МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. Ломоносова
ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

ISSN 2411-1473

Современные информационные технологии и ИТ-образование

Научный журнал

Том 2 (№ 11)

Москва
2015

УДК [004:377/378](063)
ББК 74.5(0)я431+74.6(0)я431+32.81(0)я431
С 56

Современные информационные технологии и ИТ-образование. Т. 2 (№ 11), 2015. - 614 с. (ISSN 2411-1473)

В данном выпуске журнала представлены доклады X Юбилейной международной научно-практической конференции «Современные информационные технологии и ИТ-образование», прошедшей в Московском государственном университете имени М.В. Ломоносова 20-22 ноября 2015 года.

Журнал «Современные информационные технологии и ИТ-образование» включен в наукометрическую базу «Российский индекс научного цитирования» с размещением полнотекстовых версий в научной электронной библиотеке eLIBRARY.RU. URL: http://elibrary.ru/title_about.asp?id=52785

Издание осуществлено при финансовой поддержке Российского фонда фундаментальных исследований (Грант РФФИ № 15-07-20760_г)

Учредитель:
Фонд содействия развитию интернет-медиа, ИТ-образования, человеческого потенциала «Лига интернет-медиа»

Издатель:
Фонд содействия развитию интернет-медиа, ИТ-образования, человеческого потенциала «Лига интернет-медиа»

Адрес редакции:
119991, г. Москва, ГСП-1, Ленинские горы, д. 1, стр. 52, факультет ВМК МГУ имени М.В. Ломоносова, каб. 375. E-mail: sukhomlin@mail.ru, тел./факс (495) 939-46-26.

Журнал зарегистрирован Федеральной службой по надзору в сфере связи, информационных технологий и массовых коммуникаций (Роскомнадзор).
Свидетельство о регистрации средства массовой информации ПИ № ФС77-61433 от 10 апреля 2015 г.
Издается с 2005 года. Выходит 1 раз в год.

Содержание

Белякова Ю. В.[1], Михалкович С. С.[2]

[1] Южный федеральный университет, г. Ростов-на-Дону, ассистент кафедры информатики и вычислительного эксперимента, julbel@sfedu.ru

[2] Южный федеральный университет, г. Ростов-на-Дону, доцент кафедры алгебры и дискретной математики, miks@sfedu.ru

КОНЦЕПТ-ПАРАМЕТРЫ КАК МЕХАНИЗМ РАЗВИТИЯ СРЕДСТВ ОБОБЩЁННОГО ПРОГРАММИРОВАНИЯ В ЯЗЫКЕ C#

КЛЮЧЕВЫЕ СЛОВА

Обобщённое программирование, объектно-ориентированный язык, концепты, интерфейсы, ограничения, типовые параметры, C#, Java, Scala, F-ограниченный полиморфизм, концепт-паттерн, концепт-параметры.

АННОТАЦИЯ

Современные объектно-ориентированные языки реализуют различные инструменты обобщённого программирования. В статье обсуждаются ключевые критерии, по которым отличаются эти инструменты, сформулированные в результате их сравнительного анализа. Рассматриваются некоторые проблемы средств обобщённого программирования в промышленных языках C#/Java, которые привели к появлению целого ряда расширений, улучшающих поддержку обобщённого программирования в этих языках. На основании выделенных критериев разработан механизм обобщённого программирования на основе концепт-параметров, не уступающий средствам в других языках, но предлагающий принципиально иной стиль обобщённого программирования: мотивация и дизайн механизма для языка C# представлены в статье.

Введение

Обобщённое программирование (ОП) [1] поддерживается на уровне средств языка во многих языках программирования, однако по выразительной силе и удобству использования эти средства существенно отличаются [2]. К наиболее выразительным инструментам ОП относятся классы типов Haskell, в то время как промышленные объектно-ориентированные (ОО) языки программирования C# [3] и Java [4] обладают самыми бедными средствами: некоторые существенные недостатки этих средств рассматриваются в Разд. 2. В течение последних десяти лет было предложено несколько расширений C#/Java, улучшающих возможности обобщённого программирования; возникли новые ОО-языки с лу́чшей поддержкой ОП; был пополнен список возможностей, влияющих на удобство ОП [2,6,12,13]. Исследовав инструменты обобщённого программирования в этих ОО-языках и расширениях, мы выделили несколько критериев, которые характеризуют средства и определяют стиль обобщённого программирования в языке: они обсуждаются в Разд. 3. На основании этих критериев был разработан язык C#^{CP} – расширение C# с механизмом ОП на основе *концепт-параметров*, которые принципиально отличают обобщённое программирование в C#^{CP} от подходов к ОП, принятых в других языках, но не уступают им по выразительной силе. Дизайн концепт-параметров представлен в Разд. 4.

