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 2. ОБЗОР НОВЫХ РЕШЕНИЙ

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

Generic programming (GP) is supported in programming languages in different ways. A review of the mechanisms for GP in modern object-oriented languages Scala, Ceylon, Rust, Swift, and G is presented in this paper; several language extensions improving the level of support for GP in C# and Java are also explored. We extend a list of language properties facilitating generic programming, which has been initially introduced by Garcia et al. (2003), and compare languages and extensions considered against these properties.

1 Введение

Средства поддержки обобщённого программирования (ОП) в различных языках существенно отличаются [1]. Наиболее выразительными средствами обладает функциональный язык Haskell с его классами типов (type classes), в то время как промышленные объектно-ориентированные языки C# и Java, реализующие так называемый *F-ограниченный полиморфизм* через ограничения подтипирования и интерфейсы, предоставляют довольно бедные инструменты. Недостатки средств ОП в этих языках подробно проанализированы в первой части данной статьи [2].

В этой части статьи мы рассмотрим несколько расширений C# и Java с улучшенными средствами обобщённого программирования, а также средства ОП в достаточно новых ОО-языках Scala, G, Ceylon, Rust и Swift. Мы не касаемся способов и эффективности реализации этих средств в компиляторах[5], основной целью данной работы является сравнительный анализ их *выразительных возможностей*. Следующие разделы посвящены обзору средств обобщённого программирования в выбранных языках и расширениях. При описании соответствующих возможностей используется терминология, введённая в первой части [2]: ретроактивное моделирование, ассоциированные типы, мультипараметрические ограничения, концепт-паттерн, согласованность моделей. Мы также будем использовать понятия динамической диспетчеризации и множественной динамической диспетчеризации. Пусть определены функции `foo(A a)` и `bar(C c,`

[5] Для рассматриваемых в статье языков программирования Scala, Ceylon, Rust и Swift существуют компиляторы, сложнее обстоит дело с проектами расширений: компилятор JavaGI [10] существует и находится в свободном доступе; компиляторы языка G [7] и Genus [11] по всей видимости существуют, но авторам не удалось их найти; для проекта языка C# с концептами [9] компилятора нет.

```
shared interface Comparable<Other> of Other
        given Other satisfies Comparable<Other>  {
    shared formal Integer compareTo(Other other);
    shared Integer reverseCompareTo(Other other) {
        return other.compareTo(this);
}   }
```

Рис. 1. Интерфейс `Comparable<Other>` в Ceylon, использующий собственный тип

`D d`). Если для различных потомков $\mathtt{A}_i$ класса `A`, переопределяющих реализацию `foo`, вызов `foo(`$\mathtt{a}_i$`)`, где `A` $\mathtt{a}_i$ `= new` $\mathtt{A}_i$`()`, обеспечивает различное поведение на этапе выполнения, то говорят, что функция `foo` реализует динамическую диспетчеризацию (различное поведение в зависимости от типа-наследника). Аналогично, если для различных пар потомков ($\mathtt{C}_i$, $\mathtt{D}_i$) классов (`C`, `D`), переопределяющих реализацию `bar`, вызов `bar(`$\mathtt{c}_i$`,` $\mathtt{d}_i$`)`, где `C` $\mathtt{c}_i$ `= new` $\mathtt{C}_i$`()`, `D` $\mathtt{d}_i$ `= new` $\mathtt{D}_i$`()`, обеспечивает различное поведение на этапе выполнения, говорят, что функция `bar` реализует двойную динамическую диспетчеризацию. Множественная динамическая диспетчеризация (multiple dynamic dispatch) вводится аналогично для N параметров.

2 Модифицированные интерфейсы

Одним из наиболее очевидных способов улучшения средств обобщённого программирования в ОО-языке является изменение нотации интерфейсов. Модификации могут быть небольшими, как в расширении языка C# ассоциированными типами и распространением ограничений [8], рассмотренном в первой части данной статьи [2]; то же касается и интерфейсов в языке Ceylon [16] (Разд. 2.1): сохраняются проблемы ретроактивного моделирования, статических методов и мультипараметрических интерфейсов. Эти проблемы решаются в JavaGI [10] (Разд. 2.2) – расширении языка Java с генерализованными интерфейсами.

