Средства обобщённого программирования в объектно-ориентированных языках: особенности, недостатки, возможные альтернативы

Ю. В. Белякова
julbel@sfedu.ru

Южный федеральный университет
Институт математики, механики и компьютерных наук им. И. И. Воровича

24 октября 2016 г.
JetBrains Research:
семинар лаборатории языковых инструментов

Содержание

Обобщённое программирование

Термин «обобщённое программирование» (ОП) предложен в 1989 году Александром Степановым и Дэвидом Массером [1].

Идея

Код пишется в терминах **абстрактных** типов и операций.

Основная цель

Повторное использование кода.

Пример неограниченного обобщённого кода (Haskell)

```
count :: [a] -> (a -> Bool) -> Integer
count    []        p        =  0
count    (x:xs)    p        =  (if p x then 1 else 0) + count xs p
```

Рис.: Подсчёт числа элементов в обобщённом списке, удовлетворяющих предикату p (a — тип элементов списка)

Использование функции count:

```
ints  = [3, 2, -8, 61, 12]
evCnt = count ints (\x -> x `mod` 2 == 0)

strs     = ["hi", "bye", "hello", "stop"]
evLenCnt = count strs (\x -> length x `mod` 2 == 0)

main = do
    print evCnt          -- 3
    print evLenCnt       -- 2
```

Пример неограниченного обобщённого кода (C#)

```
static int Count<T>(T[] vs, Predicate<T> p)
{                           // p : T -> Bool
    int cnt = 0;
    foreach (var v in vs)
        if (p(v)) ++cnt;
    return cnt;
}
```

Рис.: Подсчёт числа элементов в массиве, удовлетворяющих предикату

Использование:

```
int[] ints = new int[]{ 3, 2, -8, 61, 12 };
var evCnt = Count(ints, x => x % 2 == 0); // 3

string[] strs = new string[]{ "hi", "bye", "hello", "stop" };
var evLenCnt = Count(strs, x => x.Length % 2 == 0); // 2
```

Пример неограниченного обобщённого кода (C#)

```
static int Count<T>(T[] vs, Predicate<T> p)
{                           // p : T -> Bool
    int cnt = 0;
    foreach (var v in vs)
        if (p(v)) ++cnt;
    return cnt;
}
```

Рис.: Подсчёт числа элементов в массиве, удовлетворяющих предикату

Использование:

```
int[] ints = new int[]{ 3, 2, -8, 61, 12 };
var evCnt = Count(ints, x => x % 2 == 0); // 3

string[] strs = new string[]{ "hi", "bye", "hello", "stop" };
var evLenCnt = Count(strs, x => x.Length % 2 == 0); // 2
```

Функцию Count<T> можно инстанцировать любым типом

Обобщённый код слишком узкоспециализирован!

Посмотрим ещё раз на параметр vs:

```
static int Count<T>(T[] vs, Predicate<T> p)
{ ... }

int[] ints = ...
var  evCnt = Count(ints, ...

string[] strs = ...
var  evLenCnt = Count(strs, ...
```

Обобщённый код слишком узкоспециализирован!

Посмотрим ещё раз на параметр vs:

```
static int Count<T>(T[] vs, Predicate<T> p)
{ ... }

int[] ints = ...
var  evCnt = Count(ints, ...

string[] strs = ...
var  evLenCnt = Count(strs, ...
```

Проблема

Обобщённая функция Count<T> недостаточно обобщённая.
Она работает только с массивами.

Реальный C# код для функции Count

Решение: используем вместо массива **обобщённый интерфейс**.

```
interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();  ...
}
```

Рис.: Интерфейс IEnumerable<T>

```
static int Count<T>(IEnumerable<T> vs, Predicate<T> p)
{                                      // p : T -> Bool
    int cnt = 0;
    foreach (var v in vs) ...
```

Рис.: Подсчёт числа элементов в vs, удовлетворяющих предикату p

```
var   ints = new int[]{ 3, 2, -8, 61, 12 };
var  evCnt = Count(ints, x => x % 2 == 0);    // array

var intSet = new SortedSet<int>{ 3, 2, -8, 61, 12 };
var evSCnt = Count(intSet, x => x % 2 == 0);  // set
```

Когда необходимы ограничения

Как реализовать обобщённую функцию, которая находит максимальный элемент в коллекции?