1. Обобщённое программирование

В отличие от обычного кода, который оперирует значениями конкретных типов (int, string, и т. д.), *обобщённые* алгоритмы и структуры данных зависят от *типовых параметров* и оперируют значениями этих типов-параметров. Примером обобщённой структуры данных является обобщённый класс Stack<T> – стек значений типа T, где T – типовый параметр, представляющий произвольный тип. Процесс подстановки конкретных типов вместо типовых параметров называют *инстанцированием* обобщённого кода, а полученный конкретный код – *инстанцией*. Например, инстанция класса Stack<int> представляет стек целых чисел, а Stack<string> – стек строк. Обобщённый код не всегда можно инстанцировать произвольными типами: например, в обобщённом алгоритме сортировки элементов типа T необходимо уметь сравнивать значения типа T друг с другом, поэтому инстанцировать алгоритм сортировки можно только типами, для

которых определена операция сравнения. Подобные требования («тип T поддерживает сравнение») называют *ограничениями* на типовые параметры.

2. Недостатки механизмов обобщённого программирования C#/Java

В C#/Java требования к типовым параметрам обобщённого кода явно выражаются с помощью *ограничений подтипирования* T : C, где T – это типовый параметр обобщённого кода, а C – конкретный класс, и *ограничений-интерфейсов* T : I, где I – интерфейс; в рамках *F-ограниченного полиморфизма* [5] допускаются также *рекурсивные* ограничения вида T : I<T>, где I<> – обобщённый интерфейс. На Рис. 1 представлен обобщённый интерфейс IComparable<T> из стандартной библиотеки .NET и выдержка из обобщённого алгоритма сортировки массива Sort<T>: секция where алгоритма содержит рекурсивное ограничение на типовый параметр T, благодаря которому в теле Sort<T> для значений типа T можно использовать метод сравнения CompareTo.

```
interface IComparable<T> { int CompareTo(T other); }
void Sort<T>(T[] values) where T : IComparable<T>
{ …
  if (values[i].CompareTo(values[j]) < 0) …
}
```

Рис. 1. Примеры обобщённого интерфейса и алгоритма (C#)

Интерфейсы C#/Java не поддерживают обращение к *собственному типу* (то есть типу, реализующему интерфейс), именно поэтому ограничение «в типе T есть метод сравнения со значением типа T» можно выразить только через рекурсивное обобщённое ограничение-интерфейс T : IComparable<T>.

Одним из наиболее существенных недостатков механизма ОП в C#/Java является отсутствие так называемого *ретроактивного моделирования*: если некоторый тип Foo не реализует интерфейс I, установить отношение Foo : I ретроактивно, то есть после определения типа, невозможно, даже если семантически тип удовлетворяет интерфейсу. Поэтому, например, алгоритм сортировки на Рис. 1 нельзя применить к массиву объектов типа Foo, определённому следующим образом:

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

Поскольку тип реализует интерфейс *единственным образом*, отсортировать значения этого типа с помощью алгоритма Sort<> также можно лишь одним способом. Например, целые числа можно отсортировать только по возрастанию, но не по убыванию. Другие серьёзные недостатки средств ОП C#/Java подробно обсуждаются в статьях [8,9], написанных нами, а также в [2,10,11].

```
interface IComparer<T> { int Compare(T a, T b); }
void Sort<T>(T[] values, IComparer<T> cmp)
{ …
  if (cmp.Compare(values[i], values[j]) < 0) …
}
```

Рис. 2. Пример использования концепт-паттерна (C#)

Проблемы ретроактивного моделирования и единственности реализации интерфейса решаются использованием *концепт-паттерна* [6]: вместо явного ограничения на типовый параметр в секции where, в обобщённый код в качестве параметра передаётся некий объект, реализующий требуемую от типа T функциональность; такой параметр называют *концептом*. На Рис. 2 представлена вторая версия алгоритма сортировки из стандартной библиотеки .NET, использующая концепт-паттерн. Теперь за сравнение отвечает не сам тип T, а внешний по отношению к T объект cmp типа IComparer<T> с операцией сравнения двух значений типа T. Чтобы отсортировать массив значений типа Foo, нужно определить новый класс, называемый *моделью*, который реализует интерфейс IComparer<Foo>, и передать объект этого класса в алгоритм сортировки. Используя объект другого класса, реализующего всё тот же интерфейс IComparer<Foo>, можно отсортировать массив другим способом.

