ISSN 0361-7688, Programming and Computer Software, 2015, Vol. 41, No. 4, pp. 224–230. © Pleiades Publishing, Ltd., 2015.
Original Russian Text © A. Pelenitsyn, 2015, published in Programmirovanie, 2015, Vol. 41, No. 4.

Associated Types and Constraint Propagation for Generic Programming in Scala

A. Pelenitsyn

Southern Federal University, ul. B. Sadovaya 105/42, Rostov-on-Don, 344006 Russia
e-mail: apel@sfedu.ru

Received March 5, 2015

Abstract—Generic programming is a programming paradigm for creation of highly resuable software components through decoupling algorithms from specific data structures which are being processed. The rise of research on ways of handling generic programming in various programming languages took place last years. We analyze and develop a number of generic programming features, in particular associated types and constraint propagation, for the Scala programming language designed by Martin Odersky in École Polytechnique Fédérale de Lausanne.

DOI: 10.1134/S0361768815040064

1. INTRODUCTION

Generic programming is a programming paradigm for creation of highly reusable software components through decoupling algorithms from specific data structures which are being processed [1].

Various programming languages provide facilities for generic programming. We observe three approaches to express ideas of generic programming in modern programming languages. The first one is to employ current language features. Examples of this are C++ templates and generic types in common object-oriented languages (Java, C#). The second approach is to build corresponding language extensions (e.g., for C# language in [2] and JavaGI extension from [3]). The third approach is to develop new programming languages which possesses desired features from the very beginning. Main example here is Scala programming language, though we can mention other languages like Agda and Coq mostly targeted to proof assistance and hence used mainly in academics.

One notable example of language with well-developed generic programming facilities is Haskell. It was developed as a language with one major compiler opened for experiments in type theory. Some results of these are being included in new revisions of a language standard. Even basic Haskell features like type classes and widely used parametric polymorphism present well-developed framework for application of generic programming. Thus, Haskell satisfy both, first and second approaches to generic programming implementation mentioned above. A number of nonstandard but widely adopted extensions for Glasgow Haskell Compiler (GHC), such as multi-parameter type classes and functional dependencies strengthen language positions in this area.

Historically first programming language widely adopted principles of generic programming was C++. Unconstrained C++-templates give much freedom in employing type parameters, basic tool of generic programming. Though actual usage of templates has a number of well-known drawbacks, such as complicated even unreadable error messages and hence tough debugging process, as well as unavailability of separate translation for generic units (functions and classes templates). New language standard, known as C++11, should have improved support for generic programming in C++. Though the "Remove-decision" banned so called *concepts* proposal and ruined these hopes a lot. Nevertheless two reference implementation for generic programming used in research today are that of C++ and Haskell: they give main examples to compare with for evaluating new approaches to generic paradigm.

Indeed, proposed in [2] C# extension is tested against an example from Boost Graph Library (BGL) written in C++. The main idea there is how to overcome difficulties arise when use common C# language for implementation of a BGL fragment. The extension consists of associated types and generic type constrain propagation.

One of the most outstanding feature of Scala programming language, *implicits*, is described in [4]. Several tasks of various difficulty are implemented via implicits there: from modeling of Haskell type classes which shows Scala ability to support of generic programming to type-level computations such as computation of a type for function

$$\texttt{zipWithN} :: (a_1 \to a_2 \to \ldots \to a_n) \to ([a_1] \to [a_2] \to \ldots \to [a_n])$$

[1] The article was translated by the authors.

Here n is a parameter. The function takes a function from $n - 1$ parameter and $n - 1$ lists and returns a list of results after application the function given to corresponding elements of the lists. This type-level computation task resembles one discussed in [5] where multivariate polynomials implemented in C++ using similar ideas.

While presenting one particular language feature, [4] deduce a number of general properties of Scala language concerning generic programming. In particular, the paper extends comparative table of handling generic programming features in modern programming languages. The first version of the table was published in seminal paper [6], which didn't consider Scala language and explored the ways of handling an example from above mentioned BGL library in number of programming languages.

An analysis of possible implementation for fragment of BGL in terms of extended C# is done in [2]. We perform here similar analysis for Scala programming language, which appears more promissory by means of generic programming. We explore Scala elements which allow for more succinct implementation of ideas of associated types and constraint propagation when solving a number of typical problems of generic programming. We evaluate our solution on example from [2].