2.1 Интерфейсы в Ceylon

Интерфейсы в языке Ceylon [16], в отличие от «классических» интерфейсов, допускают реализацию методов и позволяют выражать собственные типы (self types). Примером использования собственного типа может служить интерфейс сравнения `Comparable<Other>`[6] (Рис. 1): декларация `of Other` указывает, что именно тип `Other` реализует интерфейс

[6] В методе `reverseCompareTo` объект `this` используется в качестве аргумента `compareTo`, который ожидает объект типа `Other`; без декларации `of Other` типом `this` оказывается `Comparable<Other>`, а он не является подтипом `Other`, поэтому вызов `other.compareTo(this)` не откомпилируется.

```
interface EQ { boolean eq(This that); }
boolean contains<X>(List<X> list, X x) where X implements EQ {...}

abstract class Expr { ... }      class IntLit extends Expr { ... }
class PlusExpr extends Expr { Expr left; Expr right; ... }
implementation EQ [Expr] { boolean eq(Expr that) {return false;} }
implementation EQ [PlusExpr] { boolean eq(PlusExpr that) { ... } }

interface ObserverPattern [Subject, Observer]
{  receiver Subject  {  void register(Observer o);
                        void notifyObservers();  }
   receiver Observer {  void update(Subject s);  } }
```

Рис. 2. Примеры генерализованных интерфейсов в JavaGI

`Comparable<Other>` (то есть является его собственным типом), поэтому вызов `other.compareTo(this)` в методе `reverseCompareTo` корректен. Интересной особенностью Ceylon является разный синтаксис для ограничений подтипирования и ограничений-интерфейсов. Для последних используется конструкция `given X satisfies Y`. В остальном интерфейсы Ceylon страдают от тех же проблем, что и интерфейсы C#/Java.

2.2 Генерализованные интерфейсы JavaGI

Генерализованные интерфейсы JavaGI [10] (generalized interfaces) во многом похожи на классы типов: они обеспечивают возможность ретроактивного моделирования, решают проблему бинарных методов благодаря наличию собственных типов, поддерживают мультипараметрические ограничения и статические методы. Генерализованные интерфейсы также частично поддерживают множественную динамическую диспетчеризацию.

На Рис. 2 представлен пример интерфейса `EQ` с бинарным методом `eq`: ключевое слово `This` явно указывает собственный тип интерфейса. Пример мультипараметрического интерфейса – `ObserverPattern`, интерфейс паттерна Наблюдатель [10]. В мультипараметрических интерфейсах отсутствует собственный тип `This`: в отличие от интерфейсов для одного типа, они не могут использоваться в роли типов. Реализации интерфейса (например, `EQ` для `Expr` и `PlusExpr`) вводятся с помощью конструкции `implementation`. Реализация интерфейса также может быть обобщённой и содержать ограничения на типовый параметр, как в следующем примере:

```
implementation<X> EQ [List<X>] where X implements EQ { ... }
```

В JavaGI есть возможность определения абстрактной реализации интерфейса (abstract implementation definition) [10], которая позволяет смодели-

```
interface RichExpr { int depth(); List<RichExpr> subExprs(); }
abstract implementation RichExpr [RichExpr] {
  int depth() { ... for (RichExpr e : subExprs())
            { i = Math.max(i, e.depth()); } ...
} }
implementation RichExpr [Expr] extends RichExpr [RichExpr]
{ List<RichExpr> subExprs() { return new LinkedList<RichExpr>(); }}

interface ReadOnlyList<X>
{   int size();    boolean isEmpty();    X elementAt(int i);   }
abstract implementation<X> ReadOnlyList<X> [ReadOnlyList<X>]
{   boolean isEmpty() { return this.size() == 0; }   }
```

Рис. 3. Абстрактная реализация и обобщённый интерфейс в JavaGI

ровать реализацию методов интерфейса по умолчанию: на Рис. 3 таким образом реализован метод `depth` интерфейса `RichExpr`. Реализации интерфейсов можно наследовать (`RichExpr [Expr]`, Рис. 3).

Наиболее существенной проблемой JavaGI можно назвать невозможность определения нескольких реализаций интерфейса для одного типа (то же самое касается классов типов в Haskell). Механизм абстрактных реализаций имеет довольно громоздкий синтаксис; восприятие кода ещё больше усложняется в случае обобщённых интерфейсов, так как возникают типовые параметры двух видов – типы, реализующие интерфейс, и параметры интерфейса. На Рис. 3 представлен пример обобщённого интерфейса списка `ReadOnlyList<X>` и абстрактной реализации для него.

3 Новые языки программирования: трейты и протоколы вместо интерфейсов

В новых ОО-языках программирования на смену классическим интерфейсам пришли альтернативные конструкции, которые, как и интерфейсы C#/Java, используются в двух ролях: (1) как ограничения на типовые параметры; (2) как типы. В языках Scala [13] (Разд. 3.1) и Rust [17] (Разд. 3.2) реализованы трейты (traits), которые позволяют определять не только сигнатуры методов, но и их реализацию. В языке Swift [18] (Разд. 3.3) вместо интерфейсов используются протоколы (protocols).