Возможность использования различных моделей является одновременно и достоинством, и серьёзной проблемой концепт-паттерна. На Рис. 3 представлен метод объединения двух множеств типа HashSet<T>: класс множества HashSet<T> содержит в качестве поля концепт-

компаратор, который используется для сравнения элементов множества типа T; при объединении множеств для результирующего множества используется компаратор первого аргумента (s1.Comparer). Пусть есть два множества строк, ss1 и ss2 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(ss1, ss2) будет отличаться от GetUnion(ss2, ss1), так как объекты ss1, ss2 используют разные модели сравнения на равенство. Причём *согласованность этих моделей* нельзя проверить на этапе компиляции кода.

```
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. 3. Функция объединения множеств типа HashSet<T> (C#)

Заметим, что в примере на Рис. 2 интерфейс IComparer<T> выступает не в роли *ограничения* (как IComparable<T> на Рис. 1) а в роли *типа*, то есть своей изначальной роли в объектно-ориентированном программировании. Один блок кода может использовать интерфейсы и в обеих ролях. Например, алгоритм сортировки произвольной коллекции значений типа T:

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

Здесь интерфейс ICollection<T> используется в роли типа, а IComparable<T> – в роли ограничения. Подобная *неоднозначность семантики* интерфейсов усложняет обобщённое программирование и затрудняет восприятие кода. Представляется более правильным использовать интерфейсы исключительно в роли типов, а для *ограничения* типовых параметров обобщённого кода ввести **новую конструкцию языка**. В исследовании [7] на примере почти 14 млн. строк Java-кода было показано, что на практике один и тот же интерфейс никогда не используется в двух ролях, а либо всегда используется в роли типа, либо всегда в роли ограничения. Так, например, интерфейс Comparable<X> – полный аналог интерфейса IComparable<T> в C# – всегда используется в роли ограничения.

3. Поддержка обобщённого программирования в современных объектно-ориентированных языках и их расширениях

В последние годы появился целый ряд объектно-ориентированных языков программирования, в которых на смену «классическим» интерфейсам C# и Java пришли новые ОО-конструкции: в Scala [6,14] и Rust [15] это *трейты*, в Swift [16] – *протоколы*. Подобно C#/Java, в этих языках реализован механизм ОП на основе *явных ограничений*, но вместо интерфейсов для ограничения типовых параметров используются указанные конструкции, которые, как и интерфейсы, применяются и в качестве типов. Средства обобщённого программирования в упомянутых языках обладают бóльшей выразительной силой, чем аналогичные инструменты C#/Java. Вместе с тем для языков C# и Java было предложено несколько расширений, улучшающих возможности ОП посредством модификации интерфейсов или внедрения новых конструкций для ограничения типовых параметров: расширенные интерфейсы C# [10]; C#^{cpt} [8] – язык C# концептами, разработанный нами; генерализованные интерфейсы JavaGI [11]; Genus [12] – язык Java с конструкциями-ограничениями. Мы выполнили обзор различных средств обобщённого программирования в современных ОО-языках и расширениях C#/Java и провели их сравнительный анализ; результаты этого исследования представлены в работе [13]. Здесь мы приведём некоторые выводы, касающиеся современных подходов к организации средств ОП в ОО-языках.

Использование ограничений в роли типов. Возможны два варианта реализации ограничений на типовые параметры:

1. *Конструкции, используемые для ограничения типовых параметров, могут использоваться и как типы*. Этот подход используется во всех рассмотренных нами ОО-языках: C# [3], Java [4], Scala [14], Ceylon [17], Rust [15], Swift [16]. Возможность использования конструкции в качестве типа означает, что эта конструкция описывает функциональность *одного объекта* некоторого типа (как интерфейсы C#/Java и трейты Scala) и, возможно, статические методы этого типа (как интерфейсы Ceylon, трейты Rust и протоколы Swift). Из этого следует, что выразить естественным образом *мультипараметрические ограничения*, то есть ограничения на несколько типов, при таком подходе невозможно: для каждого типа из семейства связанных типов нужно определить обобщённую конструкцию,

типовые параметры которой представляют остальные типы из этого семейства, а также привести соответствующие ограничения для каждого типа. На Рис. 4 представлено определение обобщённого паттерна «Наблюдатель» на языке C# (аналогичный пример на Java был приведён ранее в [11]): этот паттерн связывает два типа – субъекта и наблюдателя, причём методы наблюдателя зависят от субъекта, а методы субъекта – от наблюдателя. Ограничения на оба типа приходится указывать трижды: в определениях интерфейсов для каждого из типов семейства, а также в обобщённом коде.

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

Рис. 4. Обобщённый паттерн Наблюдатель (C#)

Следует отметить, что трейты Rust и протоколы Swift поддерживают *собственные типы*, поэтому, в отличие от C#/Java/Scala, в этих языках не нужны рекурсивные ограничения, обсуждавшиеся в Разд. 2. Например, аналогичный интерфейсу IComparable<T> (Рис. 1) трейт Rust определяется следующим образом:

trait Comparable { **fn** CompareTo(**self**, other: **Self**) -> **bool**; }

Тип Self означает тип, реализующий данный трейт, а значение self – объект, для которого вызывается метод CompareTo. Соответственно, обобщённый код вместо рекурсивного ограничения-интерфейса T : IComparable<T> содержит не рекурсивное ограничение-трейт T : Comparable.

2. *Для ограничения типовых параметров обобщённого кода используются отдельные конструкции, которые не могут выступать в роли типов*. Такими конструкциями являются концепты G [18] (не путать с концепт-паттерном), мультипараметрические генерализованные интерфейсы JavaGI [11], ограничения Genus [12] и концепты C#^{cpt} [8]. На Рис. 5 приведён пример C#^{cpt} концепта сравнения CComparable[T] и обобщённого алгоритма сортировки, в котором используется соответствующие *концепт-ограничение* (или концепт-требование). Концепт-ограничение выступает в роли *предиката на типы*. При таком подходе нет принципиальной разницы между ограничениями на один и несколько типов: одна конструкция описывает все требования к семейству связанных типов, она же используется в качестве ограничения в обобщённом коде. Соответствующий пример для паттерна «Наблюдатель» приведён на Рис. 5. Заметим также, что параметром алгоритма сортировки на Рис. 5 является коллекция элементов типа T, представленная обобщённым интерфейсом ICollection<T>, но путаницы между типом и ограничением, как при использовании концепт-паттерна, не возникает.