Когда необходимы ограничения

Как реализовать обобщённую функцию, которая находит максимальный элемент в коллекции?

```
static T FindMax<T>(IEnumerable<T> vs)
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx < v)        // ERROR: operator ‘‘<’’
            mx = v;        // is not provided for the type T
    ...
```

Рис.: Первая попытка: неудача

Когда необходимы ограничения

Как реализовать обобщённую функцию, которая находит максимальный элемент в коллекции?

```
static T FindMax<T>(IEnumerable<T> vs)
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx < v)        // ERROR: operator ‘‘<’’
            mx = v;        // is not provided for the type T
    ...
```

Рис.: Первая попытка: неудача

Чтобы найти максимум в vs,
нужно уметь сравнивать между собой элементы типа T!

“Сравнимость” это **ограничение**.

Пример обобщённого кода с ограничениями (C#)

```
interface IComparable<T> { int CompareTo(T other); }

static T FindMax<T>(IEnumerable<T> vs)
    where T : IComparable<T>                // F-bounded polymorphism
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx.CompareTo(v) < 0) mx = v;
    return mx;
}
```

Рис.: Вычисление максимального элемента в коллекции vs

Пример обобщённого кода с ограничениями (C#)

```
interface IComparable<T> { int CompareTo(T other); }

static T FindMax<T>(IEnumerable<T> vs)
    where T : IComparable<T>                // F-bounded polymorphism
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx.CompareTo(v) < 0) mx = v;
    return mx;
}
```

Рис.: Вычисление максимального элемента в коллекции `vs`

`FindMax<T>` можно инстанцировать только типами, которые реализуют интерфейс `IComparable<T>`.

```
var ints = new int[]{ 3, 2, -8, 61, 12 };
var iMax = FindMax(ints);   // 61
var strs = new LinkedList<string>{ "hi", "bye", "stop", "hello" };
var sMax = FindMax(strs);   // "stop"
```

Явные ограничения на типовые параметры

Языки программирования предоставляют различные механизмы обобщённого программирования на основе **явных ограничений**:

- Haskell: классы типов;
- SML, OCaml: модули;
- Rust, Scala: трейты;
- Swift: протоколы;
- Ceylon, Kotlin, C#, Java: интерфейсы;
- etc.

C++

В C++ шаблоны *не*ограниченные!

Явные ограничения на типовые параметры

Языки программирования предоставляют различные механизмы обобщённого программирования на основе **явных ограничений**:

- Haskell: классы типов;
- SML, OCaml: модули;
- Rust, Scala: трейты;
- Swift: протоколы;
- Ceylon, Kotlin, C#, Java: интерфейсы;
- etc.

C++

В C++ шаблоны *не*ограниченные!

Более ранние исследования показали, что в контексте обобщённого программирования C# и Java уступают многим языкам программирования [2—4].

Основные проблемы поддержки ОП в C# и Java I

- отсутствие ретроактивного моделирования (невозможность реализовать интерфейс после определения класса);

```
interface IWeighed { double GetWeight(); }
static double MaxWeight<T>(T[] vs) where T : IWeighed { ... }

class Foo { ... double GetWeight(); }

MaxWeight<Foo>(...) // ERROR: Foo does not implement IWeighed
```

- отсутствие ассоциированных типов и распространения ограничений [5];

```
interface IEdge<Vertex> { ... }
interface IGraph<Edge, Vertex> where Edge : IEdge<Vertex>{ ... }

... BFS<Graph, Edge, Vertex>(Graph g, Predicate<Vertex> p)
    where Edge : IEdge<Vertex>
    where Graph : IGraph<Edge, Vertex> { ... }
```

