GENERIC PROGRAMMING IN SCALA

A Thesis

by

OLAYINKA N'GUESSAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2006

Major Subject: Computer Science

GENERIC PROGRAMMING IN SCALA

A Thesis

by

OLAYINKA N'GUESSAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2006

Major Subject: Computer Science

ABSTRACT

Generic Programming in Scala. (December 2006)

Olayinka N'guessan, B.S., Minnesota State University, Mankato

Chair of Advisory Committee: Dr. Jaakko Järvi

Generic programming is a programming methodology that aims at producing reusable code, defined independently of the data types on which it is operating. To achieve this goal, that particular code must rely on a set of requirements known as concepts. The code is only functional for the set of data types that fulfill the conditions established by its concepts. Generic programming can facilitate code reuse and reduce testing.

Generic programming has been embraced mostly in the C++ community; major parts of the C++ standard library have been developed following the paradigm. This thesis is based on a study (by Garcia et al.) on generic programming applied to other languages (C#, Eiffel, Haskell, Java and ML). That study demonstrated that those languages are lacking in their support for generic programming, causing difficulties to the programmer.

In this context, we investigate the new object-oriented language Scala. This particular language appealed to our interest because it implements "member types" which we conjecture to fix some of the problems of the languages surveyed in the original study. Our research shows that Scala's member types are an expressive language feature and solve some but not all of the problems identified in the original study (by Garcia et al.).

Scala's members types did not resolve the problem of adding associated types to the parameter list of generic methods. This issue led to repeated constraints, implicit instantiation failure and code verbosity increase. However, Scala's member types enabled constraint propagation and type aliasing, two significantly useful generic programming mechanisms.

To my amazing parents: Etienne and Stella N'guessan

ACKNOWLEDGMENTS

I feel extremely fortunate to have had Dr. Jaakko Järvi as my advisor. He is one of the best advisors one could wish to have for a research project. In addition to being an extremely hard-working professor, he has proven to be very attentive to my needs as a graduate student. I am very impressed with his unfailing excitement at research, his inspiration, technical direction and his patience. It did not matter if he was at the other end of the world, he would still follow my work and progression closely. He was very prompt at answering my lengthy emails whenever I had questions and at providing me with the most detailed answers. Even though Dr. Jaakko Järvi is a strict grader, he is a very compassionate: he would always allow some flexibility whenever I needed more time to accomplish my tasks. Also, I will always remember his sense of humor especially the famous array of pandas (`array[Panda]`). Without Dr. Jarvi's help, this thesis would not have been possible.

I would like to express my gratitude to the members of my committee, Dr. Bjarne Stroustrup and Dr. Roland Allen, for their interest in willing to be part of my thesis defense; I am extremely honored by their presence. I would also like to thank Dr. Martin Odersky for his help and advice for this research.

On a final note, I would like to acknowledge my entire family for their unending love and support. I would like to thank my father, Etienne N'guessan for all that he has done for me; he will forever be my greatest role model. I praise my mother, Stella N'guessan for giving me love and strength even when I fell apart. I feel blessed to have the most amazing and understanding sisters: I thank Christine for her wisdom and Lucie for her attentiveness. Also, I would like to thank my brother Stephen for motivating me and making me laugh every day.

Ante non volebant me, num calida sum, plene sunt ad me.

Mike Jones

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Generic programming is a language-independent programming paradigm where algorithms are not expressed in terms of particular types but in terms of *concepts* (properties of types). Generic programming aims at achieving high degree of code reusability and reduced code verbosity without sacrificing performance. The domain of reusable libraries of software components is one area where generic programming has proven to be particularly effective [1, 2].

Generic programming started to gain momentum since the inclusion of the Standard Template Library (STL) [3] in the C++ standard library in 1994. Several libraries such as the Boost Graph Library (BGL) [4] and the Matrix Template Library (MTL) [5, 6] have been developed using the generic programming methodology. C++ has been the prime language used for such generic libraries. Even though C++ remains the most commonly used language for implementating generic libraries, various languages possess generic programming enabling features.

The starting point of this thesis is the comparative study conducted in [7] that evaluated the suitability of different mainstream programming languages (C++ [8], C# [9, 7], Eiffel [10], Haskell [11], Java [12] and ML [13]) for generic programming. In that study, eight language features essential to generic programming were identified [7]:

1. Multi-type concepts
2. Multiple Constraints

The journal model is *IEEE Transactions on Automatic Control.*

3. Associated type access

4. Retroactive modeling

5. Type aliasing

6. Separate Compilation

7. Implicit Instantiation

8. Concise Syntax

A problem area for the object-oriented languages proved to be the lack of support for accessing associated types (we give a detailed definition of associated types in Chapter II). The studied object-oriented languages lack a direct mechanism to support access to associated types. As a work-around, one adds associated types to the parameter list of generic methods and classes. This results in an increase in the code verbosity.

Scala [14], a recently introduced object-oriented language was not part of the languages studied in [7]. Scala supports generics similarly to Java and C#; but unlike these languages, Scala supports "member types". We conjectured that these member types would suffice in expressing and accessing associated types. Consequently, this thesis investigates Scala's support for generic programming with a particular focus on associated types using member types.

To conduct our study, we have implemented a model library by extracting a small but significant example of generic programming from the Boost Graph Library. The result of the analysis of our implementation constitute the main points of this thesis. The result of this thesis serve on one hand to language designers and on the other hand to library writers who desire to follow the generic programming approach in developing software libraries in Scala.

This thesis has been organized as follows: Chapter II gives a detailed presentation of generic programming including its problems, Chapter III briefly describes the Boost Graph Library, Chapter IV presents the Scala programming language, Chapter V describes our implementation and analysis of the Boost Graph Library in Scala, and Chapter VI sums up the findings of our work.

CHAPTER II

GENERIC PROGRAMMING

1. Generic Programming: Definition

Generic programming is a branch of computer science that focuses on finding abstract representation of useful algorithms, data structures and software concepts with the goal of making them adaptable for direct use in software construction. The following list explains the key ideas of generic programming [15]:

- Designing algorithms with minimal assumptions about data abstractions and expressing data abstractions with minimal assumptions about the algorithm that will use them.
- Abstraction should not sacrifice efficiency when lifting a concrete algorithm to a more generic level; performance should stay the same.
- Providing specialized forms of algorithms if a single algorithm is not efficient enough for all uses of an algorithm, while ensuring that the most efficient specialized form is automatically chosen when possible.
- Supporting several generic algorithms for the same purpose if there is no single algorithm that would provide the best efficiency for all inputs.
- Providing a precise characterization of the abstractions of a particular domain, such that the abstractions enable the definition of useful and efficient algorithms in that domain.

Generic programming can also be viewed as a program design and implementation methodology that separates data structures and algorithms through the use of abstract requirement specifications [4]. Generic algorithms are expressed in terms of properties of types instead of in terms of particular types. A generic algorithm can thus be reused with any type that embodies the necessary properties. The way to express such properties of types is done with *concepts* [16].

2. Concepts

The principal notion of generic programming is the notion of a *concept.* A concept is the formalization of an abstraction as a set of semantic and syntactic requirements on one or more types [9, 17]. When a type, or types, satisfy the requirements of a concept we say that those types *model* the concept. A concept typically consists of four different kinds of requirements [9]: *associated types, function signatures, semantic invariants* and *complexity guarantees.*

1. *Associated types* are generally defined to be auxiliary types related to the type that models the concept. The associated types of a concept specify mappings from the modeling type(s) to other collaborating types (for example, mapping from a container type to the type of its elements) [9].

2. *Function signatures* (or *valid expressions*) specify the operations that must be implemented for the modeling type. Calls to functions and operators defined with these signatures must be syntactically valid for any types that model the concept [4].

3. *Semantic invariants* are run-time properties of objects or values of the modeling and associated types that must always be true. The invariants often take the form of preconditions and post-conditions [4].

4. *Complexity guarantees* are maximum limits on the execution time complexities of the valid expressions, or limits on how much other resources their computation

uses [4].

In this thesis we focus on the first two components of concepts as Scala's type system cannot express semantic invariants or complexity guarantees.

A fundamental notion that we need in this thesis is the notion of *refinement* between concepts. A first concept is said to refine a second concept if the first concept includes all the requirements of the second concept. Thus, if a type is a model of a particular concept, it is also a model of all the concepts that the particular concept refines.