```
concept CComparable[T] { int Compare(T a, T b); }
void Sort<T>(ICollection<T> values) where CComparable[T] {…}

concept CObserverPattern[S, O] {…}
void GenericUpdate<O, S>(S subj, O obs) where CObserverPattern[O, S] {…}
```

Рис. 5. Примеры обобщённого кода на C#^{cpt}

Ретроактивное моделирование несомненно является одной из важнейших возможностей обобщённого программирования. Подавляющее большинство современных языков и расширений поддерживают ретроактивное моделирование: в Rust и Swift разрешена ретроактивная реализация трейтов и протоколов, а для концептов G/C#^{cpt}, генерализованных интерфейсов JavaGI и ограничений Genus ретроактивно реализуются *модели* (модель задаёт способ, которым тип или набор типов реализуют концепт/генерализованный интерфейс/ограничение).

Множественное определение моделей. В языках C# и Java типы реализуют интерфейс единственным образом. То же касается интерфейсов, трейтов и протоколов в языках Ceylon, Scala, Rust и Swift, а также концептов G и генерализованных интерфейсов JavaGI, где для каждого набора типов в области видимости допускается лишь одна модель. Из-за единственности способа реализации ограничений обобщённый алгоритм сортировки, например, нельзя использовать для сортировки значений одного типа в разном порядке. В языках C#, Java и Scala множественная реализация симулируется использованием концепт-паттерна [6], проблемы которого обсуждались в Разд. 2. Лишь два расширения языков C# и Java – C#^{cpt} [8] и Genus [12] – поддерживают *множественное определение моделей* для одного набора типов. На Рис. 6 представлен заголовок класса HashSet<T>, определённого на C#^{cpt}: вместо поля-компаратора значений типа T используется концепт-ограничение на тип T. В методе объединения множеств это ограничение можно не указывать, так как оно выводится из использования типа HashSet<T> для параметров. При этом гарантируется, что для объектов s1 и s2 используется одна и та же модель компаратора: проверка выполняется на этапе *компиляции* кода. В примере на Рис. 6 объединение множеств ss1 и ss2 не допустимо, так как для множеств ss1 и ss2 используются разные модели концепта сравнения на равенство: для ss1 используется модель по умолчанию (определена с ключевым словом default), реализующая регистрозависимое сравнение строк, а для ss2 – модель StringEqCaseIS регистронезависимого сравнения.