3.1 Трейты и имплиситы в Scala

Трейты в Scala не обеспечивают ретроактивное моделирование, но поддерживают реализацию методов и абстрактные типы, которые можно

```
trait Printable { def print(): String; }
def foo[A <: Printable](x: A): Unit { x.print(); }

trait Ordering[A] { abstract def compare(x: T, y: T): Int
  def equiv(x: T, y: T): Boolean = this.compare(x, y) == 0
}
(s-1) def sort[T : Ordering](values: List[T]) { ... }
(s-2) def sort[T](values: List[T])(implicit ord: Ordering[T]) {...}
```

Рис. 4. Примеры обобщённого кода в Scala

считать аналогом ассоциированных типов [14]. Классы, трейты и объекты могут быть быть обобщёнными: типовые параметры поддерживают ограничения подтипирования и надтипирования, при этом в качестве верхних границ могут быть указаны и трейты. В Scala активно используется концепт-паттерн [12], недостатки которого подробно обсуждались в [2]. Использование концепт-паттерна облегчается благодаря имплиситам (implicits) – аргументам функций и конструкторов по умолчанию, а также специальному синтаксическому сахару, который облегчает использование концепт-паттерна в случае ограничений на один типовый параметр. На Рис. 4 представлены некоторые примеры. В методе `foo[A]` трейт `Printable` используется для ограничения типового параметра `A` как классический интерфейс, без возможности ретроактивного моделирования. А вот трейт `Ordering[A]` подобен интерфейсу `IComparer<T>` в C# [2] – он используется в рамках концепт-паттерна. Версия `(s-1)` алгоритма сортировки – это *синтаксический сахар* для версии `(s-2)` с имплисит-параметром `ord`. Ограничение `T : Ordering` называется контекстной границей (context bound) и переводится в имплисит-параметр `ord : Ordering[T]` в `(s-2)`.

3.2 Трейты в Rust

В языке Rust [17] нет привычного понятия класса, вместо классов используются структуры (structs), которые хранят только данные, и реализации методов для этих структур (method implementations). Структуры и методы могут быть обобщёнными. Ещё одна конструкция языка – трейт (trait) – описывает функциональность типа. Трейты прежде всего используются в роли ограничений на типовые параметры обобщённого кода. На Рис. 5 представлен код трейта `Equatable`: тип `Self` это собственный тип трейта; метод `not_equal` имеет реализацию по умолчанию; символ `&` свидетельствует о передаче параметров по ссылке. Трейты могут быть обобщёнными, включать ограничения на типовые параметры всего трейта или отдельных функций; трейты могут содержать статические методы

```
trait Equatable
{  fn equal(&self, that: &Self) -> bool;
   fn not_equal(&self, that: &Self) -> bool { !self.equal(that) } }
trait Printable { fn print(&self); }

impl Equatable for i32
{ fn equal(&self, that: &i32) -> bool { *self == *that } }

struct Pair<S, T>{ fst: S, snd: T }
impl <S : Equatable, T : Equatable> Equatable for Pair<S, T>
{ fn equal(&self, that: &Pair<S, T>) -> bool {
     self.fst.equal(&that.fst) && self.snd.equal(&that.snd) } }

fn contains<T>(xs: &Vec<T>, x: &T) -> bool where T : Equatable {..}
```

Рис. 5. Примеры использования трейтов в Rust

и ассоциированные типы. Трейт может наследовать другие трейты.

Реализация трейта расширяет функциональность типа и является ретроактивной; трейт можно реализовать для любых типов, включая примитивные. На Рис. 5 трейт `Equatable` реализован для типов `i32` (целое число) и обобщённой пары `Pair<S, T>`, где на типовые параметры реализации наложены ограничения-трейты (это называют обобщённой условной реализацией): пары можно сравнивать на равенство, только если их элементы можно сравнивать на равенство. Ограничения можно указывать и в секции `where`, как в функции `contains<T>`.

Трейты можно использовать не только как ограничения, но и как типы, таким образом обеспечивается динамическая диспетчеризация на основе полиморфизма через трейты (классического полиморфизма через наследование классов в Rust нет). Например, трейт `Printable` (Рис. 5), реализованный для типов `i32` и `f64`, может быть использован как тип элементов полиморфной коллекции (нужно использовать ссылки, `&`):

```
let pr1 = 3; let pr2 = 4.5; let pr3 = -10;
let polyVec: Vec<&Printable> = vec![&pr1, &pr2, &pr3];
for v in polyVec { v.print(); }
```

Основным недостатком механизма трейтов в Rust является невозможность определения нескольких реализаций трейта для типа. Кроме того, поскольку трейт описывает функциональность одного типа (подобно интерфейсам), ограничения на несколько типов можно выразить только с помощью нескольких связанных трейтов:

```
trait Observer<S> where S : Subject<Self>  { ... }
trait Subject<O>  where O : Observer<Self> { ... }
```

```
protocol Equatable { func equal(that: Self) -> Bool; }
extension Equatable
{ func notEqual(that: Self) -> Bool { return !self.equal(that) } }

protocol Printable { func print(); }
extension Int : Printable {...}  extension Double : Printable {...}

func contains<T : Equatable>(values: [T], x: T) -> Bool { ... }
class Pair<S, T> { var first: S; var second: T; ... }
```