Основные проблемы поддержки ОП в C# и Java II

- неоднозначная роль интерфейсов: одна и та же конструкция используется и как тип, и как ограничение;

```
interface IEnumerable<T> { ... }   // type
interface IComparable<T> { ... }   // constraint

static T FindMax<T>(IEnumerable<T> vs) where T : IComparable<T>
```

- проблема бинарных методов [6]: используем типовый параметр `T` для моделирования домена бинарной операции `Compare(T, T)` и накладываем рекурсивные F-ограничения `T : I<T>`, чтобы обеспечить совпадение типа получателя и аргумента в `T.CompareTo(T)`;

```
interface IComparable<T> { int CompareTo(T other); }

static T FindMax<T>(...) where T : IComparable<T> { ... }
```

Основные проблемы поддержки ОП в C# и Java III

- невозможность реализации методов по умолчанию;

```
interface IEquatable<T>
{
    bool Equal(T other);
    bool NotEqual(T other); // return !this.Equal(other);
}
```

- отсутствие статических методов;

```
interface IMonoid<T>
{
    T BinOp(T other);
    T Ident(); // ???
}
static T Accumulate<T>(IEnumerable<T> vs) where T : IMonoid<T>
{
    T result = ???;     // Ident
    foreach (T val in values)
      result = result.BinOp(val);
    return result;
}
```

Основные проблемы поддержки ОП в C# и Java IV

- проблема мультипараметрических ограничений: вместо одного ограничения на несколько типов

```
double Foo<A, B>(A[] xs) where <single constraint on A, B>
// the constraint includes functions like B[] Bar(A a)
```

приходится использовать набор ограничений;

```
interface IConstraintA<A, B>  where A : IConstraintA<A, B>
                              where B : IConstraintB<A, B> {...}
interface IConstraintB<A, B>  where A : IConstraintA<A, B>
                              where B : IConstraintB<A, B> {...}
double Foo<A, B>(A[] xs)
    where A : IConstraintA<A, B>
    where B : IConstraintB<A, B> {...}
```

- отсутствие поддержки нескольких моделей на уровне языка (модель — способ реализации типом ограничения).

Поддержка ОП в современных ОО-языках

Часть проблем, связанных с поддержкой обобщённого программирования, снимается в более современных объектно-ориентированных языках:

- **Scala** (2004, 2016): абстрактные типы (можно использовать в качестве ассоциированных [7]), реализация методов по умолчанию;
- **Rust** (2010, 2016): ретроактивное моделирование, ассоциированные типы, собственные типы (решают проблему бинарных методов), статические методы, реализация методов по умолчанию;
- **Ceylon** (2011, 2016): собственные типы, статические методы, реализация методов по умолчанию;
- **Kotlin** (2011, 2016);
- **Swift** (2014, 2016): ретроактивное моделирование, ассоциированные типы, статические методы, реализация методов по умолчанию;
- **Java 8** (2014, 2016): статические методы, реализация методов по умолчанию.

Примеры обобщённого кода в ОО-языках

```
trait Eqtbl {
  fn equal(&self, that: &Self) -> bool;
  fn not_equal(&self, that: &Self) -> bool { !self.equal(that) }
}
impl Eqtbl for i32
{ fn equal (&self, that: &i32) -> bool { *self == *that } }
```

Рис.: ОП в Rust: собственные типы, реализация методов по умолчанию, ретроактивное моделирование