```
class HashSet<T> where CEquatable[T] {…}
static HashSet<T> GetUnion<T>(HashSet<T> s1, HashSet<T> s2)
{
  var us = new HashSet<T>(s1);
  us.UnionWith(s2);   return us;
}

model default StringEqCaseS for CEquatable[string] {…}
model StringEqCaseIS for CEquatable[string] {…}

var ss1 = new HashSet<string>(…);
var ss2 = new HashSet<string>[using StringEqCaseIS](…);
var ss3 = GetUnion(ss1, ss2); // Ошибка компиляции!
```

Рис. 6. Решение проблемы согласованности моделей в C#^{cpt}

В Genus для применения моделей используется иной синтаксис: вызов конструктора множества ss2 выглядит как **new** Set[String **with** StringEqCaseIS](…). Ограничения записываются аналогично, в секции where. Авторы называют типы в Genus *типами, зависящими от моделей*: как и в C#^{cpt}, согласованность моделей проверяется на этапе компиляции.

4. Обобщённое программирование на основе концепт-параметров в языке C#^{CP}

В данном разделе мы представляем мотивацию и дизайн инструментов обобщённого программирования *на основе концепт-параметров* на примере языка C#^{CP} – расширения C# (компилятор C#^{CP} в настоящее время находится на стадии разработки). Подобно G, C#^{cpt}, JavaGI и Genus, для описания *ограничений* в C#^{CP} используется отдельная конструкция языка – *концепт*. Принципиальное отличие концептов C#^{CP} от концептов G [18]/C#^{cpt} [8], генерализованных интерфейсов JavaGI [11] и ограничений Genus [12] заключается в **способе ограничения** типовых параметров обобщённого кода: во всех перечисленных проектах ограничения имеют форму *предикатов на типы*. Заметим, что в G и JavaGI ограничения where C[T^1..T^N] действительно являются предикатами вида «для набора типов T^1..T^N определена модель концепта C[…]», так как в этих языках для данного набора типов можно определить лишь одну модель концепта. В то же время в C#^{cpt} и Genus, в которых поддерживается множественное определение моделей, ограничения в действительности *не* являются предикатами – при инстанцировании обобщённого кода на место каждого концепт-требования подставляется одна модель, выбранная из множества возможных; даже название «тип, зависящий от модели» говорит о том, что наряду с типовыми параметрами модель, фактически, также является *параметром* обобщённого типа. В C#^{CP} вместо концепт-ограничений (или концепт-требований) в форме предикатов обобщённые конструкции *явно зависят* от **концепт-параметров**.

Основываясь на результатах сравнительного анализа средств обобщённого программирования в современных объектно-ориентированных языках [13] и выводах, представленных в предыдущем разделе, мы выделили несколько ключевых вопросов, по которым должны быть приняты решения при проектировании дизайна средств обобщённого программирования:

1. Можно ли использовать в *роли типов* конструкции, применяющиеся для *ограничения типовых параметров* обобщённого кода, или ОО-элементы языка, через которые реализуется полиморфизм, и ограничения на типовые параметры представлены разными конструкциями;
2. Поддерживаются ли *ограничения подтипирования* и *надтипирования* на типовые параметры обобщённого кода;
3. Каким образом обеспечивается возможность *ретроактивного моделирования* ограничений типами;
4. Возможно ли *множественное* определение моделей.

В C#^{CP} концепты используются только для описания *ограничений*, а интерфейсы, в том числе обобщённые, выступают в качестве типов. Можно привести по крайней мере три аргумента в пользу такого решения вопроса 1: конструкции-ограничения, используемые в роли типов, плохо справляются с мультипараметрическими ограничениями (см. Разд. 3); использование одной и той же конструкции в различных ролях усложняет восприятия кода из-за неоднозначной семантики этой конструкции (см. Разд. 2); как показало исследование [7], множество интерфейсов-типов и множество интерфейсов-ограничений на практике действительно не пересекаются, поэтому невозможность использования ограничений в роли типов не вызовет практических проблем. Помимо требований к функциональности, концепты C#^{CP} поддерживают ограничения *подтипирования* T <: B и *надтипирования* T >: D, где T – типовый параметр, а B, D – конкретные классы. Для любого набора типов можно определить именованную модель концепта, причём моделей может быть несколько. Таким образом поддерживается *множественное ретроактивное* определение моделей.