Рис. 6. Примеры обобщённого кода и протоколов в Swift

3.3 Протоколы в Swift

В языке Swift [18] есть привычные классы, наследование и полиморфизм, а вместо интерфейсов используются протоколы (protocols), которые, подобно трейтам Rust, в первую очередь используются в роли ограничений. Протоколы, подобно интерфейсам и трейтам, описывают функциональность типа, реализующего протокол. Протоколы не могут быть обобщёнными, но могут содержать ассоциированные типы и ограничения на эти типы, а также статические методы. У протокола есть собственный тип `Self`; протоколы могут наследовать другие протоколы.

Функциональность классов Swift можно расширять (extend) ретроактивно: расширение может просто добавлять функциональность к классу или устанавливать отношение реализации протокола. Расширять можно не только классы, но и протоколы: расширение добавляет к протоколу новые методы с реализацией (при определении протокола указываются лишь заголовки методов). На Рис. 6 представлено несколько примеров: протокол `Equatable` с методом `equal`, а также его расширение, добавляющее метод `notEqual`; протокол `Printable` и его реализация для примитивных типов; обобщённая функция `contains<T>` с ограничением на тип `T`; класс обобщённой пары `Pair<S, T>`. Как и в Rust, протокол можно использовать в качестве типа элементов полиморфной коллекции:

```
var polyArr: [Printable] = [3, 45.5, -1];
for v in polyArr { v.print() }
```

Основные недостатки средств ОП в Swift те же, что и в Rust: единственность способа реализации протокола, проблема мультипараметрических ограничений. Кроме того, расширять обобщённый тип можно только в целом, отдельные специализации как в Rust не допускаются:

```
extension Pair : Equatable          // ошибка компиляции
where S : Equatable, T : Equatable { ... }
```

```
concept Monoid<T> {   fun identity_elt()  -> T;
                      fun binary_op(T, T) -> T;  };
fun accumulate<Iter>
where { InputIterator<Iter>, Monoid<InputIterator<Iter>.value> }
(Iter first, Iter last) -> InputIterator<Iter>.value {
    let init = identity_elt(); ...
}
model Monoid<int> {   fun identity_elt() -> int@ { return 0; }  };
```

Рис. 7. Пример концепта и его использования в языке G

4 Конструкции-ограничения как объекты второго класса

Все механизмы, рассмотренные в предыдущих разделах, объединяет одна важная черта: конструкции, которые используются для *ограничения* типовых параметров обобщённого кода, одновременно могут выступать и в роли *типов*. В работе [15] авторы, исследовав почти 14 млн. строк Java-кода, демонстрируют, что на практике интерфейсы, используемые в качестве ограничений (как `Comparable<X>` в Java), никогда не используются в роли типов. Это свойство называется разделением материи и формы (material-shape separation): интерфейс-ограничение представляет собой форму, а интерфейс-тип – материю. Аналогичное разделение было замечено и в коде на языке Ceylon [16]. В данном разделе мы рассмотрим ещё три механизма обобщённого программирования: концепты в языке G [7], проект концептов для языка C# [9] и проект Genus [11] расширения языка Java. В этих расширениях, как и в классах типов Haskell, для описания ограничений на типовые параметры обобщённого кода используются конструкции языка, *не имеющие собственного типа*: эти конструкции являются *объектами второго класса*, они не хранятся в обычных переменных и не могут быть использованы в роли типов.

4.1 Концепты C++ и G

Концепты (concepts) – это новая конструкция языка, которая должна решить проблемы неограниченных шаблонов C++, обсуждавшиеся в первой части данной статьи [2]. Работа над механизмом концептов началась около 2000-го года, с тех пор было представлено несколько дизайнов [3, 7] и ожидалось, что концепты войдут в стандарт C++0x. Сейчас разрабатывается новая версия – концепты C++1z (Concepts Lite) [5], анализу которых посвящена статья [6] из этого сборника. В данном разделе мы рассмотрим концепты на примере языка G [7] (подмножество языка

C++ с концептами, подобными концептам C++0x), который декларируется авторами как «язык для обобщённого программирования». Заметим, что по выразительной силе концепты очень близки классам типов Haskell [4].

Концепт может содержать сигнатуры функций, реализации функций по умолчанию, ассоциированные типы, концепт-требования к ассоциированным типам, требования равенства типов. Концепт не имеет собственного типа, он описывает внешние требования к типовому параметру (или параметрам). На Рис. 7 представлен концепт Моноид одного типового параметра `T` (`Monoid<T>`) и обобщённый алгоритм `accumulate<Iter>` с требованиями к типу итератора `Iter` и его ассоциированному типу `value` [7]. Концепт-требования можно считать предикатами на типах.