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

protocol Container { associatedtype ItemTy ... }
func allItemsMatch<C1: Container, C2: Container where
    C1.ItemTy == C2.ItemTy, C1.ItemTy: Equatable> ...
```

Рис.: ОП в Swift: собственные типы, реализация методов по умолчанию, ретроактивное моделирование, ассоциированные типы

Какие проблемы объединяют ОО-языки?

Все рассмотренные объектно-ориентированные языки следуют *одному* подходу к ограничению типовых параметров.

Подход «ограничения это типы»

Интерфейс-подобные конструкции используются в двух ролях:

1. как типы в объектно-ориентированном коде;
2. как ограничения в обобщённом коде.

То есть интерфейс/трейт/протокол описывает свойство **одного** типа, который его реализует. Вследствие этого, ограничения-типы принципиально не могут поддерживать две возможности обобщённого программирования:

1. мультипараметрические ограничения;
2. множественные модели.

Концепт-паттерн I

В паттерне проектирования Концепт (≈ паттерн Стратегия?)
(“Type Classes As Objects and Implicits” by Oliveira et. al., 2010 [8])
ограничения на типовые параметры заменяются
полями класса/параметрами функции — “**концептами**”.

F-Ограниченный Полиморфизм

```
interface IComparable<T>
{ int CompareTo(T other); }   // *

static T FindMax<T>(
  IEnumerable<T> vs)
  where T : IComparable<T>    // *
{
  T mx = vs.First();
  foreach (var v in vs)
    if (mx.CompareTo(v) < 0)  // *
      ...
```

Концепт-паттерн

```
interface IComparer<T>
{ int Compare(T x, T y); }    // *

static T FindMax<T>(
  IEnumerable<T> vs,
  IComparer<T> cmp)           // *
{
  T mx = vs.First();
  foreach (var v in vs)
    if (cmp.Compare(mx,v) < 0)// *
      ...
```

Концепт-паттерн II

В Scala есть специальная поддержка паттерна: контекстные ограничения (context bounds) и имплиситы (implicits).

F-Ограниченный Полиморфизм

```
trait Ordered[A] {
  abstract def compare
                (that: A): Int
  def < (that: A): Boolean = ...
}

// upper bound
def findMax[A <: Ordered[A]]
           (vs: Iterable[A]): A
{ ... }
```

Концепт-паттерн

```
trait Ordering[A] {
  abstract def compare
                (x: A, y: A): Int
  def lt(x: A, y: A): Boolean = ...
}

// context bound (syntactic sugar)
def findMax[A : Ordering]
           (vs: Iterable[A]): A
{ ... }

// implicit argument (real code)
def findMax(vs: Iterable[A])
    (implicit ord: Ordering[A])
{ ... }
```

Преимущества концепт-паттерна

Обе проблемы подхода «ограничения это типы» решаются использованием этого паттерна проектирования!

1. мультипараметрические ограничения это мультипараметрические параметры-«концепты»;

```
interface IConstraintAB<A, B>
{ B[] Bar(A a); ... }

double Foo<A, B>(A[] xs, IConstraintAB<A, B> c)
{ ... c.Bar(...) ... }
```

2. множественные «модели» представляются различными классами, реализующими один интерфейс.

```
class IntCmpDesc  : IComparer<int> { ... }
class IntCmpMod42 : IComparer<int> { ... }

var ints = new int[]{ 3, 2, -8, 61, 12 };

var   minInt = FindMax(ints, new IntCmpDesc());
var maxMod42 = FindMax(ints, new IntCmpMod42());
```

Недостатки концепт-паттерна I

Концепт-паттерн **широко используется** в стандартных обобщённых библиотеках C#, Java, и Scala, но у него есть несколько **недостатков**.

Возможные накладные расходы

Дополнительные поля класса или параметры функции.

```
interface IComparer<T>
{ ... }

class SortedSet<T> : ...
{
    IComparer<T> Comparer;
    ...
}
```

Недостатки концепт-паттерна II

Концепт-паттерн **широко используется** в стандартных обобщённых библиотеках C#, Java, и Scala, но у него есть несколько **недостатков**.

Несогласованность моделей

На этапе времени выполнения объекты одного типа могут использовать разные модели ограничения.