```
concept Equality[T]
{  bool Equal(T x, T y);
   bool NotEqual(T x, T y) { return !Equal(x, y); } }

class HashSet<T ! Equality[T] eq>
{ …
   void UnionWith(HashSet<T!eq> other) {…}
}
static HashSet<T!eq> GetUnion<T ! Equality[T] eq>(HashSet<T!eq> s1,
                                HashSet<T!eq> s2)
{  var us = new HashSet<T!eq>(s1);
   us.UnionWith(s2);   return us;  }
```

Рис. 7. Примеры обобщённого кода в C#^{CP}

Рис. 7 иллюстрирует несколько примеров обобщённого кода с использованием концептов в языке C#^{CP}. Концепт Equality[T] описывает возможность сравнения значений типа T на равенство, причём метод NotEqual имеет *реализацию по умолчанию* (эта возможность поддерживается также в Scala, Ceylon, Rust, Swift, JavaGI, G и C#^{cpt}). Обобщённый класс HashSet<…> множества зависит не только от типа элементов T, но и от концепт-параметра Equality[T] eq; объединение (метод UnionWith) допустимо только со множеством, использующим такую же модель сравнения элементов, то есть со множеством типа HashSet<T!eq>. Заголовок метода GetUnion<…> ясно показывает, что s1 и s2 это множества с одинаковым типом элементов и общей моделью сравнения элементов eq.

На Рис. 8 представлены определения двух моделей концепта Equality[] для строк: eqString для регистрозависимого и eqCInsString для регистронезависимого сравнения. При инстанцировании класса множества нужно указать не только тип элементов, но и модель концепта Equality[] для этого типа. При создании инстанции класса с помощью оператора **new** HashSet<string>(…) концепт-параметр eq инициализируется моделью Equality[string] по умолчанию, если такая есть, в противном случае будет получена ошибка компиляции; модель

можно указать и явно, как при создании ss12 и ss2. Множества ss11 и ss12 используют одну и ту же модель Equality[string], eqString, поэтому к ним можно применить операцию объединения, в то время как ss11 и ss2 – разные, поэтому попытка получить их объединение приведёт к ошибке *компиляции*.

```
model default eqString for Equality[string]
{  bool Equal(string x, string y) { return x == y; }  }

model eqCInsString for Equality[string]
{  bool Equal(string x, string y) { return x.ToLower() == y.ToLower(); }}

var ss11 = new HashSet<string>(…);
var ss12 = new HashSet<string!eqString>(…);
var ss1  = GetUnion(ss11, ss12);  // OK

var ss2 = new HashSet<string!eqCInsString>(…);
var ss3 = GetUnion(ss11, ss2);     // Ошибка компиляции!
```

Рис. 8. Определение моделей и инстанцирование в C#^{CP}

Модели могут быть обобщёнными и зависеть от других моделей. На Рис. 9 представлено определение модели концепта Equality[] для типа обобщённой пары Pair<S, T>: пары одного типа можно сравнивать на равенство, если указаны модели сравнения для их элементов. Заметим, что так называемое условное определение моделей, то есть определение обобщённых моделей с требованиями к типовым параметрам, поддерживается в языках Rust, G, JavaGI и C#^{cpt}, а вот в Genus, где поддерживается множественное определение моделей, реализована полноценная зависимость моделей от моделей.