3. Constraining Type Parameters with Concepts

Concepts allow the concise expression of constraints on type parameters of generic algorithms [18]. When a generic method or class is defined, one may apply restrictions to its type parameters. The purpose of such restrictions is to guarantee that when the algorithm is *instantiated* with some concrete types, those same concrete types support all the methods that the generic method uses. In other words, those constraints are requirements that the types must satisfy, so as not to produce a compile-time error or a run-time error in the body of the algorithm.

Different languages provide different ways of representing concepts and using them to constrain type parameters. Consider the following Scala example:

```
def methodA[T <: SomeClass] (x: T) : T = {
    x.foo();
}
```

This particular example shows a method called `methodA` that takes the value `x` of type `T` as input and makes a call to the `foo` method defined in T or in one of its superclasses. The constraint on the type parameter T is expressed with the syntax: `[T <: SomeClass]`. This simply means that the method `methodA` requires that type

T must be a subtype of type SomeClass. In Scala, SomeClass can be a class or a *trait* (see Chapter IV), and the "type models a concept" relation is thus expressed as "type is a subtype of a trait."

Constraints allow methods to be separately type-checked. This means that at compile-time the type-correctness of a method's body is verified against the constraints of the method.

4. Goals of Generic Programming: Maximal Reuse and Efficiency

One of the key advantages of generic programming is maximal reuse of classes and methods resulting in an increase in programmer productivity. The principal language mechanism applied is type parameterization. Consider the two non-generic functions in Figure 1. The swapint method takes as input the parameters x and y of type int, and the swapstring method takes as input the parameters x and y of type String. Both int and String are specific types, both subtypes of the type Any. It would be time-consuming and counter-productive to write a swap method for every single distinct type.

```
def swapint(x: int, y:int) : Unit = {
  // This method swaps values x and y of type int.
  var temp = x; x = y; y = temp;
}
def swapstring(x: String, y:String) : Unit = {
  // This method swaps values x and y of type String.
  var temp = x; x = y; y = temp;
}
```

Fig. 1. Code redundancy.

A way to solve this problem is to write a single generic method that is parameterized, and that works for any type that supports assignment. This is guaranteed by the constraint that `T` must be a subtype of `Any`:

```
def swap[T <: Any] (x: T, y: T) : Unit = {
  // This method swaps values x and y of any type.
  var temp = x; x = y; y = temp;
}
```

This `swap` method can be used with any type that is a subtype of the datatype `Any` (`int`, `String` and many more).

The goal of generic programming is to generalize software components (classes and methods) so as to facilitate their reuse [19]. In software engineering, reusability is defined as the degree to which a software module can be used in more than one software system with very little or no modification [20]. If one single method is written such that it can be applied on a multitude of types that satisfy all of its requirements, then that method becomes potentially reusable in many context. Reusable classes and methods reduce implementation time, cost and testing which is essential to efficient software development. Generic programming has the potential to realize these reductions.

5. Problems in Supporting Generic Programming

The study done in [7] reports a general comparison of six programming languages (C++ [8], Generic Java [12], C#, Eiffel [10], ML [13, 21, 22], Haskell [11]) in their ability to support programming following the generic programming paradigm. The comparison was based on experiences collected from implementing a significant portion of the Boost Graph Library (BGL) [4] in each of the six languages.

One of the main results of the comparative study was that some mainstream object-oriented languages (like Java and C#) have difficulties expressing important aspects of generic programming. In particular, these include: *access to associated* *types*, *constraint propagation* (we also refer to this as the problem of repeated constraint), *type aliases* and *implicit instantiation.* We next discuss each of these issues in more detail.

a. Access to Associated Types

An associated type requirement in a concept expresses that a type name must be accessible from the "main" types modeling the concept. As an example, consider a graph type: its associated types could be its vertex type and its edge type. In other words, for every graph type, it must be possible to access its vertex and edge types. Many object-oriented languages such as Generic C#, Java or Eiffel, do not support associated types directly. Associated types can, however, be represented as type parameters of interfaces and classes. This may unfortunately lead to verbose code: associated types that are not properly encapsulated in generic interfaces, must be written explicitly every time the interface or class is referred to. For example, associated types become parameters of generic methods, whether the associated types are needed in the method's implementation or not.

An example of a language mechanism that is capable of expressing associated types is the *trait class* mechanism [7, 23]. This technique, introduced by Nathan Myers [23], is one of the essential techniques used in generic programming in C++ [4].

b. Repeated Constraints

For languages like C#, Java and Eiffel, that use subtyping as their mechanism to establish refinement, the problem of repeated constraints arise. Consider the following Generic `C#` example inspired by [7]:

```
interface A<T extends Someclass> { }
```

```
class B {
    public static <U, V extends A<U>>
}
```

Note the constraint on the type parameter `T` in `class A`. We do not repeat the constraint for type parameter `U` in `class B`. This seems reasonable, because the type-checker will have to check that `U` conforms to the restriction defined in `A`, when type-checking the constraint `V extends A<U>`. However, the code above fails to typecheck because the type-checker cannot use this information and thus `U` does not implement the class `Someclass`. It becomes necessary to repeat the constraint on the type `U` again in the definition of `class B`. The corrected version is as follows:

```
class B {
  public static <U,
                 V extends A<U extends Someclass> >
}
```

This constraint repetition leads to code verbosity as the number of constraint increases.

There is certain degree of correlation between the lack of associated types and the problem of repeated constraints: If a language does not directly support associated types, they may still be accessed in a generic function if they are added to the type parameter list. Unfortunately, this then implies that not only does one have to repeat all type arguments every time one refers to a generic interfaces, but one also has to repeat all constraints on those type arguments.

c. Type Aliases

Type aliasing is the ability of a programming language to allow a programmer to choose an alternative name for a type. This mechanism is very helpful, especially when the parameterization of components introduces long type names, because it improves code readability. For example, C++ uses the syntax `typedef` to realize type aliasing. In the following example the complicated type `property_map<G,PropertyTag>` can be simply referred to with `Map`:

```
typedef typename property_map<G, PropertyTag>::type Map;
```

The need to repeat complex types increases the probability of error. Type aliases come in handy to avoid long type name repetition and to allow type abstraction without losing static type accuracy [7].

d. Implicit Instantiation

Implicit instantiation is the ability of a compiler to use the types of the function arguments to automatically deduce the types that should be bound to type parameters during instantiation, without the programmer needing to explicitly specify those types.

Consider the example with Generic C# in Figure 2.

In the case of explicit instantiation, it is necessary to explicitly specify the types of the values `3` (`int`) and `6.0` (`double`). This task becomes more tedious as the number of type parameters increases. In the case of implicit instantiation, there is no need to specify the types of the arguments. Consequently, the verbosity is reduced. In languages that lack implicit instantiation (i.e. Eiffel), representing associated types as type parameters will worsen the wordiness of the code.

6. Scala Member Types

We investigated Scala so as to evaluate how well this new programming language handles the problems of generic programming that have previously been described. Before we engaged ourselves into this task, we had some assumptions:

```
public class someclass {
public static void go<T, U>(T a, U b) {
  if (a == b)
  {
   return true;
  }
  else
  {
   return false;
  }
 }
}
someclass bfs = new someclass;
//instantiating the class someclass bfs.go<int, double>(3, 6.0);
//explicit instantiation bfs.go(3,6.0); //implicit instantiation
```

Fig. 2. Implicit and explicit instantiation in C#.

- Scala has traits, so it was natural to research how this language can express concepts. Note that Scala traits are like Java interfaces, but they allow default implementation of the required methods. Traits in Scala are thus entirely unrelated to the C++ traits.

- Scala is often considered to be a "Java-like" language, it was then expected that implicit instantiation would be fully supported like it is in Java.

- Scala has member types that could potentially serve as associated types just like in C++. Consequently, it was also anticipated that member types would serve as a type aliasing mechanism.

These were the aspects we focused on in our evaluation.

CHAPTER III

SCALA BOOST GRAPH LIBRARY DESCRIPTION

The Boost Graph Library (BGL) [4] is an extensible and widely used C++ generic library that was developed following the generic programming paradigm. It uses similar forms of documentation and coding conventions as the (STL) [24]. Interfaces (concepts) are central to the BGL because similarly to the STL iterator concepts, BGL defines a set of graph concepts that enables graph algorithms to be written independently of the particular data types they operate on [4]. The BGL implements a large selection of generic graph algorithms and data structures.