```
static SortedSet<T> GetUnion<T>(SortedSet<T> a, SortedSet<T> b)
{
    var us = new SortedSet<T>(a, a.Comparer);
    us.UnionWith(b);
    return us;
}
```

Внимание! Результат `GetUnion(s1, s2)` может отличаться от `GetUnion(s2, s1)`!

Альтернативный подход к ОП

Было предложено несколько языковых расширений для обобщённого программирования, созданных под влиянием классов типов Haskell [9]:

- Концепты C++ [10, 11] (2003–2014) и концепты в языке G [12] (2005–2011);
- Генерализованные интерфейсы в JavaGI [13] (2007–2011);
- Концепты для C# [3] (2015);
- Ограничения в Java Genus [14] (2015).

Все эти расширения реализуют *альтернативный* подход к ограничению типовых параметров.

Подход «ограничения это НЕ типы»

Для ограничения типовых параметров используется отдельная конструкция языка. Она не может использоваться в роли типа.

Классы типов Haskell

```
class Eq a => Ord a where                -- type class (concept)
  compare :: a -> a -> Ordering
  (<=) :: a -> a -> Bool
  ...

instance Ord Int where                   -- instance (model)
  ...    -- Ord functions implementation
```

Рис.: Класс типов «упорядочение»

```
findMax :: Ord a => [a] -> a             -- a is constrained with Ord
...
findMax (x:xs) = ... if mx < x ...
```

Рис.: Использование класса типов Ord

Классы типов Haskell

```
class Eq a => Ord a where                -- type class (concept)
  compare :: a -> a -> Ordering
  (<=) :: a -> a -> Bool
  ...

instance Ord Int where                   -- instance (model)
  ...    -- Ord functions implementation
```

Рис.: Класс типов «упорядочение»

```
findMax :: Ord a => [a] -> a             -- a is constrained with Ord
...
findMax (x:xs) = ... if mx < x ...
```

Рис.: Использование класса типов `Ord`

Поддерживаются мультипараметрические классы типов

Множественные инстанции запрещены

Ограничения в Java Genus I

```
interface Iterable[T] { ... }

constraint Eq[T]                          { boolean T.equals(T other); }
constraint Comparable[T] extends Eq[T] { int  T.compareTo(T other); }

static T FindMax[T](Iterable[T] vs) where Comparable[T]
{   ...
        if (mx.compareTo(v) < 0) ...   }
```

Рис.: Вычисление максимального элемента в `vs`

Благодаря тому, что конструкции-ограничения являются внешними по отношению к типам, поддерживаются:

- статические методы;
- ретроактивное моделирование;
- мультипараметрические ограничения.

Ограничения в Java Genus II

Допускается реализация нескольких моделей, при этом согласованность моделей обеспечивается на уровне типов.

```
interface Set[T where Eq[T]]    {...}

model StringCIEq for Eq[String] {...} // case-insensitive equality model
model StringFLEq for Eq[String] {...} // equality on first letter

// case-sensitive natural model is used by default
Set[String] s1 = ...;

Set[String with StringCIEq] s2 = ...;
Set[String with StringFLEq] s3 = ...;