Говорят, что тип удовлетворяет концепту, если определена (ретроактивно) соответствующая модель (model) концепта для этого типа. Как и в JavaGI, можно определять обобщённые условные модели, то есть модели с ограничениями на типовые параметры. Концепт может уточнять (refine) другой концепт: уточняющий концепт включает все требования уточняемого (это похоже на наследование). Уточнение концептов обеспечивает уникальную для C++/G возможность перегрузки на основе концептов, которая позволяет определить несколько версий одного алгоритма/класса, отличающиеся ограничениями на параметры шаблона[7].

Как и в JavaGI, в G нельзя определить несколько моделей концепта для одного набора типов. Кроме того, механизм наследования никак не взаимодействует с механизмом концептов: если определена модель концепта `C<A>`, она не может быть использована для объектов типа `B`, где класс `B` наследует `A`, – необходимо специально определить модель `C<B>`.

4.2 Язык C# с концептами

В проекте языка C#$^{\text{cpt}}$ (C# с концептами), разработанном нами [9], механизм концептов G/C++, рассмотренный в Разд. 4.1, интегрируется с ограничениями подтипирования: на типовые параметры и ассоциированные типы можно накладывать ограничения подтипирования и надтипирования; возможно определение нескольких моделей, автоматическая генерация модели по умолчанию, а также определение анонимных моделей; возможно наследование моделей; для облегчения доступа к элементам концептов, концепт-требования можно именовать. Концепты можно уточнять, но перегрузка на основе концептов не поддерживается.

[7]Классическим примером перегрузки на основе концептов является алгоритм `advance` продвижения итератора.

```
concept CEquatable[T] { bool Equal(T x, T y);
    bool NotEqual(T x, T y) { return !Equal(x, y); } }
concept CObserverPattern[O, S] { void UpdateSubject(O obs, S subj);
                                 void NotifyObservers(S subj);... }
bool Contains<T>(T[] values, T x)
    where CEquatable[T] using CEq  { ...if (cEq.Equal(...))... }
class MySet<T> where CEquatible[T} { ... }

model default StringEqCaseS for CEquatable[String] { ... }
model StringEqCaseIS for CEquatable[String] { ... }
```

Рис. 8. Примеры обобщённого кода в C# с концептами

На Рис. 8 представлено несколько примеров. В определении концепта `CEquatable[T]` используется реализация по умолчанию для метода `NotEqual`; `CObserverPattern[O, S]` – пример мультипараметрического концепта. В обобщённой функции `Contains<T>` используется концепт-требование с алиасом `cEq`, который «через точку» позволяет обратиться к элементам концепта. Модели `StringEqCaseS` и `StringEqCaseIS` реализуют регистрозависимое и регистронезависимое сравнение строк соответственно; первая модель с помощью `default` помечена как модель по умолчанию: если при инстанцировании модель не указана явно, будет использована модель по умолчанию. Согласованность моделей проверяется на этапе компиляции. Например, если множества строк `s1`, `s2` определены следующим образом:

```
var s1 = new MySet<String>(...);
var s2 = new MySet<String>[using StringEqCaseIS](...);
```

Операция `s1.UnionWith(s2)` недопустима, так как `s1` и `s2` используют разные модели концепта.

Концепты C# не поддерживают динамическую диспетчеризацию, к недостаткам также можно отнести довольно громоздкий синтаксис использования именованных моделей. Ограничения на типовые параметры обобщённого класса можно накладывать только на верхнем уровне, дополнительные ограничения в методах невозможны.

4.3 Ограничения в Genus

Ограничения (constraints) в Genus [11] напоминают концепты, они играют роль предикатов на типах. Ограничения могут содержать сигнатуры методов и статические методы; ограничения могут наследоваться. На Рис. 9 представлен пример простого ограничения `Eq[T]` и мультипараметрического ограничения `GraphLike[V, E]`, связывающего типы вершины и ребра. В методах обобщённого класса/интерфейса разрешается накла-

```
constraint Eq[T] { boolean T.equals(T other); }
constraint GraphLike[V,E] {... Iterable[E] V.incomingEdges();
                               V E.source();  ...              }
class TreeSet[T where Comparable[T]] { ... }

model CIEq for Eq[String] {  bool equals(String str)
                            { return equalsIgnoreCase(str); }  }
model DualGraph[V,E] for GraphLike[V,E] where GraphLike[V,E] g {..}
```

Рис. 9. Примеры обобщённого кода в Genus

дывать дополнительные ограничения на типовые параметры, например:

```
interface List[E] { boolean remove(E e) where Eq[E]; ... }
```

Как и в концептах C#, можно определять несколько именованных моделей ограничения для одного набора типов. Если типы содержат операции, требуемые ограничением, для них автоматически генерируется естественная модель (в концептах C# она называлась моделью по умолчанию). Genus также обеспечивает проверку согласованности моделей: модель считается частью типа, поэтому типы называют типами, зависящими от моделей (model-dependent types). Например, в следующем примере будет получена ошибка компиляции:

```
Set[String] s0 = ...;            Set[String with CIEq] s1 = ...;
s1 = s0; // ошибка компиляции
```

Модели могут быть обобщёнными, содержать ограничения и *зависеть от других моделей*, как в примере модели `DualGraph[V,E]` на Рис. 9). Модели можно наследовать, а также ретроактивно расширять, таким образом обеспечивается множественная динамическая диспетчеризация.

