДК 517.9; 519.6
ББК 22.161.6; 22.19
Т78

Печатается по решению редакционной комиссии по математическим наукам
Института математики, механики и компьютерных наук им. И. И. Воровича
(протокол № 7 от 6 ноября 2015 г.)

Т78 Труды научной школы И. Б. Симоненко. Выпуск второй / под ред. М. Э. Абрамяна, Я. М. Ерусалимского, В. С. Пилиди, Б. Я. Штейнберга ; Южный федеральный университет. – Ростов-на-Дону : Издательство Южного федерального университета, 2015. – 304 с.
ISBN 978-5-9275-1607-0

В юбилейном сборнике, посвященном 80-летию известного российского математика И. Б. Симоненко, представлены работы по теории операторов, математической физике, асимптотическим методам, методам программирования, теории кодирования, распараллеливанию программ, алгоритмам на графах. Рекомендуется научным работникам, преподавателям вузов, аспирантам.

Статьи публикуются в авторской редакции.

ISBN 978-5-9275-1607-0

УДК 517.9; 519.6
ББК 22.161.6; 22.19

От редакторов

Профессору, доктору физико-математических наук Игорю Борисовичу Симоненко (16.08.1935 – 22.03.2009) в этом году могло бы исполниться 80 лет. Он оставил после себя 230 публикаций, которые отличаются не только глубиной исследований, но и разнообразием тем: алгебраическая топология, функциональный анализ, теория псевдодифференциальных операторов, теория функций комплексного переменного, дифференциальные уравнения, теория краевых задач Римана, теория линейных операторов, гидромеханика, гидроакустика, электростатика, методы приближенных вычислений, дискретная математика, нормированные кольца, теория выпуклых множеств. Среди его работ и замечательная научно-популярная статья «Как доказывать трансцендентность чисел».

Творческое наследие Игоря Борисовича не сводится только к его публикациям. Игорь Борисович Симоненко искал внедрения математических достижений в другие науки или сферы человеческой деятельности, и эти поиски имели много проявлений. Он руководил хоздоговорными работами по электростатике (для проектирования электронных схем в ТРТИ, г. Таганрог) и гидроакустике. Его теоретические результаты о возможности возникновения вибрационной конвекции в невесомости были подтверждены экспериментом на американской космической станции «Скайлэб». На его имя (с соавторами) зарегистрировано изобретение «Исследование горизонтального статического взаимодействия электровоза и пути». Серию публикаций последних лет Игорь Борисович посвятил разработке новых методов вычислений с оценками быстродействия и оценками погрешностей.

Заслуженный деятель науки Российской Федерации, Игорь Борисович возглавлял крупную научную школу, в состав которой входили и входят его 33 непосредственных ученика (кандидаты и доктора наук) и еще 60 «учеников учеников». Множество государств, в которых работают ученые из этого списка, говорит об огромном влиянии И. Б. Симоненко на мировую науку. К последователям И. Б. Симоненко относят себя и многие математики, которые выступали с докладами на научном семинаре Игоря Борисовича.

В этом сборнике представлены работы его непосредственных учеников, учеников его учеников, а также коллег Игоря Борисовича, которые считают себя причастными к его научной школе.

СРЕДСТВА ОБОБЩЁННОГО ПРОГРАММИРОВАНИЯ В СОВРЕМЕННЫХ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКАХ. ЧАСТЬ 1. АНАЛИЗ ПРОБЛЕМ

Белякова Ю. В., Михалкович С. С.
Южный федеральный университет, Ростов-на-Дону

Generic programming is supported in programming languages by various techniques, C# and Java providing the poorest ones. Essential drawbacks of the instruments for generic programming in these mainstream object-oriented languages are carefully discussed in this paper. Pitfalls of F-bounded polymorphism integrated with classes and interfaces as well as so-called concept pattern, which is widely used in C#, Java, and Scala generic libraries, are considered.

1 Введение

В рамках парадигмы обобщённого программирования (generic programming), предложенной Александром Степановым и Дэвидом Массером в 1989 году [1], алгоритмы и структуры данных реализуются для абстрактных, а не конкретных типов и операций. Например, вместо множества похожих реализаций алгоритма поиска элемента в контейнере для разных типов данных (целого значения в массиве, символа в строке, строки в односвязном списке и т. д.), можно написать один *обобщённый* алгоритм поиска элемента в абстрактном контейнере. Инстанцирование (instantiation) или «подстановка» в этот обобщённый алгоритм конкретных типов и операций порождает всё разнообразие конкретных алгоритмов поиска. Таким образом обобщённое программирование позволяет избежать дублирования кода алгоритма.