```
struct Pair<S, T> { public S first; public T second; … }

model eqPair<S, T ! Equality[S] eqS, Equality[T] eqT>
    for Equality[Pair<S, T>]
{
 bool Equal(Pair<S, T> a, Pair<S, T> b){
 { return eqS.Equal(a.first, b.first) && eqT.Equal(a.second, b.second);}
}
```

Рис. 9. Пример обобщённой модели в C#^{CP}

```
Genus:
interface List[E]{ boolean remove(E e) where Eq[E]; … }
List[string] ss = …;
ss.remove[with eqCInsString]("qwerty");

C#CP:
interface IList<T>{ bool Remove<!Equality[T] eq>(T x); … }
IList<string> ss = …;
ss.remove<!eqCInsString>("qwerty");
```

Рис. 10. Ограничения на типовые параметры в методах в Genus и C#^{CP}

Ещё одной интересной возможностью C#^{CP} является возможность накладывать ограничения на типовые параметры в методах класса, которая также реализована в языках Rust и Genus. В статье, посвящённой Genus [12], приводится пример обобщённого интерфейса списка с методом удаления элемента, для которого необходимо сравнение элементов списка на равенство. Соответствующий код приведён на Рис. 10: метод remove накладывает на тип элементов E ограничение в форме предиката, а при вызове метода в квадратных скобках можно указать конкретную модель сравнения элементов. В аналогичном примере на C#^{CP} (Рис. 10) концепт

сравнения является *явным параметром* метода Remove и заменяется на конкретную модель при инстанцировании.

Концепты C#^{CP} образуют пространства имён, в которые включены не только функции, но и имена типовых параметров концепта. В обобщённом коде не обязательно специфицировать все типовые параметры концепт-параметра. Соответствующий пример иллюстрирует Рис. 11: концепт Weighing[T, Weight] описывает возможность взвешивания значений типа T, где вес задаётся типом Weight; FindMaxWeight<…> – алгоритм поиска элемента массива с наибольшим весом, параметризованный типом элементов T и концептом взвешивания Weighing[T, ?] w, где вместо типа веса указан символ «?». Можно было бы сделать тип веса явным типовым параметром функции:

static int FindMaxWeight<T, Weight ! Weighing[T, Weight] w>(T[] values)

Как и на базовом C#, в котором тип веса указывать необходимо:

static int FindMaxWeight<T, Weight> **where** T : IWeighed<Weight> (T[] values)

Но тип веса не имеет значения в заголовке функции FindMaxWeight и лишь делает определение более громоздким. В реализации же функции к типу веса можно обратиться через концепт-параметр: w.Weight. Поскольку при инстанцировании кода вместо w будет подставлена конкретная модель, тип веса можно вывести. Рис. 11 также демонстрирует возможность *уточнения концептов*: концепт взвешивания включает все возможности концепта упорядочения для типа веса. При определении модели нужно указывать модель уточняемого концепта, как это сделано в моделях stringLetCntWeight (вес определяется как количество букв) и stringPnctCntWeight (вес – количество знаков пунктуации). В результате в переменную indMaxLetCnt будет записан индекс строки с наибольшим количеством букв, а в indMinPnctCnt – индекс строки с наименьшим количеством знаков пунктуации. Следует отметить, что возможность *не указывать* типовые параметры концептов в ограничениях является уникальной возможностью языка C#^{CP}. В C#^{cpt} и Genus, где для ограничения типовых параметров используются концепт-ограничения в форме предикатов, такая возможность не предусмотрена.

```
concept Ordering[T]
{  int Compare(T x, T y);
   bool Greater(T a, T b) { return this.Compare(x, y) > 0; } … }
concept Weighing[T, Weight] refining Ordering[Weight]
{  Weight GetWeight(T x);  }

static int FindMaxWeight<T ! Weighing[T, ?] w>(T[] values)
{  … // обработка null и пустого массива
   int indMax = 0;   w.Weight weightMax = w.GetWeight(values[0]);
   for (int i = 1; i < values.Length; ++i)
      if (w.Greater(w.GetWeight(values[i]), weightMax))
      { indMax = i;  weightMax = w.GetWeight(values[i]); }
   return indMax;
}

model default intAscOrd for Ordering[int] {…} // упорядочение int по возрастанию
model intDescOrd for Ordering[int] {…}       // упорядочение int по убыванию
model stringLetCntWeight refining intAscOrd for Weighing[string, int]
{  int GetWeight(string x) { return x.Count(Char.IsLetter); }  }
model stringPnctCntWeight refining intDescOrd for Weighing[string, int]
{  int GetWeight(string x) { return x.Count(Char.IsPunctuation); }  }