The first main result of our paper is application of refinement types for description of constraints on associated types. Next, we show how to eliminate parametric polymorphism there with path-dependent method types. This allows for more clear and succinct solution for task considered in [2].

The second main result is finding and removal of a particular flaw from [2], namely the absence of retroactive modeling property which was listed as one of criteria for support of generic programming in [6]. General approach for handling this in Scala is considered in [4], we show how to adapt it in typical setting of industrial-strength programming library.

Our results expand the description of Scala support for generic programming given in [4]. We use more accessible example which at the same time is closer to industrial problems than in [4], namely a fragment of BGL. More realistic example motivates usage of Scala features which weren't involved in such kind of problems earlier (refinement types) or were used in a minor way (abstract types, path-dependent types). We compare construction obtained with results from [2], as well as with some generic programming tools from C++ and Haskell.

In [7], Section 6 Related work it is noted that Scala has some prospects in respect of generic programming. Authors of [7] mention their plans on performing research on associated types and implicits exploring these prospects, but no evidence of realization of the plans found, so we perform such research in this paper.

The paper consists of introduction, three sections and conclusion. In Section 2 we recall results from [2] in convenient way: a fragment of BGL is described, a C#-implementation (including extensions from [2]) for it is given. Moreover, we discuss the weaknesses of these results. In Section 3 we demonstrate Scala features for generic programming using the same BGL example. We note Scala advantages over extended C# from [2]. In Section 4 we note some digression on generic programming principles taken in [2] and construct solution for this using ideas from [4]. Also, we perform some analysis here about overhead of using generic programming form both, client's and library author's points of view. At the end of the section we give some remarks on the same topic concerning other programming languages. Two main results of the paper described above are in Section 3 and Section 4 correspondingly.

2. A BGL FRAGMENT AND ITS IMPLEMENTATION BY MEANS OF EXTENDED C#

For the purpose of illustration generic facilities of extended C# the following function from BGL (C++) is taken in [2]:

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

Given a graph object and one of its vertex this function returns some adjacent vertex ("first" in some ordering). One obvious technical drawback of this code is the mandatory `typename` keyword designated nested type `vertex_type` from class member of type parameter `Graph`. More fundamental flaw of unconstrained templates of C++ visible here is that header and body of the function contain lots of assumptions about type parameter `Graph`, which are not expressed directly but rather outlined in documentation. Function client should either read through documentation carefully or take cut-and-try approach running compiler and trying to guess about those assumptions from weird compilation error messages. Compilation time in C++ is long and though huge efforts were performed to make compiler messages more clear, second approach still poses heavy burden on a programmer.

The way taken by C++ community to overcome the problem is to develop formalized documentation for generic libraries. Central notion on the way is *concept*—a set of constraints put on type parameters of generic functions and classes. Enhancing documentation is quite limited though. Therefore an attempt to add concepts directly to language syntax was made (cf. [8] for complete overview of this process with relevant references). Eventually concepts proposal was postponed and did not find its way to C++11 standard.

We consider this as C++'s failure in enhancing generic programming support [11].

Alternative for unconstrained templates of C++ in handling generic programming could be found in modern object-oriented languages, e.g. Java and C#. The difference between the two (C++ and Java/C#) is well-studied in type theory: C++ support parametric impredicative polymorphism while Java/C# implement bounded polymorphism formalized in terms of System $F_{<:}$—the union of classical System F and subtyping [9]. The difference could be expressed as follows: allowed everything that is not forbidden (unconstrained templates) versus forbidden everything that is not allowed (bounded polymorphism). Second approach leads to increase in amount of type parameters and more precise description their dependencies. It turns out that basic facilities of Java/C# are not well-suited for this. Consider example of above mentioned function reimplemented in standard C#.

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

As before there are lots of assumptions for type parameters in `first_neighbor`. Now they reflected directly in function signature in contrast to C++ version. Second obvious but merely stylistic difference from C++ is that function calls become class methods calls.

Let us exemplifiy assumption about type of parameter g in function above: the type should allow for call `out_edges` method. In pure object-oriented language this kind of assumptions usually fulfilled through implementation of an interface. We call this interface `IncidenceGraph`. The method returns an iterator for a collection of adjacent edges and we have to introduce another type parameter `G_OutEdgeIterator` implementing standard interface `IEnumerable`. The latter one has a property `Current` which gives access to the current edge, and so on.