In this thesis, we implemented a subset of the BGL in Scala so as to evaluate Scala's support for generic programming. This subset was chosen because it demonstrates many typical situations found in generic libraries, and thus serves as a stress-test for the implementation language. For example, all the chosen algorithms are highly parameterized, generic algorithms are called from within other generic algorithms, etc.

We implemented the following graph concepts from the BGL:

- *VertexListGraph*
- *IncidenceGraph*
- *EdgeListGraph*

Moreover we used the *Read Map* and *Read/Write Map* concepts (which are variants the *property map* concepts) so as to provide a convenient way to express relations between graph elements and domain-specific data [7]. For example, the edge of a graph may have a weight that may symbolize a distance or some type of quantity; property maps provide a way to express such quantities.

We implemented the following generic algorithms from the BGL:

- *Breadth First Search*
- *Depth First Search*
- *Dijkstra's Shortest Paths*
- *Bellman-Ford Shortest Paths*
- *Johnson's All-Pairs Shortest Paths*
- *Prim's Minimum Spanning Tree*

Additionally, the above generic algorithms use three auxiliary generic algorithms that are internal to the library:

- *Graph Search*
- *DFS Graph Search*
- *Relax*

The *Graph Search* algorithm is used by the *Breadth First Search* and *Dijkstra's Shortest Paths* algorithms. The *DFS graph search* algorithm is used by the *Depth First Search* algorithm. *Bellman-Ford Shortest Paths* algorithm relies on *Relax*. Finally, the *Dijkstra's Shortest Paths* is used by both *Johnson's All-Pairs Shortest* and *Prim's Minimum Spanning Tree* algorithms. All the generic graph algorithm that we use are parameterized with a graph type that must model both the *Incidence Graph* and the *Vertex List Graph* concepts.

Figure 3 taken from [7], illustrates the graph algorithms we implemented, their relationship and their ideal parameterization. A large rectangle corresponds to an algorithm and the attached small boxes represent its type parameters. An arrow from an algorithm to another signifies that the first algorithm uses the second one. Finally, an arrow from a type parameter to an un-boxed name means that the type parameter must model the concept.

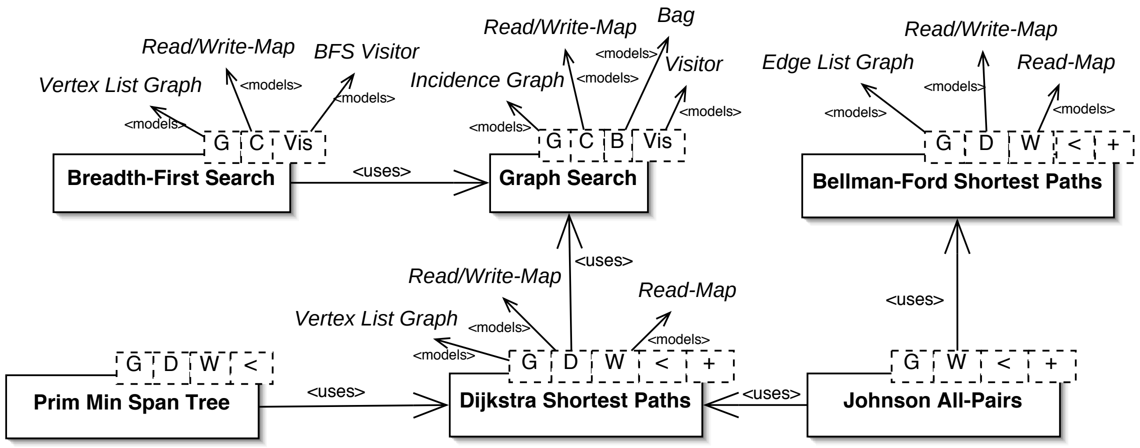

Fig. 3. Boost Graph Library generic algorithm organization and parameterization

In our implementation, we were able to respect the relationship between the algorithms. However, we were not able to respect the ideal parameterization of Figure 3. We had to add more parameters to the graph algorithms because of the issues of expressing associated types, described in Chapter V.

CHAPTER IV

THE SCALA LANGUAGE

1. Motivation

Scala has been developed between 2001 and 2004 by Martin Odersky in the programming methods laboratory at EPFL (École Polytechnique Féderale de Lausanne). He has co-designed and implemented the Pizza [25] and GJ [26] extensions of the Java language. Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant and type-safe way [14]. Additionally, Scala is said to smoothly fuse features of object-oriented and functional languages [14]. Scala targets the construction of components and component system, which is an elusive goal of the software industry [27]. Scala is the result of a research effort to develop better language support for component software. Scala was designed to be compatible with mainstream platforms such as Java and .NET.

2. Basic Language Features

In Scala, every value (including functions) is an object, so Scala can be characterized as a pure object-oriented language. The superclass of all classes is `scala.Any` which has two direct subclasses [14]. Its first subclass `scala.AnyVal` corresponds to primitive types such as integers and floating point numbers. Its second subclass `scala.AnyRefs` corresponds to reference classes (i.e user-defined classes) [14].

User-defined types can be defined with classes or traits. Classes are static templates that can be instantiated into many objects at runtime [14]. The example in Figure 4 illustrates how classes (and methods) are defined in Scala. The class `Point` defines two variables (`x` and `y`) and two methods, `move` and `toString`. The return

```
class Point(xc: Int, yc: Int) {
  var x: Int = xc;
  var y: Int = yc;
  def move(dx: Int, dy: Int): Unit = {
    x = x + dx;
    y = y + dy;
  }
  override def toString(): String = "(" + x + ", " + y + ")";
}
```

Fig. 4. Scala classes and method definition.

type `Unit` for the function `move` corresponds to the `void` type, say, in Java. This implies that function `move` does not return anything.

The example in Figure 5, taken from [14], illustrates the syntax of class instantiation and method calls. In the case of class instantiation, `pt` is an instance of the class

```
object Classes {
 def main(args: Array[String]): Unit = {
  val pt = new Point(1, 2); // class instantiation
  Console.println(pt);
  pt.move(10, 10);  // method calls
 }
}
```

Fig. 5. Scala class instantiation.

`Point`; `pt` may use all the methods defined in that class. Note the difference between *val* and *var* constructs: *val* defines a constant and *var* an updatable variable.

Similar to Java interfaces, traits are used to define interfaces of object types by specifying the signature of the methods that the object types must support [14]. Unlike java interfaces, Scala traits allow methods to have a default implementation.

Contrary to classes, traits cannot have constructors. Traits are what we will use in Scala to represent concepts. The following example illustrates the mechanism of traits:

```
trait Similarity {
  def isSimilar(x: Any): Boolean;
  def isNotSimilar(x: Any): Boolean = !isSimilar(x);
}
```

Traits collect a set of method signatures. Classes that inherit from a trait have to provide an implementation for all the signatures declared in the trait. This corresponds to the mechanism of *trait integration* which is analogous to implementing interfaces in Java. In the above example, any classes integrating this trait will have to implement the method `isSimilar`. Note that the method `isNotSimilar` has a default implementation which will be used if an integrating class does not provide an implementation for this method.

The following example shows how traits are integrated:

```
class Point(xc: Int, yc: Int) extends Similarity {
  // Trait integration
  var x: Int = xc;
  var y: Int = yc;
  def isSimilar(obj: Any) =
   obj.isInstanceOf[Point] &&
   obj.asInstanceOf[Point].x == x;
}
```

Trait integration is expressed using the `extends` keyword. Here, the method `isSimilar` is provided with an implementation. The method `isNotSimilar` is generated by the default implementation.

Class abstractions can be extended by subclassing. The sub-classing mechanism is very similar to that of Java or C++. Subclassing is also expressed with the keyword `extends`. The following example illustrates class extension and method overloading in Scala:

```
class Point(xc: Int, yc: Int) {
  val x: Int = xc
  val y: Int = yc
  def move(dx: Int, dy: Int): Point =
    new Point(x + dx, y + dy)
}
class ColorPoint(u: Int, v: Int, c: String) extends Point(u, v) {
  val color: String = c;
  def compareWith(pt: ColorPoint): Boolean =
    (pt.x == x) && (pt.y == y) && (pt.color == color)
  def move(dx: Int, dy: Int): ColorPoint =
    new ColorPoint(x + dy, y + dy, color);
}
```

Note that in this example, `Point` is the superclass of `ColorPoint` and `ColorPoint` is a subclass of `Point`. This implies that the class `ColorPoint` inherits all members from its superclass `Point`. In other words, `ColorPoint` inherits the values `x`, `y`, as well as the method `move`.