Ограничения в Genus не могут содержать ассоциированные типы и реализацию методов по умолчанию. Также не поддерживаются ограничения подтипирования/надтипирования на типовые параметры.

5 Сравнительный анализ дизайна средств ОП

В предыдущих разделах был выполнен обзор механизмов обобщённого программирования в современных объектно-ориентированных языках и расширениях C# и Java с улучшенными средствами ОП, в том числе проекте концептов для C#, разработанном нами [9]. Табл. 1 представляет *сравнение выразительных возможностей* рассмотренных механизмов[8], а также классов типов Haskell. В первой колонке перечислены возможности (характеристики) этих средств, в колонках языков – уровень их

[8]Колонка для Java полностью аналогична C#, поэтому не представлена отдельно; C#$^{\text{ext}}$ соответствует C# с ассоциированными типами [8]; C#$^{\text{cpt}}$ соответствует C# с концептами [9] (Разд. 4.2).

	Haskell	C#	Scala	C#$^{\text{ext}}$	Ceylon	Rust	Swift	JavaGI	G	C#$^{\text{cpt}}$	Genus
Мультипараметрические ограничения	●	⚹	⚹	○	○	○	○	●	●	●	●
Использование ограничений как типов	○	●	●	●	●	●	●	◐	○	○	○
Явные собственные типы	–	○	◐	○	●$^{-}$	●	●	◐	–	–	–
Динамическая диспетчеризация	–	●	●	●	●	●	●	●	○	○	●
Множеств. динамич. диспетчеризация	–	○	○	○	○	○	○	◐	○	○	●
Расширение моделей	○	○	○	○	○	○	○	○	○	○	●
Одиночное наследование ограничений	○	○	○	○	○	○	○	○	○	●	○
Множественное уточнение ограничений	●	●	●	●	●	●	●	●	●	○	●
Реализация методов по умолчанию	●	○	●	○	●	●	●	◐	●	●	○
Статические методы	●^{a}	○	○	○	●	●	●	●	●^{a}	●^{a}	●^{a}
Ассоциированные типы	●	○	●	●	○	●	●	○	●	●	○
Ограничения на ассоциированные типы	◐	–	●	●	–	●	●	–	●	●	–
Ограничения равенства типов	◐	–	●	●	–	●	●	–	●	●	–
Ограничения подтипирования	–	●	●	●	●	–	●	○	○	●	○
Ограничения надтипирования	–	○	●	○	○	–	○	○	○	●	○
Другой синтаксис для огранич. подтип.	–	○	○	○	●	–	○	–	–	●	–
Множественные ограничения на типы	●	●	●	●	●	●	●	●	●	●	●
Ретроактивное моделирование	●	⚹	⚹	○	○	●	●	●	●	●	●
Применимость моделей к подтипам	–	●	●	●	●	–	●	●	○	●	●
Наследование моделей	○	_^{b}	_^{b}	_^{b}	_^{b}	○	○	●	○	●	●
Обобщённые условные модели	●	⚹	⚹	–	–	●	○	●	●	●	●
Множественное определение моделей	○	⚹	⚹	○	○	○	○	○	○	●	●
Типы, зависящие от моделей	_^{c}	○	○	_^{c}	_^{c}	_^{c}	_^{c}	_^{c}	_^{c}	●	●
Обобщ. модели, зависящие от моделей	_^{d}	_^{d}	_^{d}	_^{d}	_^{d}	_^{d}	_^{d}	_^{d}	_^{d}	○	●
Ограничения на типы в методах	–	○	○	○	○	●	○	○	○	○	●
Перегрузка на основе ограничений	○	○	⚹	○	○	○	●	◐^{e}	○	○	○

[a]Поскольку у ограничений нет собственных типов, роль статических методов играют просто внешние функции.
[b]Поскольку ретроактивное моделирование не поддерживается, для определения моделей нет отдельных конструкций, поэтому и наследование «моделей» принципиально невозможно.
[c]Когда модель определяется единственным образом, нет смысла говорить о типах, зависящих от моделей.
[d]Когда модель определяется единственным образом, нет смысла говорить о моделях, зависящих от моделей.
[e]Концепты C++0x, в отличие от G, обеспечивают полную поддержку (●) перегрузки на основе ограничений.