s1 = s2;           // Static ERROR, s1 and s2 have different types
s2.UnionWith(s3); // Static ERROR, s2 and s3 have different types
```

Рис.: Согласованность моделей

Какой подход более выразителен?

«Ограничения это типы»

Отсутствие языковой поддержки для мультипараметрических ограничений и множественных моделей, при этом Концепт-паттерн обладает собственными недостатками.

Ограничения можно использовать как типы.

«Ограничения это НЕ типы»

Поддержка мультипараметрических ограничений и множественных моделей на уровне языка.

Ограничения нельзя использовать в роли типов.

Голосуем за «Ограничения это НЕ типы»

Есть как минимум 3 аргумента в пользу этого подхода:

Голосуем за «Ограничения это НЕ типы»

Есть как минимум 3 аргумента в пользу этого подхода:

- Согласно [15] («разделение материи и формы»), на практике интерфейсы, которые используются в качестве ограничений (такие как `IComparable<T>`), почти никогда не используются в роли типов.

Голосуем за «Ограничения это НЕ типы»

Есть как минимум 3 аргумента в пользу этого подхода:

- Согласно [15] («разделение материи и формы»), на практике интерфейсы, которые используются в качестве ограничений (такие как `IComparable<T>`), почти никогда не используются в роли типов.
- В то же время мультипараметрические ограничения и множественные модели часто необходимы для обобщённого программирования.

Голосуем за «Ограничения это НЕ типы»

Есть как минимум 3 аргумента в пользу этого подхода:

- Согласно [15] («разделение материи и формы»), на практике интерфейсы, которые используются в качестве ограничений (такие как `IComparable<T>`), почти никогда не используются в роли типов.
- В то же время мультипараметрические ограничения и множественные модели часто необходимы для обобщённого программирования.
- Все остальные языковые возможности, существенные для обобщённого программирования, могут поддерживаться при любом подходе к ограничениям.

Концепт-параметры против концепт-предикатов I

Если поддерживаются множественные модели, ограничения на типовые параметры больше *не являются предикатами*.
Более точно: ограничения являются параметрами времени компиляции аналогичными типовым параметрам.

```
module type Eq = sig
  type t val
  equal : t -> t -> bool
end

let foo  {EL : Eq} xs ys =
  if EL.equal(xs, ys)
  then xs else xs @ ys

let foo' {E : Eq}  xs ys =
  if (Eq_list E).equal(xs, ys)
  then xs else xs @ ys
```

```
implicit module Eq_int =
struct
  type t = int
  let equal x y  = ...
end
implicit module Eq_list {E : Eq} =
struct
  type t = Eq.t list
  let equal xs ys = ...
end
let x = foo [1;2;3] [4;5]
let y = foo' [1;2;3] [4;5]
```

Рис.: OCaml modular implicits [16]

Концепт-параметры против концепт-предикатов II

Концепт-предикаты

```
// model-generic methods
interface List[T] { ...
  boolean remove(T x) where Eq[T];
}
List[int] xs = ...
xs.remove[with StringCIEq](5);

// model-generic types
interface Set[T where Eq[T]] {...}
Set[String] s1 = ...;
Set[String with StringCIEq] s2=...;

// constraints depend on types
constraint WeighedGraph[V, E, W]{.
Map[V, W] SSSP[V, E, W](V s)
where WeighedGraph[V, E, W] {...}

...pX = SSSP[MyV, MyE, Double
        with MyWeighedGraph](x);
```

Концепт-параметры

```
// model-generic methods
interface List<T> { ...
  boolean remove<! Eq[T] eq>(T x);
}
List<int> xs = ...
xs.remove<!StringCIEq>(5);

// model-generic types
interface Set<T ! Eq[T] eq>  {...}
Set<String> s1 = ...;
Set<String ! StringCIEq> s2 = ...;

// types can be taken from cons-s
constraint WeighedGraph[V, E, W]{.
Map<V, W> SSSP<V, E, W
! WeighedGraph[V, E, W] wg>(V s){.