Поддержка обобщённого программирования (ОП) в языках программирования реализуется различными средствами: существует два основных подхода к организации этих средств. Первый условно можно обозначить как *«разрешено всё, что не запрещено»*, классическим примером которого являются «неограниченные» шаблоны C++ [2]. Код шаблона может включать любые синтаксически корректные конструкции, проверка семантики для него *не* выполняется – она откладывается до этапа инстанцирования шаблона конкретными типами. При этом возможны следующие ситуации:

1. Код шаблона содержит семантические ошибки, из-за которых не может быть откомпилирована никакая инстанция шаблона. Например, ошибка приведения типов в операторе присваивания `int x = "foo"`.

2. Одни инстанции шаблона компилируются, другие – нет. Это означает, что не все типы «подходят» для инстанцирования данного шаблона, но код шаблона корректен.

1.1 Проблемы шаблонов C++

Неограниченные шаблоны дают программисту почти полную свободу при написании обобщённого кода, так как допускают использование всех синтаксически корректных конструкций в теле шаблона. Однако, несмотря на такую выразительность и гибкость, механизм шаблонов обладает серьёзными недостатками:

1. *Позднее обнаружение ошибок* [5]. Даже если код шаблона некорректен для всех типов (сиуация 1, Введение), это не может быть обнаружено на этапе «компиляции» шаблона – только на этапе инстанцирования конкретными типами. Кроме того, даже столкнувшись с ошибкой при инстанцировании, не всегда можно сразу отличить ситуации 1 и 2.

2. *Сложные сообщения об ошибках и нарушение инкапсуляции* [6]. Если код шаблона не может быть инстанцирован только некоторыми типами (ситуация 2, Введение), это означает, что данные типы не удовлетворяют ограничениям, которые *неявно* накладываются в теле шаблона. Но сообщения об ошибках не отражают этого непосредственно, так как у компилятора нет возможности проверить соответствие типов ограничениям. Вместо этого текст ошибки отсылает программиста к коду шаблона, тем самым раскрывая его реализацию. Такая проблема встречается достаточно часто при работе со стандартной библиотекой шаблонов C++.

3. *Невозможность раздельной компиляции* [7]. Код шаблона хранится в виде синтаксического дерева [5] и используется каждый раз при инстанцировании: типовые параметры в теле шаблона заменяются конкретными типами, а полученный код компилируется. Таким образом, код самого шаблона не может быть откомпилирован в модуль.

1.2 Механизм обобщённого программирования на основе явных ограничений

Второй подход симметрично можно назвать *«запрещено всё, что не разрешено»* – это механизм обобщённого программирования на основе явных ограничений. В этом случае при определении обобщённого типа или алгоритма на типовые параметры *явным образом*, при помощи конструкций языка, накладывается ряд ограничений (constraints), например: для объектов типа определена операция «+», у типа есть конструктор без параметров, etc. Это позволяет выполнить полный семантический анализ обобщённого кода, причём независимо от инстанций – относительно этих ограничений. Если компиляция прошла успешно, гарантируется, что данный код может быть инстанцирован любыми типами, удовлетворяющими ограничениям, поэтому проверка типов при инстанцировании обобщённого кода сводится к проверке ограничений для типов-аргументов. Механизм ОП на основе ограничений реализуется различными средствами во многих языках программирования, например: классы типов (type classes) в Haskell, подтипы и интерфейсы (interfaces) в C# и Java, трейты[1] (traits) в Scala, сигнатуры (signatures) и функторы (functors) в SML и т. д. В сообществе C++ также идёт работа над новой конструкцией языка, так называемыми концептами[2] (concepts) [5, 9], которые должны обеспечить поддержку явных ограничений на параметры шаблона.

Механизм явных ограничений, как правило, лишён недостатков шаблонов C++, обсуждавшихся в Разд. 1.1. Его узким местом является выразительная мощь ограничений – удобство и гибкость средств работы с ними в языках программирования существенно отличаются [3]. Некоторые средства достаточно выразительны (например, классы типов Haskell), другие – довольно бедны. К числу последних относятся ограничения на основе подтипирования и интерфейсов в языках C# и Java, которые реализуют *F-ограниченный полиморфизм* [20]. К наиболее известным проблемам обобщённых средств в этих языках можно отнести: отсутствие ретроактивного моделирования [3, 15] (Разд. 2.1), отсутствие ассоциированных типов [3, 10] (Разд. 2.3), проблема мультипараметрических ограничений (то есть ограничений, связывающих несколько типов) [3, 13] (Разд. 2.4). Есть и другие недостатки, более скудно освещённые в известных авторам источниках, которые, однако, представляются не менее важными. Они будут проанализированы далее.

[1] В некоторых русскоязычных источниках traits переводятся как «характеристики».