string[] ls = { "Oh, no", "Hello, world!", "cat" };
var indMaxLetCnt  = FindMaxWeight<string ! stringLetCntWeight>(ls);  // 1
var indMinPnctCnt = FindMaxWeight<string ! stringPnctCntWeight>(ls); // 2
```

Рис. 11. Ограничения на типовые параметры в методах в Genus и C#^{CP}

Заключение

Новые объектно-ориентированные языки программирования и расширения предлагают более выразительные инструменты обобщённого программирования по сравнению с

классическим F-ограниченным полиморфизмом на основе интерфейсов, реализованным в языках C# и Java. Мы выполнили сравнительный анализ средств ОП в современных языках и расширениях, и выделили несколько ключевых параметров, характеризующих эти инструменты: ретроактивное моделирование, использование конструкций-ограничений в роли типов, множественное определение моделей. Предложенный механизм обобщённого программирования на основе **концепт-параметров** не уступает по выразительной силе средствам ОП в других языках, но предполагает принципиально иной стиль обобщённого программирования: «ограничения как параметры» вместо «ограничения как предикаты на типы». Использование конструкций-ограничений как явных параметров обобщённого кода облегчает написание и восприятие кода, если разрешено множественное определение моделей, что, в свою очередь, является удобным элементом обобщённого программирования. Концепт-параметры также позволяют упростить «интерфейсы» обобщённых компонент, так как типы, которые могут быть выведены из моделей, можно не указывать в качестве параметров: таким образом, заголовки обобщённых компонент содержат только параметры, существенные для пользователей этих компонент.

Литература

1. David R. Musser, Alexander A. Stepanov. Generic Programming // Proceedings of the International Symposium ISSAC '88, p. 13–25. Springer-Verlag, London, UK, UK, 1989.
2. 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.
3. Andrew Kennedy, Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime // SIGPLAN Not., 36(5), May 2001, p. 1–12. ACM, New York, NY, USA.
4. 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 13th ACM SIGPLAN Conference OOPSLA '98, p. 183–200. ACM, New York, NY, USA, 1998.
5. Peter Canning, William Cook, Walter Hill, Walter Olthoff, John C. Mitchell. F-bounded Polymorphism for Object-oriented Programming // Proceedings of the Fourth International Conference FPCA '89, p. 273–280. ACM, New York, NY, USA, 1989.
6. 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.
7. Ben Greenman, Fabian Muehlboeck, Ross Tate. Getting F-bounded Polymorphism into Shape // Proce-edings of the 35th ACM SIGPLAN Conference PLDI '14, p. 89–99. ACM, New York, NY, USA, 2014.
8. 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.
9. Белякова Ю. В., Михалкович С. С. Средства обобщённого программирования в современных объектно-ориентированных языках. Часть 1. Анализ проблем. // Труды научной школы И. Б. Симоненко. Второй выпуск. Ростов н/Д: Изд-во ЮФУ, 2015. С. 60–74.
10. Jaakko Jarvi, Jeremiah Willcock, Andrew Lumsdaine. Associated Types and Constraint Propagation for Mainstream Object-oriented Generics. // Proceedings of the 20th Annual ACM SIGPLAN Conference OOPSLA '05, p. 1–19. ACM, New York, NY, USA, 2005.
11. 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.
12. Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, Andrew C. Myers. Lightweight, Flexible Object-oriented Generics // Proceedings of the 36th ACM SIGPLAN Conference PLDI '2015, p. 436–445. ACM, New York, NY, USA, 2015.
13. 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. 75–89.
14. A. Pelenitsyn. Associated types and constraint propagation for generic programming in Scala // Programming and Computer Software, 41(4), p. 224–230. Pleiades Publishing, 2015.
15. The Rust programming language, version 1.2.0. August 2015.
16. http://doc.rust-lang.org/stable/reference.html
17. The Swift Programming Language, version 2.0. August 2015.
18. http://developer.apple.com/swift/resources/
19. The Ceylon language specification, version 1.1.0. October 2014.
20. http://ceylon-lang.org/documentation/1.1/spec
21. Jeremy Siek, Andrew Lumsdaine. A Language for Generic Programming in the Large // Sci. Comput. Program., 76(5), p. 423–465, May 2011. Elsevier North-Holland, Inc., Amsterdam, The Netherlands.