“Programming in Algorithms”: Generic Programming and its Implementation

By Daniel Giovannelli

April 25, 2013

Abstract

The decisions which language designers make when implementing new features, and the effects which those decisions have on a finished language, is a fascinating field of study which has been largely overlooked by many programming language researchers. In this paper, we will explore the implementation of generic programming, a powerful programming technique, in four modern programming languages (C++, C#, Java, and Haskell). We discuss the process of designing each implementation and how those design choices affect their use in each language. In doing so, we build a coherent theory of trade-offs in language design and how those trade-offs can be minimized.

Contents

Section 1: Background Material 2
  Chapter 1: Introduction .................................................. 2
  Chapter 3: The History of Generic Programming ........................ 5
  Chapter 4: Further Background and Code Examples .................... 6

Section 2: Implementations 9
  Chapter 5: Generic Programming in C++ .................................. 9
    Introduction and History .............................................. 9
    Design and Implementation ........................................ 10
    Advantages and Disadvantages .................................... 12
    Testing Template Code Expansion .................................. 12
  Chapter 6: Generic Programming in Java .............................. 14
    Introduction and History .............................................. 14
    Design and Implementation ........................................ 15
    Advantages and Disadvantages .................................... 17
  Chapter 7: Generic Programming in C# ................................... 18
    Introduction and History .............................................. 18
    Design and Implementation ........................................ 19
    Advantages and Disadvantages .................................... 20

Part 3: Final Thoughts and Conclusion 21
  Chapter 8: Comparing Implementations of Generic Programming ....... 21
  Chapter 9: Generic Programming in Haskell ............................ 24
  Chapter 10: Conclusion .................................................. 25

Appendix A: C++ Template Experiment Data 27

Appendix B: C++ Template Experiment Code 32
Section 1: Background Material

Chapter 1: Introduction

The design and implementation of programming languages is a fascinating field of study which has grown by leaps and bounds in recent years. It often seems that exciting new programming languages are being designed and released every day. That being said, studies of programming language design often focus specifically on technical details. Many researchers focus primarily on what unique things a given programming language can do and how it goes about doing them; or, alternatively, how a language implements old features in a new way. Just as fascinating, however, is the question of what kind of decisions are made when a language is designed, and why those decisions are made the way they are. All sorts of factors - from philosophical stances to usability concerns to questions of backwards-compatibility - can play a role in how a language is designed, and even the smallest changes can wind up having wide-reaching effects on a finished language. This paper will explore how design decisions are made in a programming language. What kinds of concerns influence these decisions? How do certain constraints shape implementation details? And how does this ultimately affect a finished programming language?

While the question of how languages are designed is an intriguing one, with a great deal available to study, it is much too large and far-reaching to attempt to examine directly. Rather, we will study the process of language design through the lens of one specific programming language feature: genericity mechanisms. Generally speaking, ‘genericity mechanisms’ refer to the ways in which a programming language can support an abstract construct known as ‘generic programming’. The specifics of generic programming will be covered in detail in a later chapter, but in short it refers to the process of writing algorithms which are abstracted away from the representation of the data on which they operate, allowing these generic algorithms to work on a wide variety of data regardless of the specifics (such as type) of the data itself.

Genericity mechanisms provide a valuable test case for studying language design for one main reason: generic programming is a purely abstract programming model that is totally independent of any given programming language, rather than a specific implementation detail. This means that language designers have a great deal of freedom in determining a language’s genericity mechanism - as long as that mechanism allows for generic programming, the implementation details of the genericity mechanism are irrelevant. Because of this, very similar languages can (and often do) have very different genericity mechanisms. By studying the differences in genericity mechanisms across very similar languages, we can focus on how design decisions can lead to said differences without worrying too much about how they are shaped by the specifics of the underlying languages.