[2] В некоторых русскоязычных источниках concepts переводятся как «концепции».

Следует отметить, что на сегодняшний день именно F-ограниченный полиморфизм является основной моделью обобщённого программирования в *объектно-ориентированных (ОО)* языках. Недостатки данного механизма частично компенсируются использованием так называемого концепт-паттерна [17]. Однако, использование этого паттерна создаёт новые проблемы: они будут рассмотрены в Разд. 3.

2 Недостатки средств обобщённого программирования, реализующих F-ограниченный полиморфизм на основе интерфейсов

В «промышленном масштабе» F-ограниченный полиморфизм [20] был реализован в таких объектно-ориентированных языках как C# [11] и Java [14] в конце 1990-х – начале 2000-х годов. Довольно скоро стало ясно, что по выразительной силе и удобству обобщённого программирования эти языки значительно уступают, в частности, многим функциональным языкам [3, 12]. В этом разделе рассматриваются основные недостатки средств обобщённого программирования на основе F-ограниченного полиморфизма и интерфейсов. Мы будем использовать синтаксис языка C#, но все рассуждения в равной степени относятся и к Java.

2.1 Невозможность ретроактивного моделирования

Основным способом выразить требования к типовым параметрам обобщённого кода в C# являются ограничения вида «типовый параметр реализует интерфейс», которые представляют собой разновидность ограничений подтипирования. На сегодняшний день в большинстве ОО-языков программирования, в том числе в C#, Java, C++ и Scala, используется механизм подтипирования, при котором ограничения вида «тип `B` является подтипом `A`»/«тип `B` реализует интерфейс `I`» могут быть удовлетворены, только если в *определении* типа `B` присутствует соответствующая декларация. Если тип `B` уже определён без такой декларации, его нельзя «заставить» ни стать подтипом `A`, ни реализовать интерфейс `I`, даже если синтаксически тип удовлетворяет интерфейсу. Возможность добавить отношение «тип `B` реализует (моделирует) интерфейс `I`» после определения типа `B` называют ретроактивным моделированием или ретроактивной реализацией интерфейсов.

```
(ICmp-1)  interface IComparable<T> { int CompareTo(T other); }
(s-1)     void Sort<T>(T[]) where T : IComparable<T>;
```

Рис. 1. Интерфейс `IComparable<T>` и его применение (C#)

```
(1) interface IComparableTo<S> { int CompareTo(S other); }
    int Find<T, S>(T[] vals, S x) where T : IComparableTo<S> {...}

(2) interface IComparable<T> where T : IComparable<T>
        { int CompareTo(T other); }
    void Sort<T>(T[] vals)   where T : IComparable<T> {...}
```

Рис. 2. Интерфейсы `IComparable<T>` и `IComparableTo<S>` (C#)

Рассмотрим, например, обобщённый алгоритм сортировки из стандартной библиотеки .NET (см. Рис. 1) и тип `Foo`:

```
class Foo { ... int CompareDefault(Foo other); ... }
```

Семантически тип `Foo` соответствует ограничениям метода `Sort<T>`: объекты типа `Foo` можно сравнивать. Но `Sort<Foo>` не является допустимой инстанцией метода сортировки, так как тип `Foo` не реализует интерфейс `IComparable<Foo>`, и нет способа сделать её таковой. Единственный выход – использовать паттерн Адаптер: необходимо описать новый класс `FooCmp`, реализующий интерфейс `IComparable<FooCmp>`, создать массив объектов `FooCmp` из исходных объектов типа `Foo`, отсортировать объекты типа `FooCmp` и извлечь из них отсортированный массив объектов типа `Foo`.

2.2 Проблемы рекурсивных ограничений

Возможность накладывать рекурсивные ограничения (или F-ограничения) вида «`T : I<T>`» – главное преимущество F-ограниченного полиморфизма [20] по сравнению с классической System $F_{<:}$, в которой допускаются только не рекурсивные ограничения подтипирования. Рекурсивные ограничения позволяют решить проблему бинарных методов [21] в объектно-ориентированных языках. Однако, они вносят дополнительную сложность и неясность в обобщённое программирование, и делают обобщённые определения громоздкими.

Пример 1. В названии интерфейса `IComparable<T>` термин «comparable» будто бы описывает умение объектов типа `T` сравнивать себя друг с другом. Определение же интерфейса (`(ICmp-1)`, Рис. 1) лишь описывает умение объектов *некоторого* типа (реализующего данный интерфейс) сравнивать себя с объектами типа `T`. Поэтому, для фиксации отношения «тип `T` сравним с собой», в заголовке алгоритма сортировки указано ре-

```
interface IDataVertex<Vertex, DataType>
    where Vertex : IDataVertex<Vertex, DataType>    // (1)
{
    ...
    IEnumerator<Vertex> OutVertices { get; }
}
interface IDataGraph<Vertex, DataType>
    where Vertex : IDataVertex<Vertex, DataType>    // (2)
{ ... }
```

Рис. 3. Интерфейсы `IDataGraph<,>` и `IDataVertex<,>` (C#)

курсивное ограничение `where T : IComparable<T>`, которое может выглядеть неестественно, но является необходимым.