Таблица 1. Поддержка обобщённого программирования в ОО-языках

поддержки в соответствующем языке. Символ ● означает полноценную поддержку, ◐ – частичную, ○ – возможность не поддерживается, ⚹ – поддержка симулируется концепт-паттерном, «–» – характеристика не применима к данному языку.

Характеристики обобщённых средств в таблице сгруппированы в связанные группы (по строкам) и разделены горизонтальной чертой. Приведём некоторые разъяснения. Возможности «мультипараметрические ограничения» и «использование ограничений как типов» практически взаимоисключают друг друга: использование конструкции-ограничения в качестве типа означает, что у этой конструкции есть собственный тип, реализующий соответствующее ограничение, но тогда ограничения на

несколько параметров выразить естественным образом невозможно. Исключение составляют генерализованные интерфейсы JavaGI (Разд. 2.2), в которых интерфейсы для одного типа могут использоваться как типы, а мультипараметрические интерфейсы – нет. Благодаря наличию собственного типа, за счёт полиморфизма на основе подтипирования, в конструкциях «ограничениях-как-типах» обеспечивается динамическая диспетчеризация (ДД). Чтобы реализовать ДД в конструкциях без собственных типов, нужны дополнительные средства. Полноценная поддержка множественной динамической диспетчеризации (МДД) есть только в Genus: она реализуется через механизм расширения моделей. Возможность «множественное определение моделей», реализованная с помощью концепт-паттерна, вызывает проблему несогласованности моделей на этапе выполнения [2], поэтому в строке «типы, зависящие от моделей» для C# и Scala указано ○. Во всех остальных языках, кроме C#$^{\text{cpt}}$ и Genus, модель определяется единственным образом.

Отметим, что возможность ретроактивного моделирования несомненно является одной из важнейших характеристик средств обобщённого программирования, независимо от того, используются ли конструкции-ограничения в роли типов (C#/Java/Scala/Ceylon/Rust/Swift), или нет (G/C#$^{\text{cpt}}$/Genus). Использование концепт-паттерна для симуляции ретроактивного моделирования позволяет симулировать и множественное определение моделей (МОМ), но ведёт к серьёзной проблеме их несогласованности. Лишь два проекта – C#$^{\text{cpt}}$ и Genus – поддерживают МОМ на уровне языка и в этом аспекте имеют существенное преимущество перед остальными языками. Ещё одним достоинством Genus является поддержка МДД: в остальных языках с конструкциями-ограничениями как объектами второго класса (G и C#$^{\text{cpt}}$), динамическая диспетчеризация не поддерживается вовсе. Ассоциированные типы поддерживаются в большинстве современных языков и расширений, за исключением Genus и JavaGI. Вопрос о пользе ограничений подтипирования и надтипирования, которые поддерживаются не более, чем половиной рассмотренных механизмов, требует дополнительного исследования наряду с вопросом о поддержке ковариантности и контрвариантности типовых параметров.

6 Заключение

Первое известное нам исследование, посвящённое сравнению средств обобщённого программирования, было выполнено в 2003 году [1] для

шести языков программирования; в 2007 оно было расширено до восьми языков, включая C#, Java и Haskell. В результате были выделены свойства языков программирования, существенно влияющие на удобство обобщённого программирования, и составлена сравнительная таблица, которая позже расширялась и дополнялась данными для новых языков и расширений [7, 11, 12]. В данной статье представлен обзор и сравнительный анализ исключительно *объектно-ориентированных* языков программирования и их расширений: были выбраны современные языки и актуальные проекты, опубликованные вплоть до середины 2015 года. Мы расширили список характеристик инструментов обобщённого программирования и выделили особенности, характерные для ОО-языков. Полученная в результате таблица и выводы представлены в Разд. 5. Как показывает анализ, наблюдается «борьба» двух принципиально разных подходов к проектированию средств обобщённого программирования: (1) интеграция F-ограниченного полиморфизма с новыми конструкциями языка; (2) замена F-ограниченного на неограниченный параметрический полиморфизм при поддержке новых конструкций языка.

В заключение отметим, что разнообразие рассмотренных выше обобщённых средств, возникшее в последние годы, свидетельствует о том, что разработка сбалансированного по сложности и выразительности механизма ОП для ОО-языка является актуальной задачей.

ЛИТЕРАТУРА

[1] *Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, Jeremiah Willcock.* A Comparative Study of Language Support for Generic Programming // Proceedings of the OOPSLA '03, 2003, p. 115–134. ACM, New York, NY, USA.

[2] *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. 63–77.

[3] *Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine.* Concepts: Linguistic Support for Generic Programming in C++ // Proceedings of the OOPSLA '06, p. 291–310. ACM, New York, NY, USA, 2006.

[4] *Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, Andreas Priesnitz.* A Comparison of C++ Concepts and Haskell Type Classes // Proceedings of the WGP '08, p. 37–48. ACM, New York, NY, USA, 2008.