Also, note that `ColorPoint` implements two methods : `compareWith` (which determines whether two points are identical) and `move` (which returns a new point with new coordinates). The `move` method of the class `ColorPoint` overrides the `move` method of the class `Point`. Note that the `move` methods have different signatures: Covariant return types are allowed.

Access to the superclass's overridden member function is using the keyword `super`. The inherited method `move` (of the `point` class) can thus be accessed as: `super.move(...)` [14].

3. Features Relevant to Generic Programming

a. Polymorphic Methods

In Scala, methods can be parameterized with both values and types. Value parameters are enclosed in a pair of parenthesis, while type parameters are declared within a pair of brackets. The following example shows a recursive method `dup` that takes as input a value of arbitrary type `T` and an integer (`n`). This method returns a list containing `n` duplicates of the value of the type `T`:

```
 object PolyTest extends Application {
  def dup[T](x: T, n: Int): List[T] = {
    if (n == 0) Nil
    else x :: dup(x, n - 1)
    }
    // method call
  Console.println(dup[Int](3, 4));
  // method call with type inference
  Console.println(dup("three", 3));
}
```

Method `dup` is parameterized with type `T` and with the value parameters `x: T` and `n: Int`. When a generic method is called, the programmer can either specify the type arguments explicitly, or let the compiler infer them from the types of the actual arguments to the method. The first call to `dup` shows an example of specifying arguments explicitly and the second call illustrates an example of inferring type parameters [14].

b. Generic Classes

In addition to parameterized methods, Scala also supports parameterized classes. The example below shows how a stack class is implemented generically in Scala [10].

```
class Stack[T] {
  var elems: List[T] = Nil;
  def push(x: T): Unit = elems = x :: elems;
  def top: T = elems.head;
  def pop: Unit = elems = elems.tail;
}
```

This class `Stack` is represented internally as a list. When the stack class is instantiated, `elems` is an empty list. Method `push` appends an element at the beginning of the list `elems`. The methods `top` and `pop`, respectively, return the first element of the list and the rest of the list without its first element. The next example shows the use of this stack class.

```
  val stack = new Stack[Int]; // instance creation
  stack.push(1);
```

c. Variance Annotation

In Scala, subtyping of generic types remains *invariant*. This means that `Stack[T]` is a subtype of `Stack[S]` if and only if `S = T`. Scala allows however, type parameters annotations ("variance annotation") to control the subtyping behavior of generic types.

In *co-variant* subtyping [28], if `T` is a subtype of type `S` then `Stack[T]` is a subtype of `Stack[S]`. In *contra-variant*subtyping [28], if `T` is a subtype of type `S` then `Stack[S]` is a subtype of `Stack[T]`. Co-variant and contra-variant subtyping hold if type parameters are explicitly annotated with variance annotations.

The annotation `+T` declares that the type parameter `T` is co-variant. Co-variant subtyping is not type-safe in general, so a co-variant parameter can only be used in co-variant positions. Similarly, `-T` declares that the type parameter `T` is contra-variant. Contra-variant subtyping is not type-safe in general, so a contra-variant parameter can only be used in contra-variant positions. Method result type positions are categorized as co-variant, method argument positions and upper type parameter bounds are classified as contra-variant [27]. Scala's type system detects violations of these rules by keeping track of the positions where a type parameter is used [27].

Even though variance annotation is an interesting mechanism regarding generics, we have not identified it to be a key feature in our BGL implementation.

d. Upper and Lower Type Bounds

In Scala, the type parameters can be restricted using a lower or an upper bound. An upper type bound `T <: A` declares that the arbitrary type `T` is a subtype of type `A`. This declaration enables objects of type `T` to use methods declared in class or trait `A`. Consider the example in Figure 6. The method `findSimilar` works for only classes or traits that inherit from the trait `Similar`. Its inputs are some value `e` and a list. The method returns a boolean based on whether or not an instance of `e` is found. The method `isSimilar` is made usable for the object `e` of type `T` with the upper type bound expression `T <: Similar`.

A lower bound `T >: A` expresses that type `T` is a supertype of type `A`. We have found no significant use of lower bounds in our BGL implementation.

e. Abstract Type Members

Abstract types members are types whose identity is not precisely known. Consider the following code:

```
abstract class AbsCell {
  type T;
  val init: T;
  private var value: T = init;
  def get: T = value;
  def set(x: T): unit = { value = x}
}
```

```
trait Similar {
  def isSimilar(x: Any): Boolean;
}

case class MyInt(x: Int) with Similar {
  def isSimilar(m: Any): Boolean =
    m.isInstanceOf[MyInt] &&
    m.asInstanceOf[MyInt].x == x;
}

object UpperBoundTest with Application {
  def findSimilar[T <: Similar](e: T, xs: List[T]): Boolean = {
    if (xs.isEmpty) false
    else if (e.isSimilar(xs.head)) true
    else findSimilar[T](e, xs.tail);
    }

  val list: List[MyInt] = List(MyInt(1), MyInt(2), MyInt(3));
  Console.println(findSimilar[MyInt](MyInt(4), list));
}
```

Fig. 6. Type bounds in Scala.

The `AbsCell` class does not define any type or value parameters. Instead it has an abstract type member `T`. Instances of `AbsCell` can be created by binding all abstract members to concrete definitions—including the type member `T`. The following example shows how an abstract class is instantiated, and used, with respect to the previous code snippet:

```
val cell = new AbsCell {type T = int; val init = 1;}
cell.set(cell.get * 2)
```

CHAPTER V

BGL IMPLEMENTATION IN SCALA

To realize generic programming in Scala, we need to represent essential notions of generic programming with the language constructs offered by Scala. For this we make use of Scala's classes, abstract classes, traits, polymorphic methods, and abstract types members. In particular, we use:

- *traits* to represent concepts
- *inheritance* between traits to represent refinement between concepts
- `<:` symbol to represent that a type models a particular concept and to represent constraints on type parameters and associated types
- *member types* to represent associated types
- *parameterized methods* to represent generic algorithms

In this chapter, we describe how we applied these Scala features in the BGL implementation, and analyze their suitability for generic programming.

Note that we have used the Scala 1.4.0.3 compiler [14] to realize our implementation. The changes that have been made to later Scala versions should not significantly affect our BGL implementation.

1. Concepts as Traits

The means to group a set of constraints, valid expressions and associated types, is encapsulating them into a Scala trait, as function signatures and member types, respectively. The three traits that represent the graph concepts described in Chapter III serve as an example (see Figure 7).

```
trait VertexListGraph {
  type Vertex;
  type VertexIterator <: Iterator[Vertex];
  def vertices: VertexIterator;
  def num_vertices: int;
}

trait IncidenceGraph {
  type Vertex;
  type Edge <: GraphEdge;
  type OutEdgeIterator <: Iterator[Edge];
  def out_edges(v: Vertex): OutEdgeIterator;
  def out_degree(v: Vertex): int;
}

trait EdgeListGraph {
  type Edge <: GraphEdge;
  type EdgeIterator <: Iterator[Edge];
  def edges: EdgeIterator;
}
```

Fig. 7. Scala concepts.

These graph concepts contain method signatures and member types. As one may observe, none of the methods have an implementation; classes that inherit from these traits must provide an implementation for the methods. The associated types (`Vertex`, `Edge`, `EdgeIterator` and `OutEdgeIterator`) are directly expressed as member types of the traits representing the graph concepts. Furthermore, member types can be constrained with subtype constraints which can be seen, for example, in the `Edge` member of the `IncidenceGraph` trait: `Edge` type inherits from the `GraphEdge` trait shown in Figure 8.

```
trait GraphEdge {
  type Vertex;
  def source: Vertex;
  def target: Vertex;
}
```

Fig. 8. GraphEdge trait.