С точки зрения чистоты семантики, было бы лучше разделить интерфейс `IComparable<T>` на два разных (Рис. 2):

1. `IComparableTo<S>`, описывающий умение объектов некоторого типа сравнивать себя с объектами типа `S`. Соответствующее ограничение можно использовать в методе `Find<T, S>`.
2. `IComparable<T>`, описывающий умение объектов типа `T` сравнивать себя друг с другом, что и требуется в методе `Sort<T>`.

Пример 2. Для решения ряда задач анализа потоков данных в области оптимизирующих компиляторов [22] удобно ввести обобщённое определение графа со специальной структурой: каждая вершина графа хранит некоторую информацию и ссылки на входящие/исходящие вершины, а сам граф состоит из множества таких мета-вершин вместо рёбер. На Рис. 3 приведены выдержки из соответствующих определений: `IDataGraph<Vertex, DataType>` описывает интерфейс графа; `IDataVertex<Vertex, DataType>` описывает интерфейс вершины такого графа. Заметим, что мы вынуждены включать типовый параметр `Vertex` (тип вершины) в определение интерфейса вершины по тем же причинам, что и в случае интерфейса `IComparable<T>`, а также дублировать в определении интерфейса графа рекурсивное ограничение (1). Без ограничений (1) и (2) проверка типов допустила бы несогласованный граф `IDataGraph<V2, int>`, в котором вершины типа `V2` ссылаются на вершины типа `V1`, где типы вершин `V1`, `V2` определены следующим образом:

```
class V1 : IDataVertex<V1, int> { ... }
class V2 : IDataVertex<V1, int> { ... }
```

```
interface GraphEdge<Vertex> { ... }
interface IncidenceGraph<Vertex, Edge, OutEdgeIter>
    where Edge : GraphEdge<Vertex>,                          (*-1)
    where OutEdgeIter : IEnumerable<Edge> { ... }            (#-1)

G_Vrtx fst_neighb<G, G_Vrtx, G_Edge, G_OutEdgeIter>(G g, G_Vrtx v)
    where G : IncidenceGraph<G_Vrtx, G Edge, G_OutEdgeIter>,
    where G_Edge : GraphEdge<G_Vrtx>,                        (*-2)
    where G_OutEdgeIter : IEnumerable<G_Edge> { ... }        (#-2)
```

Рис. 4. Пример работы с обобщённым графом в C#

2.3 Отсутствие ассоциированных типов и распространения ограничений

Одной из серьёзных проблем обобщённых средств C# и Java является «многословность» заголовков обобщённых конструкций. Она вытекает из необходимости дублировать ограничения на типовые параметры обобщённого кода, а также большого числа типовых параметров, возникающего в чуть менее тривиальных примерах, чем, например, сортировка.

На Рис. 4 представлены: заголовки интерфейсов `IncidenceGraph<,,>` и `GraphEdge<>` [12], которые соответствуют определениям концептов[3] `Graph Edge` и `Incidence Graph` из C++-библиотеки BGL (Boost Graph Library); метод `fst_neighb<,,,>` поиска первой смежной вершины для вершины `v`. Заметим, что ограничения `(*-2)` и `(#-2)` в точности повторяют ограничения `(*-1)` и `(#-1)` соответственно: их необходимо дублировать, чтобы ограничение «тип `G` реализует интерфейс `IncidenceGraph<,,>`» в методе `fst_neighb<,,,>` было корректно. Вторая проблема: метод поиска смежной вершины содержит четыре типовых параметра, хотя для аннотации параметров `g` и `v` алгоритма нужны лишь типы графа и вершины. Однако, чтобы выразить ограничение на тип графа, наравне с типом `G` приходится добавлять в обобщённый метод все типовые параметры интерфейса `IncidenceGraph<,,>`. При этом ясно, что типы вершин, ребёр, и итератора по рёбрам на самом деле всецело зависят от типа графа.