First problem arose in this solution is dramatic increase in amount of type parameters. Next problem become evident while inspecting definition of any interface used. They are generic interfaces demanding explicit constraints for type parameters—just the same ones we have already used in definition of above mentioned function.

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

Certainly we would like to write down Edge and `OutEdgeIterator` constraints only once, when defining `IncidenceGraph` type, augmenting the need to repeat them in every function similar to `first_neighbour`.

Both problems mentioned are solved in [2] through extension of C# language with associated types and automatic constrain propagation correspondingly. Associated types feature allow for declaration of abstract (not defined) types inside interfaces along with usual abstract methods (without definition). These abstract type members should be made specific in class implementing interface just like usual abstract methods are. The notion “abstract type members” used in Scala looks more thorough than “associated types” from [2] for us.

Once adding associated types to interfaces one need to define constraints on them like it is done for type parameters. One traditional form of constraints in OO-languages is subtype constraints: type T should be subtype of some specific type. It turns out that another useful form of constraints is equality constraints for type parameters or associated types. For example type parameters (or associated types) `Vertex` used in `GraphEdge` interface and in `IncidenceGraph` interface should be equal if used in definition of the same function.

Constraints propagation is a tool allowing compiler to use type constraints from various interfaces for proof that code using these interfaces is valid in type system.

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

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

Fig. 1.

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

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

Fig. 2.

Following code obtained in [2] for initial example function on extended C# (see Fig. 1) reflects these ideas.

Labels (1) point to associated types used instead of type parameters. Labels (2) point to subtype constraints. Labels (3) point to equality constraints.

3. NESTED TYPE DECLARATIONS AND GENERIC CONSTRAINTS IN SCALA

Scala has nested type declarations (so called abstract type members) from the very beginning. It also has a number of ways for describing constraints on abstract types and type parameters for interfaces and methods. Moreover Scala modifies notion of interface into so called *trait*. Traits could be used as usual interfaces known from Java/C#. But traits can also contain method implementation. We do not use this facility in the paper but one interested in ways of handling classical problems of multiple inheritance in Scala could learn it from any Scala book, e.g. [10].

Scala compiler has thorough ability for type analysis and the feature of constraint propagation considered in [2] comes here for no price: if constraints are placed inside a trait they will be counted when object of the trait is used. As a result we can build solution by no means weaker than in [2], see Fig. 2.

Class or trait member types in Scala can be specific:

```
type Vertex = Int
```

In this case they act as usual type aliases (like *typedef* in C++). Another possibility is abstract type members labeled (1) on Fig. 2. They can be made specific in trait subtypes.

Abstract type in Scala can be bounded (cf. (2) on Fig. 2). Declaration

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

guarantees that any specialization of type **A** would be subtype of **B**.

Nice evidence of maturity of Scala type system is the fact that subtype constraints could be expressed without any dedicated keywords. This contrasts with what we saw in [2] with `require`.

Let us consider (3) from Fig. 2 closer. The whole expression has a form of (4) where B is an example of *refinement type*:

```
GraphEdge { type Vertex = Inci-
denceGraph.this.Vertex } // (5)
```

The type constitutes supertype of any type which extends `GraphEdge` and define its `Vertex` type member just as a `Vertex` in current object (this) implementing `IncidenceGraph` trait. It seems that expression `IncidenceGraph.this.Vertex` is overcomplicated compared with associated type access from [2]. In Scala one have so called *type projections* which look more familiar: `Incidence-Graph#Vertex`. This expressions describe all type members `Vertex` in any type extending `Inci-denceGraph`. It is clear that such wide notion is not suitable for the task: if one graph type defines its vertices as *Int* and other graph type prefers *String* then `first_neighbor` method working with first graph type should return an integer ant not a string.

An ability to describe constraints on associated types is listed as a criteria for support of generic programming [6]. As far as we know the usage of refinement types wasn't used before with this purpose in Scala.

Thus, (5) describes exactly the set of types we want to use for `Edge` in (3). Last thing to consider in (3) is the following. The <: operator in Scala language normally denotes subtype relation. As (5) express desired set of types we could probably think of using = instead of <:. But = turn whole (3) in an alias for exactly the huge (5) expression instead of abstract type. So we definitely should use <: and not = in (3).