...pX = SSSP<? ! MyWeighedGraph>
        (x);
```

Сравнение языков и расширений

Языковая поддержка ОП в ОО-языках	Haskell	C#	Java 8	Scala	Ceylon	Kotlin	Rust	Swift	JavaGI	G	C#^{cpt}	Genus	ModImpl
Использование ограничений как типов	○	●	●	●	●	●	●	●	◐	○	○	○	○
Явные собственные типы	—	○	○	◐	●$^{-}$	○	●	●	◐	—	—	—	—
Мультипараметрические ограничения	●	⚹	⚹	⚹	○	⚹	○	○	●	●	●	●	●
Ретроактивное расширение типа	—	●	○	○	○	●	●	●	○	○	○	○	—
Ретроактивное моделирование	●	⚹	⚹	⚹	○	⚹	●	●	●	●	●	●	●
Обобщённые условные модели	●	○	○	○	○	○	●	○	●	●	●	●	●
Статические методы	●	○	●	○	●	●	●	●	●	●	●	●	●
Реализация методов по умолчанию	●	○	●	●	●	●	●	●	◐	●	●	○	○
Ассоциированные типы	●	○	○	●	○	○	●	●	○	●	●	○	●
Ограничения на ассоциированные типы	◐	—	—	●	—	—	●	●	—	●	●	—	●
Ограничения подтипирования	◐	—	—	●	—	—	●	●	—	●	●	—	●
Перегрузка на основе ограничений	○	○	○	○	○	○	●	○	○	◐	○	○	○
Множественное определение моделей	○	⚹	⚹	⚹	⚹	⚹	○	○	○	◐^{a}	●	●	●
Согласованность моделей	—^{b}	○	○	○	○	○	—^{b}	—^{b}	—^{b}	—^{b}	●	●	●
Ограничения на типы в методах	—	⚹	⚹	⚹	⚹	⚹	●	○	○	○	○	●	—

⚹ означает поддержку через Концепт-паттерн. [a] G поддерживает модели, ограниченные пространством имён.
[b] Если множественные модели не поддерживаются, понятие согласованности не имеет смысла.

Открытые вопросы дизайна

- Вариантность концептов.

```
interface ISet<T | Equality[T] eq> { ... }
class B { ... }    class D : B { ... }
model EqB for Equality[B] { ... }
```

Допустима ли инстанция `ISet<D, EqB>`?

Можно ли считать, что класс `SortedSet<T | Ordering[T] ord>` реализует интерфейс `ISet<T|ord>`?

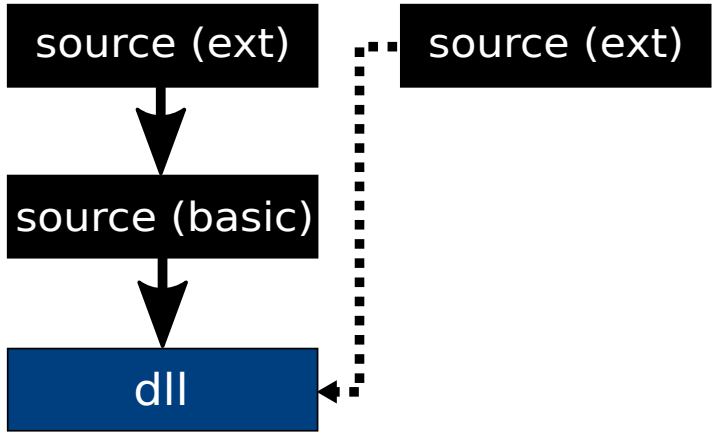
- Статическое/динамическое связывание концепт-параметров.