Возможным решением указанных проблем является расширение интерфейсов ассоциированными типами, рассмотренное в работе [12]: оно позволяет внести тип вершины в интерфейс ребра, а в интерфейс графа – типы вершины, ребра и итератора. Тем самым число типовых параметров метода `fst_neighb<>` сокращается до одного – типа графа. Благодаря распространению ограничений отпадает также необходимость дублировать

[3] В контексте стандартной библиотеки шаблонов C++ (STL [2]) концептом называют именованный набор требований к типу, неформально описанный в документации.

```
interface IncidenceGraph
{
    type Vertex;    // ассоциированный тип
    type Edge : GraphEdge;
    type OutEdgeIterator : IEnumerable<Edge>;
    ...
}
```

Рис. 5. Интерфейс `IncidenceGraph` в C# с ассоциированными типами

```
interface IFoo { void Init(int value); ... }

List<T> bar<T>(...) where T : new(), IFoo
{
    ...
    List<T> tvals = new List<T>();
    for (int i = ...)
    {
        T x = new T();
        x.Init(i);
        tvals.Add(x);
    }
    return tvals;
}
```

Рис. 6. Конструирование объектов типа-параметра (C#)

ограничения на тип ребра и итератора в `fst_neighb<>`, так как они уже указаны в определении интерфейса графа. В результате, на расширенном C# [12] заголовок `fst_neighb<>` сокращается до:

```
G::vertex_type fst_neighb<G>(G g, G::Vertex v)
    where G : IncidenceGraph { ... }
```

Интерфейс графа в C# с ассоциированными типами приведён на Рис. 5.

2.4 Отсутствие статических методов и проблема мультипараметрических ограничений

Интерфейс описывает функциональность *одного объекта* некоторого типа, реализующего интерфейс, и не может содержать статических методов. Возможность описывать статические методы решила бы проблему создания объектов типов-параметров внутри обобщённого кода: на Рис. 6 приведён пример обобщённого метода `bar<T>`, в котором требуется создать список объектов типа `T`. Ограничение `T : new()` указывает на наличие в типе `T` конструктора без параметров, другие виды конструкторов указать нельзя. Поэтому, если объекты типа `T` должны инициализироваться

```
interface IObserver<O, S> where O : IObserver<O, S>
                          where S : ISubject<O, S>
{ void Update(S subj); }

interface ISubject<O, S>  where O : IObserver<O, S>
                          where S : ISubject<O, S>
{ void Register(O obs); ... }

void GenericUpdate<S, O>(S subject, O observer)
    where O : IObserver<O, S>
    where S : ISubject<O, S>
{ observer.Update(subject); }
```

Рис. 7. Паттерн наблюдатель (C#)

какими-то данными (например, целым числом), придётся это делать не в конструкторе, а с помощью отдельного метода инициализации, как это сделано в интерфейсе `IFoo` (метод `Init`). Если бы интерфейсы поддерживали статические методы, можно было бы заменить вызов конструктора и метода инициализации на один вызов статического метода `Build`:

```
interface IFoo { static void T Build(int value); ... }
...
    for (int i = ...)
        tvals.Add(T.Build(i));
```

Было бы также удобно добавить в стандартную библиотеку интерфейс `Parseable<T>`, описывающий возможность преобразования строки в объект типа T. Сейчас для встроенных типов в C# определены методы `Parse` и `TryParse`, никак не связанные друг с другом.

```
interface Parseable<T>          // ошибка
{
    static T Parse(string s);
    static bool TryParse(string s, out T result);
}
```

Ещё одно неудобство – для пользовательских типов перегрузка стандартных операторов («!», «+», «*», etc.) выполняется через определение статических методов; например, для оператора «+» это делается так:

```
class Foo { ... public static Foo operator+(Foo a, Foo b) { ... } }
```

Интерфейс не может содержать статических методов, в том числе определений операторов, поэтому операторы нельзя использовать внутри обобщённого кода (так как не существует соответствующих интерфейсов-ограничений).

Ещё одно слабое место интерфейсов – невозможность естественным образом выразить мультипараметрические ограничения – ограничения на

```
(ICmp-2)  interface IComparer<T> { int Compare(T x, T y); }
(s-2)     void Sort<T>(T[], IComparer<T>);
```

Рис. 8. Интерфейс `IComparer<T>` и его применение

несколько типов. На Рис. 7 представлено определение паттерна Наблюдатель средствами интерфейсов (этот пример приводится для языка Java в [15]): каждый из интерфейсов `IObserver<O, S>` и `ISubject<O, S>` описывает функциональность объектов одного типа из пары `O`, `S`, причём не очевидно, какого именно, так как на оба параметра наложены аналогичные рекурсивные ограничения; чтобы установить связь между `O` и `S`, в каждом интерфейсе приходится дублировать рекурсивные ограничения на оба типовых параметра; то же самое необходимо делать и в обобщённом коде (метод `GenericUpdate<S, O>`).