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

[6] *Пеленицын А. М.* Поддержка обобщённого программирования в новом проекте концептов для C++. // Труды научной школы И. Б. Симоненко. Второй выпуск. Ростов н/Д: Изд-во ЮФУ, 2015. С. 255–269.

[7] *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.

[8] *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.

[9] *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.

[10] *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.

[11] *Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, Andrew C. Myers.* Lightweight, Flexible Object-oriented Generics // Proceedings of the PLDI '2015, p. 436–445. ACM, New York, NY, USA, 2015.

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

[13] *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.

[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] *Ben Greenman, Fabian Muehlboeck, Ross Tate.* Getting F-bounded Polymorphism into Shape // Proceedings of the PLDI '14, p. 89–99. ACM, New York, NY, USA, 2014.

[16] The Ceylon language specification, version 1.1.0. October 2014. `http://ceylon-lang.org/documentation/1.1/spec`

[17] The Rust programming language, version 1.2.0. August 2015. `http://doc.rust-lang.org/stable/reference.html`

[18] The Swift Programming Language, version 2.0. August 2015. `http://developer.apple.com/swift/resources/`

Содержание

От редакторов 3

Симоненко Р. А. Обстоятельства, из которых вырос математик (воспоминания об Игоре Борисовиче Симоненко) 4

Абрамян М. Э. Использование задачника Programming Taskbook в качестве платформы для разработки специализированных электронных задачников по программированию . . . 11

Абу-Халил Ж. М., Гуда С. А., Штейнберг Б. Я. О высокопроизводительной переносимости программ 26

Алымова Е. В., Колесникова К. М. Автоматизация построения полного по критерию набора тестов для интерфейса web-ориентированного автоматического распараллеливателя программ 34

Алымова Е. В., Кочерга М. А. Реализация задания и проверки условий применимости преобразований в оптимизирующей распараллеливающей системе 47

Бабич П. В., Левенштам В. Б. Обратная задача для уравнения теплопроводности с высокочастотным источником 57

Белякова Ю. В., Михалкович С. С.Средства обобщённого программирования в современных объектно-ориентированных языках. Часть 1. Анализ проблем 63

Белякова Ю. В., Михалкович С. С.Средства обобщённого программирования в современных объектно-ориентированных языках. Часть 2. Обзор новых решений 78

Деундяк В. М. О вычислении индекса операторов свертки в гильбертовых модулях на абелевых группах 93

Деундяк В. М., Евпак С. А. Уязвимости полилинейной системы распределения ключей в случае превышения порога мощности коалиции злоумышленников 105

Деундяк В. М., Жданова М. А., Могилевская Н. С. Об автоматизированном выборе модели потока ошибок в информационной системе оценки применимости помехоустойчивого кодирования 116

Дыбин В. Б., Ермаков В. С. Об одном разностном уравнении в весовых пространствах 126

Евпак С. А. Выбор параметров для безопасного функционирования схем специального широковещательного шифрования на q-ичных кодах Рида-Маллера 136
Евдокимова А. Ю., Кряквин В. Д. Обнаружение резких неоднородностей изображений с использованием дискретного преобразования всплесков 144
Ерусалимский Я. М. Символьный язык линейной алгебры . . . 149
Козак А. В., Штейнберг Б. Я., Штейнберг О. Б. Уравнение дискретной свертки с характеристической функцией сегмента и его приложение 157
Коненко А. С. Оптимизация работы анализа псевдонимов в ОРС 168
Косолапов Ю. В. К вопросу о вычислении средней мощности множества претендентов при случайном наблюдении . . . 179
Лернер Е. В., Мкртичян В. В. Достаточное условие существования конечной проективной плоскости 187
Лукин А. В. О свойствах квазиэквивалентности в абстрактной версии локального метода 202
Малеванный М. С., Михалкович С. С. Модель поиска точек привязки для аспектной разметки кода 216
Мелихов С. Н., Стефаненко Л. В. Об операторе решения для дифференциального уравнения бесконечного порядка в пространствах аналитических функций 230
Овчинникова С. Н. Резонансные режимы в окрестности точки бифуркации коразмерности 2 (Res 4) в задаче Куэтта-Тейлора 238
Пасенчук А. Э. Об индексе матричного оператора Римана в пространстве гладких вектор-функций 247
Пеленицын А. М. Анализ поддержки обобщённого программирования в новом проекте концептов для C++ 255
Пилиди В. С., Шаренко Т. С. Модифицированный алгоритм Хафа для нахождения шаблонов на медицинских рентгенографических изображениях 270
Скороходов В. А., Шевелев М. В. Задачи о накоплении потока в ориентированных сетях 279
Столяр А. М. Асимптотический и численный анализ начально-краевых задач о колебаниях пластин, оболочек и тросов . 293

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.