```
void foo<T | Equality[T] eq>(ISet<T|eq> s) { ... } ...
ISet<string | EqStringCaseS> s1 =
    new SortedSet<string | OrdStringCSAsc>(...);
foo(s1);
```

Какая модель `Equality[string]` должна быть использована внутри `foo<>`? Статическая `EqStringCaseS` или динамическая `OrdStringCSAsc`?

Проблемы реализации

- Обеспечение раздельной компиляции и модульности при реализации переводом в базовый язык.



- Поддержка ограничений подтипирования в концепт-подобных конструкциях (простая унификация равенств больше не подходит).
- Необходимость исследования возможностей концептов в рамках теории типов.

Литература I

D. Musser и A. Stepanov. «Generic programming». English. В: *Symbolic and Algebraic Computation*. Под ред. P. Gianni. Т. 358. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1989, с. 13—25.

R. Garcia и др. «An Extended Comparative Study of Language Support for Generic Programming». В: *J. Funct. Program.* 17.2 (март 2007), с. 145—205.

J. Belyakova и S. Mikhalkovich. «Pitfalls of C# Generics and Their Solution Using Concepts». В: *Proceedings of the Institute for System Programming* 27.3 (июнь 2015), с. 29—45.

J. Belyakova и S. Mikhalkovich. «A Support for Generic Programming in the Modern Object-Oriented Languages. Part 1. An Analysis of the Problems». В: *Transactions of Scientific School of I.B. Simonenko. Issue 2* 2 (2015), 63—77 (in Russian).

Литература II

J. Järvi, J. Willcock и A. Lumsdaine. «Associated Types and Constraint Propagation for Mainstream Object-oriented Generics». В: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, с. 1—19.

K. Bruce и др. «On Binary Methods». В: *Theor. Pract. Object Syst.* 1.3 (дек. 1995), с. 221—242.

A. Pelenitsyn. «Associated Types and Constraint Propagation for Generic Programming in Scala». English. В: *Programming and Computer Software* 41.4 (2015), с. 224—230.

B. C. Oliveira, A. Moors и M. Odersky. «Type Classes As Objects and Implicits». В: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, с. 341—360.

Литература III

P. Wadler и S. Blott. «How to Make Ad-hoc Polymorphism Less Ad Hoc». В: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, с. 60—76.

G. Dos Reis и B. Stroustrup. «Specifying C++ Concepts». В: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, South Carolina, USA: ACM, 2006, с. 295—308.

D. Gregor и др. «Concepts: Linguistic Support for Generic Programming in C++». В: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, с. 291—310.

J. G. Siek и A. Lumsdaine. «A Language for Generic Programming in the Large». В: *Sci. Comput. Program.* 76.5 (май 2011), с. 423—465.

Литература IV

S. Wehr и P. Thiemann. «JavaGI: The Interaction of Type Classes with Interfaces and Inheritance». В: *ACM Trans. Program. Lang. Syst.* 33.4 (июль 2011), 12:1—12:83.

Y. Zhang и др. «Lightweight, Flexible Object-oriented Generics». В: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. Portland, OR, USA: ACM, 2015, с. 436—445.

B. Greenman, F. Muehlboeck и R. Tate. «Getting F-bounded Polymorphism into Shape». В: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, с. 89—99.

L. White, F. Bour и J. Yallop. «Modular Implicits». В: *ArXiv e-prints* (дек. 2015). arXiv: 1512.01895 [cs.PL].

Зависимые типы

```
-- natural number
data Nat       --  Nat : Type
  = Zero       -- Zero : Nat
  | Succ Nat   -- Succ : Nat -> Nat

-- generic list
data List a           -- List : Type -> Type
  = []                --   [] : List a
  | (::) a (List a) -- (::) : a -> List a -> List a

-- vector of the length k (dependent type)
data Vect : Nat -> Type -> Type where
  Nil  : Vect Zero a
  Cons : a -> Vect k a -> Vect (Succ k) a
```

Рис.: Типы данных и зависимые типы в Idris

Если бы ОО-языки поддерживали зависимые типы, можно было обеспечить согласованность моделей (сделав компаратор частью зависимого типа).

Типобезопасный концепт-паттерн

Можно обеспечить согласованность моделей, если сделать «концепт» типовым параметром:

```
interface IComparer<T> { int Compare(T, T); }

static T FindMax<T, CmpT>(IEnumerable<T> vs)
    where CmpT : IComparer<T>, struct
{
    CmpT cmp = default(CmpT);
    T mx = vs.First();
    foreach (var v in vs)   if (cmp.Compare(mx, v) < 0) ...  }

struct IntCmpDesc  : IComparer<int> { ... }
struct IntCmpMod42 : IComparer<int> { ... }

var ints = new int[]{ 3, 2, -8, 61, 12 };
var   minInt = FindMax<int,  IntCmpDesc>(ints);
var maxMod42 = FindMax<int, IntCmpMod42>(ints);
```

См. «Classes for the Masses» на ML Workshop (ICFP).