2.5 Неоднозначная семантика интерфейсов

Дополнительную сложность работы с обобщённым кодом накладывает неоднозначная роль интерфейсов: они используются и как *типы*, в своей изначальной роли в объектно-ориентированном программировании, и как *ограничения* на типовые параметры. Например, в следующем примере `ICollection<T>` – интерфейс-тип, а `IComparable<T>` – интерфейс-ограничение:

```
void Sort<T>(ICollection<T>) where T : IComparable<T>;
```

Мы полагаем, что во избежание подобной двойственности, ограничения на типы-параметры должны выражаться при помощи принципиально другой конструкции языка.

3 Концепт-паттерн

В Разд. 2.1 была рассмотрена проблема отсутствия ретроактивного моделирования. Частичное решение этой проблемы доставляет использование концепт-паттерна (concept pattern) [17]: вместо того, чтобы накладывать ограничения-интерфейсы на типовый параметр, можно передать в обобщённый код в качестве аргумента некоторый объект, реализующий требуемые ограничения для этого типа. Заголовок соответствующей версии[4] алгоритма сортировки массива из стандартной библиотеки .NET приведён на Рис. 8. Объект с интерфейсом `IComparer<T>` выступает в ро-

[4]Стандартная библиотека .NET содержит обе версии `Sort<>`: и с ограничением-интерфейсом на типовый параметр, и с внешним объектом-компаратором.

```
static HashSet<T> GetUnion<T>(HashSet<T> s1, HashSet<T> s2)
{
    var us = new HashSet<T>(s1, s1.Comparer);
    us.UnionWith(s2);
    return us;
}
```

Рис. 9. Объединение множеств типа `HashSet<T>`

ли объекта-концепта, обеспечивающего возможность сравнения элементов типа `T`. Таким образом, если требуется отсортировать объекты типа `Foo`, не реализующего интерфейс `IComparable<Foo>` (`(ICmp-1)`, Рис. 1), необходимо определить *новый* класс, реализующий интерфейс `IComparer<Foo>`, и передать объект этого класса в `Sort<Foo>` (`(ICmp-2)`, Рис. 8). Соответствующий объект называют моделью концепта `IComparer<>` для типа `Foo`.

Заметим, что для удобства пользователей библиотеки необходимо иметь обе версии `Sort<T>`: и для «встроенного» ограничения (версия `(s-1)`, `Рис. 1`), и для внешнего объекта-модели (версия `(s-2)`, `Рис. 8`). Если ограничиться только второй версией, придётся определять внешние классы компараторов всегда, даже если сам тип реализует все необходимые операции. Но если метод можно предоставить в форме нескольких перегруженных версий, класс должен иметь только одно определение. Поэтому классы всегда хранят объекты-концепты в качестве *полей*. Например, каждый объект-множество типа `SortedSet<T>` содержит компаратор `IComparer<T>`, который инициализируется в конструкторе либо явно – аргументом конструктора, либо неявно – компаратором по умолчанию (если тип-параметр реализует `IComparable<T>`).

Поскольку объект-концепт является внешним по отношению к типовому параметру, его экземплярные методы могут служить заменой статическим методам этого типового параметра. Вместо интерфейса `Parseable<T>` (Разд. 2.4) можно определить интерфейс `Parser<T>`:

```
interface Parser<T> { T Parse(string s); }
void foo<T>(..., Parser<T> p) { ... T x = p.Parse(str); ... }
```

Таким образом, концепт-паттерн решает и проблему отсутствия статических методов в интерфейсах.

3.1 Проблема несогласованности моделей

Концепт-паттерн позволяет использовать различные модели ограничений для одной и той же инстанции обобщённого кода, что, несомненно, удобно. Например, используя разные компараторы в версии `(s-2)` метода

`Sort<T>` (Рис. 8), можно отсортировать числа по возрастанию и убыванию. Однако, такая гибкость одновременно является и серьёзной проблемой.

Рис. 9 иллюстрирует простой пример, когда динамический характер моделей в концепт-паттерне ведёт к нетривиальной ошибке. Метод `GetUnion<T>` возвращает объединение двух множеств типа `HashSet<T>`, используя для результирующего множества в качестве компаратора компаратор первого аргумента. Пусть есть два множества строк: `s1` и `s2` типа `HashSet<string>`, причём в первом используется регистрозависимое сравнение строк, а во втором – регистронезависимое. Несмотря на то, что оба множества имеют *одинаковый тип*, результат `GetUnion(s1, s2)` будет отличаться от `GetUnion(s2, s1)`, так как объекты `s1`, `s2` используют разные модели сравнения на равенство. Более естественным было бы считать эти множества принадлежащими к разным типам, но проверить согласованность моделей невозможно на этапе компиляции кода. Этот тонкий момент может стать большим сюрпризом для программиста. Стоит отметить, что рассмотренная особенность концепт-паттерна была осознана как проблема не так давно: этот вопрос обсуждается в более ранней работе авторов [13], а также в [16].