In `first_neighbor`'s header we do not need type parameters like in [2] where type parameter G gives access to nested type `G::Vertex`. In Scala one use an object reference for this purpose (g in this case) and not type projections like `G#Vertex` due to above mentioned problem with too wide notion behind this expression. If parameter type depends on type of other parameter (like type of `v` in this case depends on type of `g`) one should place them in different argument lists, i.e. different pair of parentheses—it is a language restriction, so called path-dependent method types.

A notion of *path* plays major role in Scala type system. A path is an expression in form A.B.C.D. Only four types of identifiers allowed for A, B, …: a package, an object, *val*-value, *this/super* and their variants with type arguments. If a path is valid, one say that the path define stable type, that is specific type known at compile time. This distinguishes paths from projections like `G#Vertex` which could have been used in `first_neighbor`.

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

This variant goes closer to that from [2] where there is no differences like one have in Scala with paths and projections. Though this results in weaker type control as we mentioned above. Also when using projection approach we have to change declaration of `out_edges`: this method should accept projection too. Overall result is more code and less type control. This does not look inviting.

Main OO-languages like C++, Java and C# do not have features like abstract type members, so they do not face distinction like that between paths and projections. One usually talk about type qualification in these languages.

4. IMPLICITS AS A TOOL FOR PUTTING CONSTRAINTS ON TYPE PARAMETERS

Solution from previous section comes quite close to one described in [2]. It is possible due to Scala features: abstract type members, refinement types and path-dependent method types. This solution largely utilizes subtyping in various forms and does not use type parameters. This style more common for object-oriented rather than generic approach. Consequently, it does not satisfy some important requirements for generic programming code. The one we would like to discuss here is *retroactive modeling*. In our example this means inability of `first_neighbor` to work with types which do not extend `IncidenceGraph` trait (do not implement mandatory interface in more usual terms). This is an implication of OO-approach. The problem arise for a `first_neighbor`'s client who employ custom graph type which do not "know" about `IncidenceGraph` trait and which is not subject of change (due to, e.g., backward compatibility requirements). The problem is solved by means of generic approach if a programming language supports it on considerable level. In [4] it is shown how to solve the problem in Scala. We study such solution in specific case from [2] using ideas from [4].

A set of constraints on type parameters is usually expressed without subtyping relation in generic programming. Instead a set of constraints is factored out to separate entity called a *concept* or a *type class*. First term comes from C++-community and second one is from Haskell. We will use first one here. One should define concept *model* to describe how specific type satisfy concept requirements. Note that this can be done long after the type was defined. This stands for retroactive modeling.

We have to answer on a number of questions in order to understand generic approach of Scala. First one is how does a concept defined. The answer comes by means of generic trait. Here is example concept for graph edge.

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

Fig. 3.

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

Fig. 4.

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

We may see this as a predicate: type E satisfies predicate "to be an edge type" if vertex type for it is defined and methods accessing source and target are supplied.

Once having specific type and a concept for it we can create a model. Suitable way for this is through Scala *objects*.

```
type BasicEdge = (Int, Int)
```

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

The `basicEdgeMod` object proves that `BasicEdge` type satisfies `GraphEdge` concept. Scala object can be thought of as an implementation of Singleton design pattern at the language level: it describes a type and a single object of the type at once. Objects facility is quite suitable for defining concept models. This observation first made in [4].

The whole graph concepts is defined as follows.

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

The concept requires model for graph edge type `Edge`: any sibling of the trait will have to define value for `edgeMod` member. The graph identifier is an alias for `IncidenceGraph.this` used in Section 3 for the same purpose; thus the `graph` ⇒ clause introducing the alias serves just for shortening our code.

We implement `IncidenceGraph` concept model for `AdjacencyList` type as in Fig. 3.

We do not provide an implementation for `out_edges` method as it depends on implementation of `AdjacencyList` interface. The `implicit` keyword before an object definition allows for usage an object as an implicit method argument for any method accepting such arguments. The `first_neighbor` methods delivers such an example, see Fig. 4.

Note that we do not require graph and vertex types implementing any specific interfaces no more. Instead we seek for extra argument showing type `G` has `IncidenceGraph` concept model. A call to this method may look as follows.

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

Here `randomAdjacencyList` returns random graph. Compiler should find `adjListModel` model and substitutes it for `igMod` if it is placed in suitable scope (exact rules for such scope could be found in references, e.g. chap. 21 in [10].