2. Concept Refinement in Scala

Concept refinement is represented using inheritance between traits. BGL defines a bidirectional graph concept [4]. An incidence graph is a directed graph that can be represented by a list of all the outgoing edges for each vertex of the graph. On the other hand, a bidirectional graph is an incidence graph that can also be represented by a list of all the incoming edges for each vertex of the graph. This implies that the incidence graph concept is a refinement of the bidirectional graph concept. Consequently, with respect to the definition of the `IncidenceGraph` trait in Figure 7, we may define the `BidirectionalGraph` trait as follows:

```
trait BidirectionalGraph extends IncidenceGraph  {
  type InEdgeIterator <: Iterator[Edge];
  def in_edges(v: Vertex): InEdgeIterator;
  def int_degree(v: Vertex): int;
  def  degree(e: Edge): int;
}
```

Observe that the `BidirectionalGraph` trait inherits all the types and methods of `IncidenceGraph` trait.

3. Modeling Relation in Scala

Using Scala, we can express that a type models a concept by inheriting from the trait corresponding to the concept. The example in Figure 9 shows parts of the implementation of an adjacency list data structure (`adjacency_list`) that models the `VertexListGraph`, `IncidenceGraph`, and `EdgeListGraph` concepts. Since

```
class adjacency_list extends VertexListGraph
                     with IncidenceGraph
                     with EdgeListGraph {
  type Vertex = int;
  type Edge = adj_list_edge {type Vertex = int};
  type VertexIterator = Iterator[int];
  type OutEdgeIterator = Iterator[Edge];
  type EdgeIterator = Iterator[Edge];

  def vertices: Iterator[int] = {
    vertices_.Iterator;
  }
   ...
}
```

Fig. 9. Adjacency list class.

the `adjacency_list` models all the graph concepts presented in Figure 7, it automatically inherits all the associated types of those concepts, namely `Vertex`, `Edge`, `VertexIterator`, `OutEdgeIterator`, and `EdgeIterator`, to which it must bind concrete types.

Observe that we equate concrete types `int`, `adj_list_edge {type Vertex = int}`, `Iterator[int]`, `Iterator[Edge]`, and `Iterator[Edge]` to the inherited member types `Vertex`, `Edge`, `VertexIterator`, `OutEdgeIterator`, and `EdgeIterator`, respectively. Similar to type members, classes inheriting from the graph concepts will have to provide implementations for the methods that those graph concepts require. In Figure 9, we only show the implementation of the vertices method, required by the `VertexListGraph` concept. In our full implementation of the adjacency list data structure, we provide definitions for all required methods.

The main data structure that we used to represent graphs in our BGL implementation was the adjacency list. In order to remain consistent with previous implementation of the BGL [7], we have chosen an implementation where the vertices are of type `int`.

4. Generic Algorithms

Scala does not support “free-standing” functions. Instead, all functions are members of some class or object. Scala supports both parameterized classes and parameterized methods. Generic algorithms can thus be represented either as non-generic methods of generic classes, or as generic methods of non-generic classes. We chose to use the latter: we represent generic algorithms with parameterized methods. The former choice would have ruled out Scala's type inference mechanism discussed later. Figure 10 shows an example of a generic algorithm in Scala. We name the class enclosing the parameterized function to indicate the algorithm: here, `breadth_first_search`. By convention [7], we name the parameterized function as `go`.

To call a generic algorithm means creating an instance of the enclosing class, and invoking the `go` method for this object. The object of the `breadth_first_search` class has, however, no role in the implementation of the algorithm. Hence, we would like to make `go` a static method. Unlike in Java, static methods are not supported in Scala. However, the Scala programmer can declare a method in a singleton object. In that way, method call would simply be using the object name. Consider the example in Figure 11. Observe that using a singleton object skips the instantiation process.

We could have made the `breadth_first_search` class an object so that we could have accessed the generic method `go` with the syntax `breadth_first_search.go`. However, the version of Scala that we have used for our implementation (version 1.4.0.3) did not allow us to put our generic method within singleton objects. The

```
class breadth_first_search {
  def go[Vertexb,
         Edgeb <: GraphEdge {type Vertex = Vertexb;},
         VertexIteratorb <: Iterator[Vertexb],
         OutEdgeIteratorb <: Iterator[Edgeb],
         Graphb <: VertexListGraph with IncidenceGraph
                with EdgeListGraph
                {type Vertex = Vertexb; type Edge = Edgeb;
                 type OutEdgeIterator = OutEdgeIteratorb;
                 type VertexIterator = VertexIteratorb},
         Visb <: Visitor
                {type Graph = Graphb ; type Vertex = Vertexb;
                type Edge = Edgeb;},
         ColorMapb <: ReadWritePropertyMap
                {type Key = Vertexb; type Value = int},
         QueueTypeb <: Buffer{type Value = Vertexb}]
  (g: Graphb ,s: Vertexb, vis: Visb ,color: ColorMapb ): Unit = {
  .....
  }
}
```

Fig. 10. Breadth-first-search.

invocation of the breadth-first-search algorithm is thus as follows:

```
val bfs = new breadth_first_search;
bfs.go[Vertexb, Edgeb,
       VertexIteratorb, OutEdgeIteratorb,
       Graphb, PrintingVisitorb,
       Colormapb, Queuetypeb]
     (g, 3, visitorb, colorb);
```

Note that in addition to the method parameters we also explicitly pass the type parameters to the breadth-first-search algorithm. In general, there are two ways to invoke generic functions: implicit and explicit instantiation. With implicit instantiation, type parameters are inferred from the type of the method's arguments. With

```
// method in a class
class bfs1 {
  def go1: Unit = {}
}
// method in a singleton objects
object bfs2 {
  def go2: Unit = {}
}

object testing extends Application {
  // class instantiation
  var bfs1instance = new bfs1;
  bfs1instance.go1;
 // using the singleton object
  bfs2.go2;
}
```

Fig. 11. Singleton objects versus classes.

explicit instantiation, the programmer has to explicitly state the type parameters. With implicit instantiation, the call in Figure 10 could be written as:

```
breadth_first_search.go(g, 3, visitorb, colorb);
```

In this case, type parameter inference, should deduce the types `Vertexb`, `Edgeb`, `VertexIteratorb`, `OutEdgeIteratorb`, `Graphb`, `PrintingVisitorb`, `Colormapb` and `Queuetypeb` from the types of the arguments `g`, `3`, `visitorb` and `colorb`, and hence avoid specifying them explicitly. Unfortunately, we were not able to make use of implicit instantiation. In Section 5 of this chapter and in Chapter VI, we discuss how Scala's type inference mechanism was not adequate for our purposes.

```
class breadth_first_search {
 def go[Vertexb, Edgeb <: GraphEdge {type Vertex = Vertexb;},
   VertexIteratorb <: Iterator[Vertexb],
   OutEdgeIteratorb <: Iterator[Edgeb],
   Graphb <: VertexListGraph with IncidenceGraph
             with EdgeListGraph
                   {type Vertex = Vertexb; type Edge = Edgeb;
                    type OutEdgeIterator = OutEdgeIteratorb;
                    type VertexIterator = VertexIteratorb},
   Visb <: Visitor {type Graph = Graphb; type Vertex = Vertexb;
                    type Edge = Edgeb;},
   ColorMapb <: ReadWritePropertyMap
                       {type Key = Vertexb; type Value = int;}]
  (g: Graphb, s: Vertexb, vis: Visb, color: ColorMapb): Unit = {
  var Q: queue {type Value = Vertexb;}
       = new queue {type Value = Vertexb;};
  var u_iter: VertexIteratorb = g.vertices;
  val ColorValue = new ColorValue;
  val gs = new graph_search;
    while(u_iter.hasNext) {
      var u: Vertexb = u_iter.next;
      vis.initialize_vertex(u, g);
      color.set(u, ColorValue.white);
    }
  gs.graph_search[Vertexb, Edgeb, VertexIteratorb, OutEdgeIteratorb,
    Graphb, Visb, ColorMapb, queue {type Value = Vertexb;}]
    (g, s, vis, color, Q);
}
```

Fig. 12. Scala breadth-first-search.

5. Analysis of Generic Programming in Scala

The mapping from generic programming notions to Scala language constructs suggests a direct mechanism for implementing generic libraries. We encountered, however, several obstacles when applying the mapping to implement BGL. We illustrate these problems by focusing on the implementation of the breadth-first-search algorithm as it demonstrates all the interesting aspects of the difficulties encountered.

Figure 12 shows the full implementation of the breadth-first-search algorithm.