Концепт-паттерн используется во многих языках, реализующих F-ограниченный полиморфизм и не поддерживающих ретроактивное моделирование, как, например, C#, Java или Scala. В Scala, благодаря сочетанию трейтов (traits) и имплиситов (implicits), использование концепт-паттерна проще для программиста, и часто можно обойтись более коротким кодом, чем в C# или Java. Однако, необходимость в дополнительных полях обобщённых классов и проблема несогласованности моделей сохраняются. Например, конструктор класса `TreeSet[A]` принимает объект-концепт типа `Ordering[A]`:

```
class TreeSet[A] extends SortedSet[A] ... {
    TreeSet()(implicit ordering: Ordering[A]) {...} // конструктор
    ...
}
```

Поскольку `ordering` – это имплисит-параметр, его можно указывать не всегда: при наличии имплисит-модели `Ordering[]` для конкретного типа (то есть модели по умолчанию) эта модель будет использована автоматически. Дополнительную информацию о возможностях обобщённого программирования на Scala можно найти в работах [17–19].

4 Заключение

В данной статье представлен анализ механизма обобщённого программирования, используемого в промышленных объектно-ориентированных языках C# [11] и Java [14], которые реализуют F-ограниченный полиморфизм [20] на основе типовых параметров, ограничений подтипирования, и интерфейсов. Выразительные возможности средств ОП C#/Java весьма ограничены по сравнению с другими языками программирования, поддерживающими обобщённое программирование [3]. Недостатки этих средств, в том числе мало освещённые в литературе проблемы рекурсивных ограничений и неоднозначной семантики интерфейсов, подробно обсуждаются в Разд. 2. Основываясь на результатах выполненного анализа, можно сделать несколько выводов:

1. Часть недостатков средств обобщённого программирования C#/Java обусловлена тем, что они реализуют F-ограниченный параметрический полиморфизм. Такая разновидность полиморфизма, в частности, не позволяет естественным образом выражать мультипараметрические ограничения. Как и для решения проблемы бинарных методов, в этом случае используются рекурсивные ограничения, сложность которых рассматривается в Разд. 2.2.

2. Другие проблемы (например, отсутствие ретроактивного моделирования, невозможность выразить статические методы, отсутствие ассоциированных типов) связаны с особенностями конкретной конструкции языка – интерфейсами, которые с появлением поддержки ОП в C#/Java стали использоваться в качестве ограничений на типовые параметры обобщённого кода.

Заметим, что попытка «обойти» недостатки второй категории с помощью концепт-паттерна [17] влечёт новые проблемы, в том числе серьёзную проблему несогласованности моделей на этапе выполнения кода, которая подробно обсуждается в Разд. 3.1. Однако, стандартные обобщённые библиотеки языков C#, Java и Scala построены преимущественно на использовании концепт-паттерна.

Улучшить поддержку ОП в языке, не прибегая к использованию концепт-паттерна, можно путём расширения/модификации нотации интерфейсов или интеграции ограниченного полиморфизма с другими, более выразительными конструкциями. Альтернативный, более радикальный подход предполагает отказ от ограниченного полиморфизма в пользу

неограниченного параметрического полиморфизма, который поддерживался бы с помощью новых средств языка, отличных от интерфейсов. Во второй части данной статьи [4] мы рассмотрим поддержку обобщённого программирования в современных объектно-ориентированных языках Scala, Ceylon, Rust, и Swift, а также некоторых расширениях языков C# и Java, которые демонстрируют оба этих подхода. В числе прочих будет рассмотрено и разработанное нами расширение языка C# с концептами.

ЛИТЕРАТУРА

[1] *David R. Musser, Alexander A. Stepanov.* Generic Programming // Proceedings of the International Symposium ISSAC '88 on Symbolic and Algebraic Computation, p. 13–25. Springer-Verlag, London, UK, UK, 1989.

[2] *Alexander A. Stepanov, Meng Lee.* C++ Standard Template Library // HP Laboratories, Technical Report 95-11 (R.1), November 1995.

[3] *Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock.* An Extended Comparative Study of Language Support for Generic Programming // J. Funct. Program., 17(2), March 2007, p. 145–205. Cambridge University Press, New York, NY, USA.

[4] *Белякова Ю. В., Михалкович С. С.* Средства обобщённого программирования в современных объектно-ориентированных языках. Часть 2. Обзор новых решений // Труды научной школы И. Б. Симоненко. Второй выпуск. Ростов н/Д: Изд-во ЮФУ, 2015. С. 78–92.