We will begin this paper by discussing the details of generic programming in the abstract, and describing the history and development of generic programming from its mathematical roots in the 1970s to its use today. We will then move on to a close examination of genericity mechanisms across three major modern programming languages (C++, Java, and C#). We will explore how specific decisions made during the design of the languages balanced considerations of flexibility, code re-use, type-safety, and performance, and how these decisions affected the design of the finished language. Finally, we will look at the genericity mechanism present in a fourth language, Haskell, which is a radically different kind of language than any of the other three. This will provide us with an opportunity to explore how major differences in the core language itself can shape the implementation details of a given feature.

Chapter 2: What Is Generic Programming?

Before we can discuss how genericity mechanisms are implemented in programming languages, we must first discuss the concept of generic programming in a bit more detail. Broadly, ‘generic programming’ is a programming style in which algorithms written at the broadest, most abstract possible level, independent of the form of the data on which these algorithms will be carried out. When used correctly, generic programming techniques can cut down on repetitive code and aid in code re-use (by allowing a programmer to avoid having to write multiple implementations of extremely similar algorithms) and help ensure code correctness (by allowing programmers to focus on debugging and fine-tuning one instance of a given algorithm, rather
than having to worry about many similar implementations). Early pioneers of the technique claimed that generic programming techniques would also lead to higher-quality code overall, by allowing programmers to focus broadly on the algorithms which they use rather than getting bogged down in implementation details [12].

![Figure 1: Creating Lists Without Generic Programming Techniques](image1)

![Figure 2: Creating Lists With Generic Programming Techniques](image2)

Originally, generic programming research did not specify how languages should support the practice - it was thought that individual languages would implement the specific genericity mechanisms which best suited the language itself. In reality, however, up until fairly recently genericity mechanisms in most languages focused primarily on types as the notion of the ‘data form’ which algorithms are abstracted over, and on models of parametrized types as how this abstraction occurs. In these models, an algorithm can be called with any parameters, regardless of their type, and is passed the types of its parameters as arguments along with the parameters themselves. These types can then be examined, reasoned about, and used by the algorithm itself [12]. In more recent years, the shift towards pure object-orientation in programming languages has brought with it a shift how programmers conceive of ‘data form’ from types to classes, and in most modern programming languages generic classes deal with parametrized classes, rather than parametrized types.

It should be noted here that while the parametrized types/classes model provides a fairly robust genericity mechanism, not everything that can be done with parametrized types is generic programming. More specifically, a distinction should be drawn between what Christopher Strachey refers to as ‘parametric polymorphism’ and ‘ad-hoc polymorphism’. Parametric polymorphism refers to the ability of the exact same algorithm to work on data, regardless of that data’s type. An example of this would be a `max()` function that returns the larger of two values no matter those values’ types (provided that they are both of a type which can be ordered). Ad-hoc polymorphism, on the other hand, refers to algorithms which can take as input data of any type, but behaves differently depending on the type of the data [17]. While type parametrization can be used to implement either type of polymorphism, generic programming generally refers only to the use of parametric polymorphism (and in fact other, more robust mechanisms such as operator overloading are generally preferred to implement ad-hoc polymorphism).
In 1985, Luca Cardelli and Peter Wegner wrote a seminal paper on polymorphism and type theory, entitled “On Understanding Types, Data Abstraction, and Polymorphism”. Though the majority of the paper is devoted to exploring type theory by way of a typed lambda calculus that the authors refer to as Fun and present as a model for strongly-typed, object-oriented programming languages, the authors also thoroughly explore various types of polymorphism. They argue that there are in fact four types of polymorphism - parametric and inclusion (which, together, are referred to as ‘Universal Polymorphism’), and overloading and coercion (which are referred to as ‘Ad-Hoc Polymorphism’). They also expand their definitions from those of Strachey (who they cite as an inspiration) by dividing polymorphism into universal polymorphism and ad-hoc polymorphism, and claim that “Universally polymorphic functions will normally work on an infinite number of types (all the types having a given common structure), while an ad-hoc polymorphic function will only work on a finite set of different and potentially unrelated types... In terms of implementation, a universally polymorphic function will execute the same code for arguments of any admissible type, while an ad-hoc polymorphic function may execute different code for each type of argument” [4]. More specifically, they claim that ad-hoc polymorphism can be broken down into ‘overloading polymorphism’, which is implemented via operator overloading for distinct types, and ‘coercion polymorphism’, which refers to the process of changing a value in one type to the same value as another type (e.g., turning int 1 into float 1.0) so that it can be passed to a particular function. They also claim that universal polymorphism can be further subdivided into parametric polymorphism, as defined by Strachey, and inclusion polymorphism, in which an object is considered to belong to multiple, potentially overlapping types - which is often implemented in object-oriented programming languages as inheritance. It is worth noting that while this paper will be focusing primarily on parametric polymorphism, we will touch briefly on inclusion and coercion polymorphism when we discuss how generics are implemented in pure object-oriented programming languages (particularly Java).