The implementation is clearly not ideal with respect to generic programming. In particular, one notices the abundant type parameters and constraints on them. By contrast, rigorously following the mapping described above would have given the implementation shown in Figure 13. This implementation, however, is not valid. In what follows, we explain the sources of discrepancy between the practical and ideal implementations.

```
object breadth_first_search{
   def go[Graphb <: VertexListGraph
                 with IncidenceGraph
                 with EdgeListGraph
          Visb <: Visitor {type Graph = Graphb},
          ColorMapb <: ReadWritePropertyMap
                       {type Key = Graphb.Vertex; //Incorrect syntax
                        type Value = int}]
   (g: Graphb, s: Graphb.Vertex,
    vis: Visb, color: ColorMapb): Unit = {
  var Q: queue{type Value = Graphb.Vertex;} =
         new queue{type Value = Graphb.Vertex;};
  var u_iter: Graphb.VertexIterator = g.vertices;
  val ColorValue = new ColorValue;
  val gs = new graph_search;
   while(u_iter.hasNext)
    {var u: Graphb.Vertex = u_iter.next;
     vis.initialize_vertex(u, g);
     color.set(u, ColorValue.white);}
  gs.graph_search(g,s,vis,color,Q); //call to  auxiliary method.
}
```

Fig. 13. Ideal breadth-first-search.

a. Accessing Associated Types

In an ideal representation of the breadth-first-search algorithm (generic `go` method), the only type parameters would be `Graphb`, `Visb` and `ColorMapb` which correspond to three out of four argument types of this generic algorithm. The fourth argument type (which is the vertex type of the graph `Graphb`) is determined by the type `Graphb`. This explains why the fourth argument type is represented as `Graphb.Vertex`.

However, our actual implementation contains type parameters `Vertexb`, `Edgeb`, `VertexIteratorb`, and `OutEdgeIteratorb`. This correspond to the associated types in `VertexListGraph`, `IncidenceGraph`, and `EdgeListGraph` concepts. We need to access these associated types in the constraints of the `go` method in order to establish several type equivalences. Ideally, we would access the associated types directly with Scala's "dot notation" as shown in Figure 13. For example, the expression

```
type key = Graphb.Vertex;
```

would establish that the `key` associated type in the `ColorMap` concept is equal to the `Vertex` associated type in `VertexListGraph`.

Unfortunately, the Scala language does not allow such use of the syntax with the expression `Graphb.Vertex` because of the context in which it occurs. Scala's "dot notation" (`p.t`) is allowed when `p` is a *path*.

The syntax `Graphb.Vertex` implies that `Graphb` is a path and `Vertex` a type [29]. Unfortunately, `Graphb` is not a path in the context in which we are using it. A path can only be one of the following [29]:

- (1) `C.this`, where `C` is the name of the class directly enclosing the reference.
- (2) `p.x` where `p` is a path and `x` is a stable member (member introduced by value or object definition) of p

- (3) `C.super.x` or `C.super[M].x` where `C` is the name of the class directly enclosing the reference and `x` references a stable member of the super class or designated parent class `M` of `C`.

`Graphb` does not qualify in any of these three path definitions. Additionally, we may not use `Graphb.this.Vertex` to refer to the `Vertex` type because `Graphb` is not the enclosing class. As a result, `Graphb.this` is not a path. In our situation, the enclosing class is `breadth_first_search`. Consequently, the syntax `breadth_first_search.super.Vertex` would produce a compiler error because the class `breadth_first_search` does not have a `super` class (the enclosing class of the generic algorithm (`breadth_first_search`), does not inherit from anything). Similarly, the expression `Graphb.VertexIterator` would not be allowed for the same reasons as the expression `Graphb.Vertex`.

b. Associated Types as Type Parameters

Due to the lack of flexibility of the problem of paths in Scala, we had to maintain `Vertexb, Edgeb, VertexIteratorb`, and `OutEdgeIteratorb` as "extra" type parameters of the breadth-first-search algorithm (Figure 12). In this way, we are equating two associated types to one another by using an extra parameter. For example:

- The associated type `Vertex` of the trait `VertexListGraph` is equated to the "extra" type `Vertexb`.
- The associated type `Vertex` of the trait `Visitor` is also equated to the "extra" type `Vertexb`.
- In conclusion, we have established equality between both `Vertex` associated types of the traits `VertexListGraph` and `Visitor`.

In short, since we could not directly write:

```
A.Vertex = B.Vertex
```

we had to create an extra parameter type `Verteb` and use that to express the equality transitively, as follows:

```
A.Vertex = Vertexb; B.Vertex = Vertexb;
```

Unlike, say, member typedef in C++ [8], Scala's member types are not bound statically to the enclosing class, but instead can vary from object to object. They are thus like *virtual types* in this sense [30]. This is a potential source of more equality constraints which did not, however, manifest notably in our BGL implementation. There are, however, generic algorithms where this issue arises. In practice, the effect is that the equality of the same associated type accessed via two different objects of the same type is not automatically guaranteed. Consider, the following example:

```
abstract class A {
 type Vertex;
}
 var a1 = new A {type Vertex = int}
 var a2 = new A {type Vertex = float}
```

The objects `a1` and `a2` are both instances of the abstract class `A`, however their type `Vertex` are different.

c. Renaming Type Parameters

A minor inconvenience we encountered was the fact that we had to rename some of our type parameters, in order to avoid cyclic type references. This explains why the type parameter for vertices was named `Vertexb`, instead of `Vertex` in the `go` method. A cyclic reference arises if we try to write the constraint as follows:

```
class breadth_first_search {
  def go[Vertex,
         Edgeb <: GraphEdge
                {type Vertex = Vertex;},// ERROR!
         ......
         ]}
```

The corrected version renames the type parameter in conflict:

```
class breadth_first_search {
  def go[Vertexb,
         Edgeb <: GraphEdge {type Vertex = Vertexb;}, //COMPILES.
          ....
        ]
     ...
}
```

d. Inadequate Support for Implicit Instantiation

The call to method `go` demanded a verbose declaration depicted in Figure 14. The ideal representation would have been as in Figure 15. Unfortunately, Scala's type inference mechanism fails in that ideal context. This is because the type parameters `Vertexb`, `Edgeb`, `VertexIteratorb`, and `OutEdgeIteratorb` of the `go` method do not appear in types of the arguments of the method. They only appear in constraints of other type parameters. The following example explains why in such a case type inference and thus implicit instantiation fails:

```
trait GraphEdge[T] {}
class Edge extends GraphEdge[int] {}
class algorithm {
  def go1[T, U <: GraphEdge[T]](b: U): Unit = {}
  def go2[T, U <: GraphEdge[T]](c:T, b: U): Unit = {}
}
object det1 with Application {
  var x: Edge = new Edge ;
  var y:int = 3;
```

```
val colorb = new hash_property_map
  {type Key = int; type Value = int;};
val visitorb = new printing_Visitor {
                 type Graph = adjacency_list;
                 type Vertex = int;
                 type Edge = adj_list_edge {type Vertex = int}; };
type Vertexb = int; type Edgeb = adj_list_edge{type Vertex =int};
type VertexIteratorb = Iterator[Vertexb];
type OutEdgeIteratorb = Iterator[Edgeb];
type Graphb = adjacency_list;
type PrintingVisitorb
= printing_Visitor {type Graph =adjacency_list;
                     type Vertex = int ;
                     type Edge = adj_list_edge {type Vertex = int}};
type Colormapb = hash_property_map
               {type Key = int ; type Value = int;};
type Queuetypeb =  queue{type Value = int; };
val bfs = new breadth_first_search;
 bfs.go[Vertexb, Edgeb, VertexIteratorb,
        OutEdgeIteratorb, Graphb, PrintingVisitorb, Colormapb]
    (g, 3, visitorb, colorb);
```

Fig. 14. Breadth-first-search instantiation process.

```
  var algo  = new algorithm ;
  algo.go1[int, Edge](x);  //(Case 1)Compiles
  algo.go2(y,x);  //(Case 2)Compiles. Successful Implicit instantiation.
  algo.go1(x); //(Case 3) Error !
}
```

In `Case 1`, method `go1` is explicitly instantiated; all the type parameters are explicitly defined. In `Case 2`, implicit instantiation works because every type parameter is directly used as an argument type of the method `go2`. In `Case 3`, implicit instantiation does not works because every type parameter is not directly used as a parameter type of the method `go1`. The type-checker cannot deduce a value for the type parameter `T`. `T` only occurs in the constraint `U <: GraphEdge[T]`, and even though `x` has type

```
val colorb = new hash_property_map
                {type Key = int ;
                 type Value = int;};
val visitorb = new printing_Visitor
                  {type Graph = adjacency_list ;
                   type Vertex = int ;
                   type Edge = adj_list_edge {type Vertex = int}; };
val bfs = new breadth_first_search; bfs.go(g, 3, visitorb, colorb);
// Error!
```

Fig. 15. Breadth-first-search ideal instantiation process.