Note that library method call remains the same as in approach from Section 3. At the same time function declaration header become more complicated. And function implementation become more tricky. So the main overhead on added genericity is on the library side, not the client side, which seems acceptable. Let us briefly compare this to Section 3 approach. Client have to define models for her own types but may use library models for library types. In the latter case client will not notice any differences as compared with Section 3 approach.

Can we lower the costs for defining client's own models? There are special-purpose tools for this in languages handling generic programming from the very beginning. E.g. in Haskell, type class instances (concept model full analog) can be derived automatically for some basic type classes. There is ongoing discussion in C++-community on the actual shape of concept facility in the language. In particular, it is debated how much automation for "concept maps" (which are concept models in our terms) creation is acceptable.

Consider example when automatic model deriving is helpful. Imagine we have graph type just like in Section 3 with all necessary nested type declarations and `out_edges` method. How can it be used in `first_neighbor` method from current section? We need a model where `out_edges` method is as follows.

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

Indeed, definition like this could be easily generated by compiler. Must we add this feature to compiler? Such a question is considered by B. Stroustrup in [11] (Technical Issues section) where possible concept inclusion in the C++ language standard is discussed (post-C++11 standard). Stroustrup thinks that this feature should be added. Though there is not such a feature in Scala at the moment.

5. CONCLUSIONS

In this paper we choose elements of Scala programming language for describing requirements on generic interfaces of Boost Graph Library. We undertake two approaches for implementation of the requirements: first one translates ideas from [2] to Scala and is mostly based on subtyping, second one goes closer to [4] and makes explicit such generic programming entities as concepts and their models.

Use of Scala traits may have a number of issues inherent to subtyping. Example from the end of Section 4 is just single trivial case of these issues. At the same time Scala presents *structural types* which allow for listing type members and eliminating the need in explicit extension of particular trait. We think that structural types may become more flexible basis for generic programming on Scala. Though this observation requires additional research.

The author would like to thank participants of study group on foundations of programming languages headed by assoc. prof. Stanislav Mihalkovich for detailed discussion of papers on the topic, as well as members of the study group, Julia Belyakova and Vitaly Bragilevsky for consulting and valuable advices on solution for problems touched upon in the paper.

REFERENCES

1. Musser, D.A. and Stepanov, A.A., Generic Programming, *Proceeding of International Symposium on Symbolic and Algebraic Computation*, vol. 358: Lecture Notes in Computer Science, Rome, Italy, 1988, pp. 13–25.
2. Jarvi, J., Willcock, J., and Lumsdaine, A., Associated types and constraint propagation for mainstream object-oriented generics, *OOPSLA'05 Proc. of the 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, NY, USA: ACM New York, 2005, pp. 1–19.
3. Wehr, S., Lammel, R., and Thiemann, P., JavaGI: Generalized Interfaces for Java, *Proc. of the European Conf. on Object-Oriented Programming*, Ernst, E., Ed., LNCS, vol. 4609, Berlin, Germany: Springer-Verlag, 2007, pp. 347–372.
4. Oliveira, B.C.d.S., Moors, A., and Odersky, M., Type classes as objects and implicits, *OOPSLA'10 Proc. of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications*, NY, USA: ACM New York, 2010, pp. 341–360.
5. Pelenitsyn, A., Generic and metaprogramming in software implementation of decoder for algebraic geometry codes, *Prikl. Inform.*, 2012, no. 2 (38), pp. 60–70.
6. Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J., and Willcock, J., An extended comparative study of language support for generic programming, *J. Funct. Program.,* 2007, vol. 17, no. 2, pp. 145–205.
7. Gregor, D., Jarvi, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A., Concepts: Linguistic Support for Generic Programming in C++, *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, Portland, Oregon, 2006, pp. 291–310.
8. Sutton, A. and Stroustrup, B., Design of Concept Libraries for C++, *Proc. SLE 2011 (International Conference on Software Language Engineering)*, 2011, pp. 97–118.
9. Cardelli, L. and Wegner, P., On Understanding Types, Data Abstraction, and Polymorphism, NY, USA, N.Y.: ACM Computing Surveys, 1985, vol. 17, no. 4, pp. 471–523.
10. Horstmann, C.S., *Scala for the Impatient,* Addison-Wesley, 2012.
11. Stroustrup, B., The C++0x "Remove Concepts" Decision, *Dr. Dobb's J.*, 2009; URL: http://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111.