[5] *Bjarne Stroustrup, Gabriel Dos Reis.* Concepts — Design Choices for Template Argument Checking. C++ Standards Committee Papers, Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, October 2003.

[6] *Gabriel Dos Reis, Bjarne Stroustrup.* Specifying C++ Concepts // Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL '06, p. 295–308. ACM, New York, NY, USA, 2006.

[7] *Herb Sutter, Tom Plum.* Why We Can't Afford Export. C++ Standards Committee Papers, TR N1426=03-0008, ISO/IEC JTC1/SC22/WG21, March 2003.

[8] *Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine.* Concepts: Linguistic Support for Generic Programming in C++ // Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications OOPSLA '06, p. 291–310. ACM, New York, NY, USA, 2006.

[9] *Andrew Sutton.* C++ Extensions for Concepts PDTS. C++ Standards Committee Papers, Technical Specification N4377, ISO/IEC JTC1/SC22/WG21, February 2015.

[10] *Jeremy Siek, Andrew Lumsdaine.* A Language for Generic Programming in the Large // Sci. Comput. Program., 76(5), May 2011, p. 423–465. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands.

[11] *Andrew Kennedy, Don Syme.* Design and Implementation of Generics for the .NET Common Language Runtime // SIGPLAN Not. 36(5), 2001, p. 1–12. ACM, NY, USA.

[12] *Jaakko Jarvi, Jeremiah Willcock, Andrew Lumsdaine.* Associated Types and Constraint Propagation for Mainstream Object-oriented Generics. // Proceedings of the OOPSLA '05, p. 1–19. ACM, New York, NY, USA, 2005.

[13] *Julia Belyakova, Stanislav Mikhalkovich.* Pitfalls of C# Generics and Their Solution Using Concepts // Proceedings of the Institute for System Programming, 27(3), June 2015, p. 29–45. Institute for System Programming RAS, Moscow, Russia.

[14] *Gilad Bracha, Martin Odersky, David Stoutamire, Philip Wadler.* Making the Future Safe for the Past: Adding Genericity to the Java Programming Language // Proceedings of the OOPSLA '98, p. 183–200. ACM, New York, NY, USA, 1998.

[15] *Stefan Wehr, Peter Thiemann.* JavaGI: The Interaction of Type Classes with Interfaces and Inheritance // ACM Trans. Program. Lang. Syst., 33(4), July 2011, p. 12:1–12:83. ACM, New York, NY, USA.

[16] *Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, Andrew C. Myers.* Lightweight, Flexible Object-oriented Generics // Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '2015, p. 436–445. ACM, New York, NY, USA, 2015.

[17] *Bruno Oliveira, Adriaan Moors, Martin Odersky.* Type Classes As Objects and Implicits // Proceedings of the ACM International Conference OOPSLA '10, p. 341–360. ACM, New York, NY, USA, 2010.

[18] *Bruno Oliveira, Jeremy Gibbons.* Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming // J. Funct. Program., 20(3–4), July 2010, p. 303–352. Cambridge University Press, New York, NY, USA, 2010.

[19] *A. Pelenitsyn.* Associated types and constraint propagation for generic programming in Scala // Programming and Computer Software, 41(4), p. 224–230. Pleiades Publishing, 2015.

[20] *Peter Canning, William Cook, Walter Hill, Walter Olthoff, John C. Mitchell.* F-bounded Polymorphism for Object-oriented Programming // Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture FPCA '89, p. 273–280. ACM, New York, NY, USA, 1989.

[21] *Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T. Leavens, Benjamin Pierce.* On Binary Methods // Theor. Pract. Object Syst., 1(3), Fall 1995, p. 221–242. John Wiley & Sons, Inc., New York, NY, USA.

[22] *Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman.* Compilers: Principles, Techniques, and Tools (2Nd Edition). Ch. "Code Optimization". Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

Содержание

Научное издание

ТРУДЫ НАУЧНОЙ ШКОЛЫ И. Б. СИМОНЕНКО

Выпуск второй

под ред. М. Э. Абрамяна, Я. М. Ерусалимского,
В. С. Пилиди, Б. Я. Штейнберга

Подписано в печать 10.12.2015 г. Заказ № 4906.
Тираж 100 экз. Формат 60×84 $^{1}/_{16}$. Усл. печ. л. 17,55. Уч.-изд. л. 13,04.
Отпечатано в отделе полиграфической, корпоративной и сувенирной продукции
Издательско-полиграфического комплекса КИБИ МЕДИА ЦЕНТРА ЮФУ.
344090, г. Ростов-на-Дону, пр. Стачки, 200/1, тел. (863) 247-80-51.