`Edge` and `Edge` inherits from `GraphEdge[int]`, the Scala type-checker cannot infer that `T` is `int`.

CHAPTER VI

DISCUSSION AND CONCLUSION

This sections summarizes our findings regarding support for generic programming in Scala. We first focus on the two problems identified in the previous section that we see as being a hindrance to effective generic programming in Scala: lack of access to member types and lack of full implicit instantiation. Secondly, we discuss the three advantages that have facilitated our programming experience: constraint propagation, compound types, and support for type aliasing. Thus, this chapter has the following progression:

- Member types in Scala
- Incomplete support for implicit instantiation
- Constraint propagation support
- Compound types
- Type aliasing

1. Member Types in Scala

a. Lack of Access to Scala Member Types

Member types can encapsulate associated types. From the generic programming perspective, this is an improvement over Java or C#, where a separate type parameter is needed for each associated type. However, when accessing member types from the constraints of generic algorithms, we need to translate the member types into type parameters. This brings us effectively to the same situation as with Java or C#. This was because we needed to express equality constraints between associated types of different concepts. Expressing such constraints is only possible between a type parameter and a member type, not between two member types directly.

b. Repeated Constraints

In principle, traits can enclose associated types as member types. We need to translate those member types into type parameters of generic algorithms. This implies that we need to also repeat all the constraints on those associated types. Consider Figure 16 that represents the type parameters of the `go` method of the `breadth_first_search` object. The problem that can be discerned is the one concerning repeated constraints.

```
def go[Vertexb,
       Edgeb <: GraphEdge {type Vertex = Vertexb;},
       VertexIteratorb <: Iterator[Vertexb],
       OutEdgeIteratorb <: Iterator[Edgeb],
       Graphb <: VertexListGraph
              with IncidenceGraph
              with EdgeListGraph
                    {type Vertex = Vertexb; type Edge = Edgeb;
                     type OutEdgeIterator = OutEdgeIteratorb ;
                     type VertexIterator = VertexIteratorb},
       Visb <: Visitor
                {type Graph = Graphb ;
                 type Vertex = Vertexb;
                 type Edge = Edgeb;
       ColorMapb <: ReadWritePropertyMap
                     {type Key = Vertexb; type Value = int}]
```

Fig. 16. Type parameters of the go method.

Since we were unable to deduce the associated types from the graph type `Graphb`, we had to introduce arbitrary types (namely `Vertexb`, `Edgeb`, `VertexIteratorb`, and `OutEdgeIteratorb`) to which we attributed constraints. Thus, we had to repeat the constraints already stated in the graph concepts for the corresponding associated types on the extra type parameters (namely `Edgeb`, `VertexIteratorb` and `OutEdgeIteratorb`). The constraint

```
Edgeb <: GraphEdge {type Vertex = Vertexb;}
```

is an example of this.

In principle, we can encapsulate constraints on associated types of traits. However, when we translate the associate types to type members in generic algorithms, these constraints become a proof of obligation, rather than an assumption.

2. Implicit Instantiation Failure

Implicit instantiation is supported in Scala depending on the context in which it is used. As explained previously, implicit instantiation in Scala only works if each type parameter of the generic method is found in the type of one of the arguments of that same generic method. This scenario occurred frequently in our BGL implementation. All associated types were expressed as extra type parameters of generic algorithms, and these type parameters typically only appeared as constraints of other type parameters, not in types of the method parameters. For example, in the case of the breadth-first-search algorithm, type inference was inadequate for the more complex parametrization common in generic libraries. Compared to other object-oriented languages, C#'s type inference has similar limitations as Scala. Java, on the other hand would be capable of handling the above described scenarios.

The lack of access to associated type lead us to add them as type parameters of generic methods. Doing so resulted in three problems: Code verbosity increase, type constraint repetition (leading to more code verbosity increase), and implicit instantiation failure (leading to more code verbosity increase (Figure 17).

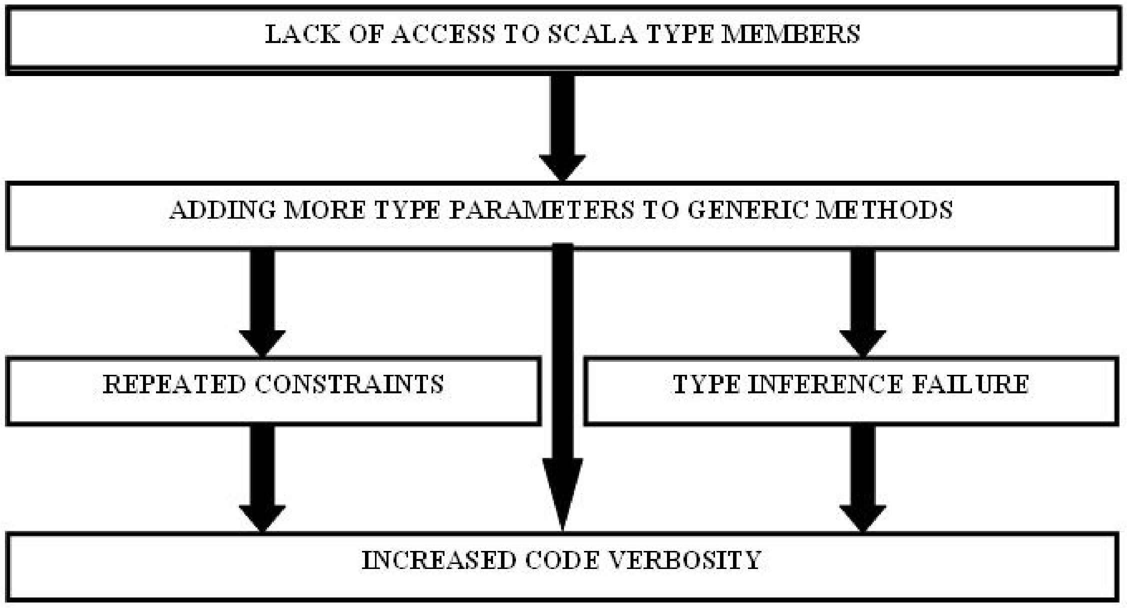
LACK OF ACCESS TO SCALA TYPE MEMBERS
ADDING MORE TYPE PARAMETERS TO GENERIC METHODS
REPEATED CONSTRAINTS
TYPE INFERENCE FAILURE
INCREASED CODE VERBOSITY

Fig. 17. Problem flow of generic programming in Scala

3. Constraint Propagation Support

We had to repeat constraints on the added types `Edgeb`, `VertexIteratorb`, and `OutEdgeIteratorb`. Consider the code snippet in Figure 18. We do not need to

```
Graphb <: VertexListGraph
             with IncidenceGraph
             with EdgeListGraph
                   {type Vertex = Vertexb; type Edge = Edgeb;
                    type OutEdgeIterator = OutEdgeIteratorb ;
                    type VertexIterator = VertexIteratorb}
```

Fig. 18. Compound type mechanism

repeat the constraints on the types `Edge`, `VertexIterator`, and `OutEdgeIterator` while assigning them to the types `Edgeb`, `VertexIteratorb`, and `OutEdgeIteratorb`, respectively. The constraints of those types were already established in the graph concept definition (see Figure 7). For example, we don't need to specify that the `VertexIterator` inherits from the `Iterator[Vertex]` type by writing again:

```
type VertexIterator <: Iterator[Vertex];
```

4. Compound Types

We found Scala's compound types useful. The inability to simultaneously constrain a type parameter with more than one interface was reported to be a problem in [7], requiring the explicit introduction of compound interfaces, such as (for example) `VertexListAndIncidenceAndEdgeListGraph`. In Scala, this is unnecessary. Scala allows parameters to have several constraints. By observing the `Graphb` type declaration in Figure 10, it can be seen that the graph type (`Graphb`) inherits from the traits `VertexListGraph`, `IncidenceGraph` and `EdgeListGraph` at the same time by using the `with` connective and grouping all the associated types within a single pair of curly braces (see Figure 18). Realizing this grouping using the appropriate syntax has lead us to obtain a compound type equivalent to a type `VertexListAndIncidenceAndEdgeListGraph`.

5. Type Aliases

Scala supports type aliasing. Since the parametrization of components introduces long type names, it was often convenient to use a shorter name to refer to them. For example, here we name a complex type with the type alias `PrintingVisitorb`.

```
type PrintingVisitorb
   = printing_Visitor
      {type Graph = adjacency_list; type Vertex = int;
       type Edge = adj_list_edge {type Vertex = int}};
```

Type aliasing helps reduce code verbosity and facilitates the abstraction of the actual type without losing type accuracy.

6. The Scala Experience

The Scala syntax is easy to pick up for programmers familiar with Java and C#. With respect to generics, Scala uses object-oriented techniques as primary mechanism to build abstractions and establish concept modeling relations. Compared to Eiffel, Java, and C#, Scala provides support for a broader version of generic programming with type members (used as associated types) and the type aliasing mechanism.

The BGL implementation in Scala resulted in a few surprises. Contrary to our initial expectations, member types support the expression of associated types in a limited way. Additionally, support for implicit instantiation is only partial. In our experiment the weak support for associated types manifested as extra type parameters in generic functions. This adds to the verbosity of generic code as every reference to generic software component (class, trait, or function) must explicitly list all its type parameters. In particular, the extra type parameters are such that they only occur in constraints of other type parameters. Scala cannot infer such type parameters in function calls, which prevents implicit instantiation for many generic algorithms. In fact, implicit instantiation does not work for any of the BGL algorithms we implemented. The combination of the above factors increased the verbosity of the entire implementation. Scala's type aliasing mechanism turned out to be beneficial in generic programming. Member types in Scala had less impact in generic programming than we had anticipated. Still, Scala remains a language with considerable support of generic programming, and as it is constantly evolving, there is still room for it to become more powerful with respects to generics.

7. Conclusion

Generic programming is a methodology for designing and implementing reusable libraries of software components. The inclusion of the Standard Template Library in the C++ standard library has propulsed the popularity of generic programming within C++. Many other mainstream languages offer varying forms of generics as well. However, several languages have been reported to lack some essential features necessary to fully embrace the generic programming paradigm [7]. In particular languages such as C# [9, 7], Eiffel [10] and Java [12] lack direct support for associated types. Instead these languages would add associated types to the parameter list of generic functions to provide access to them. Unfortunately, this technique results in cluttered and verbose code. To make matters worse, this practice of representing associated types prevents implicit instantiation of generic methods in, e.g., C# [9, 7] and Eiffel [10].

In this thesis we evaluated how Scala, a new object-oriented language can support generic programming. In particular, Scala supports member types that can in principle be used as associated types. Our experiment consisted in implementing a subset of a state-of-art generic library (the Boost Graph Library) to analyze a wide range of generic programming techniques in Scala. Determining the impact of member types and the degree of support to implicit instantiation turned out to need careful studying. We report on these aspects in detail and point out how Scala member types do not fully suffice for generic programming. We cannot entirely get rid of the problems with accessing associated types. Member types only offer a partial solution and at times we still need to resort to representing associated types using type parameters. Moreover, implicit instantiation is only possible in Scala under the condition that each element of the type parameter list corresponds to types of the arguments of the function, a condition that does not hold in typical generic libraries. Therefore, we had to explicitly instantiate calls to many generic functions.

Otherwise, we found that Scala's support for generic programming was adequate. Scala uses the inheritance mechanism to establish the concept modeling relation. Multi-type concepts and multiple constraints are supported. Scala's support for type aliasing also turned out to be very practical throughout our implementation.

REFERENCES

[1] "The Graph Template Library," 2006, Available at www.fmi.uni-passau.de/Graphlet/GTL.

[2] B. Ryder, M. Soffa, and M. Burnett, "The impact of software engineering research on modern progamming languages," in *ACM Transactions on Software Engineering and Methodology (TOSEM)*, October 2005, vol. 14, pp. 431–477.

[3] A. Stepanov, "The Standard Template Library," *Byte Magazine*, October 1995, Available at http://www.byte.com/art/9510/sec12/art3.htm.

[4] J. G. Siek, L. Lee, and A. Lumsdaine, *The Boost Graph Library*, Upper Saddle River, NJ: Addison-Wesley, 2001.

[5] D.E Knuth, *Stanford GraphBase: a platform for combinatorial computing*, New York, NY: ACM Press, 1997.

[6] "The Matrix Template Library," 2006, Available at http://osl.iu.edu/research/mtl/.

[7] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock, "A comparative study of language support for generic programming," in *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, Anaheim, CA, USA, 2003, ACM and SIGPLAN, vol. 38, pp. 115–134.

[8] B. Stroustrup, *The C++ Programming Language*, Upper Saddle River, NJ: Addison-Wesley, 3rd edition, 1997.

[9] J. Järvi, J. Willcock, and A. Lumsdaine, "Associated types and constraint propagation for mainstream object-oriented generics," in *Proceedings of the 20th annual ACM SIGPLAN Conference on Object oriented programming systems languages and applications*, San Diego, CA, USA, 2005, ACM and SIGPLAN, pp. 1–19.

[10] D. Wylder, "Introduction to Eiffel," *Linux Journal*, , no. 14, pp. 1–9, 1995.

[11] S. Thompson, *Haskell: The Craft of Functional Programming*, Upper Saddle River, NJ: Addison-Wesley, 2nd edition, 1999.

[12] G. Bracha, "Generics in Java programming language," July 2004, Available at java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[13] C. R. Spooner, "The ML approach to the readable all-purpose language," in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, New York, NY, USA, April 1986, vol. 8 of *2*, pp. 215–243.

[14] M. Odersky, "The Scala programming language," 2002, Available at http://scala.epfl.ch/.

[15] M. Jazayeri, R. Loss, D. Musser, and A. Stepanov, "Generic programming," in *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, April 1998.

[16] D. Musser, "Generic programming," Online, May 2003, Available at www.cs.rpi.edu/ musser/gp/.

[17] D. Musser, "Concepts in software engineering," Online, 2006, Available at www.cs.rpi.edu/research/gpg/.

[18] J. Siek and A. Lumsdaine, “Essential language support for generic programming,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Il, USA, 2005, ACM and SIGPLAN, pp. 73–74.

[19] “Boost C++ libraries,” Online, 2006, Available at http://boost.org/.

[20] Institute of Electrical and Electronics Engineers, “IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries,” 1990, New York, NY.

[21] J. C. Mitchell and R. Harper, “The essence of ML,” in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, 1986, vol. 8, pp. 215–243.

[22] “Standard ML,” Online, 1998, Available at www.smlnj.org/sml.html.

[23] N. Myers, “Traits : a new and useful template technique,” Tech. Rep., Borland C++ Report, June 1995, Available at http://www.borland.com/borlandcpp/news/

[24] H. M Deitel, *C++ How To Program*, Upper Saddle River, NJ: Prentice Hall, 5th edition edition, 2005.

[25] M. Odersky and P. Wadler, “Pizza into Java: Translating theory into practice,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 1997, pp. 145–159.

[26] D. M. and Alexander Stepanov, “A library of generic algorithms in ADA,” in *Proceedings of the 1987 Annual ACM SIGAda*, Boston, Massachussets, 1987, pp. 216–225.

[27] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Manet, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the Scala programming language," Tech. Rep., Ecole Polytechnique Federale de Lausanne, Lausanne Switzerland, 2004.

[28] M. Odersky, "Scala by example," Tech. Rep., Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, January 2006.

[29] M. Odersky, P. Altherr, and V. Cremet, "Scala language specification," Tech. Rep., Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, January 2006.

[30] K. Thorup, "Genericity in Java with virtual types," *Lecture Notes in Computer Science*, vol. 1241, pp. 444–461, 1997.

VITA

Olayinka N'guessan received her Bachelor of Science in Computer Science, Physics and Mathematics from Minnesota State University, Mankato in May 2004. She entered the Computer Science program at Texas A&M University in August 2004, and since then, she has been pursuing her Master of Science degree receiving it in December 2006. Her research interests include programming languages and computer-human interactions.

Her e-mail address is olanys@yahoo.com.