Generic programming is generally used in one of two ways: via generic functions or generic classes. Generic functions are functions which can take parameters of any type and act on those parameters (and their type). A commonly cited example of a generic function is max(), a function from the C++ Standard Template Library which returns the larger of two arguments. Max() could be defined (using C++ syntax - see chapter 3 for a discussion of Max() and source code for the function in other languages) as follows:

```cpp
template <typename T> T max (T a, T b) {
    if (a > b) { return a } else { return b }
}
```

This piece of code demonstrates several key points about generic programming. Firstly, notice that while the types themselves are left abstract, the relationship between the types is not. This function must take as its arguments two values of the same type, and return a value of that type. If another function attempted to call this function with two different types, or assign the return value to a variable of a type other than the types passed to it, it would return an error. This is not to say that a generic function can only be passed one type - the above function could be modified to take as input two items of different types if the first few lines were changed to:

```cpp
template <typename T, typename P> T max (T a, P b) {}  
```

Another key factor highlighted by this code example is that max() will be correct independent of type information so long as the type in question supports the > operator, providing parametric polymorphism.
This may seem self-explanatory, but it is vital to understanding generic programming.

The other major implementation of generic programming (especially in more modern, purely object-oriented languages, which do not have support for functions that are not bound to classes) is generic classes. A generic class is similar to a generic function, but which instead allows for representation of data in a particular way regardless of the data being represented. A classic example (again, from the C++ STL) is a generic list class which is able to represent a list of values of any arbitrary type. The constraints that apply to generic functions (such as type relationships and no reliance on type information) hold true for generic classes as well.

Chapter 3: The History of Generic Programming

The roots of modern generic programming date back to the research conducted by two computer scientists, David Musser and Alexander Stepanov, in the early 1970s. Originally, Stepanov and Musser conceived of generic programming as a theoretical programming process in which specific algorithms, defined for a particular form of data, are used to determine the most abstract possible form of the algorithm. This abstract algorithm can then be applied to any form of data. Note that ‘form’ in this case does not necessary mean ‘type’; while generic programming ultimately came to be expressed almost entirely through type systems, the association between generic programming and type systems came later, and Stepanov himself refers to the “algebraic structure” of data rather than any implementation details of said structure [11]. From the beginning, generic programming was designed specifically to maximize programmer productivity by cutting down on code re-use wherever possible. Musser and Stepanov also cited greater code reliability in their designs for generic programming, arguing that one algorithm working across a variety of data representations would be easier to debug and fine-tune than a variety of algorithms each working on a specific data representation [12].

While Musser and Stepanov are the best known researchers in the field of generic programming (and in fact were responsible for coining the term ‘generic programming’ itself in 1989), other researchers at the time performed similar work on abstracting algorithms from implementation. The best known of these is Joseph Gougen, whose work on a model he called ‘parameterized programming’ evolved alongside Musser and Stepanov’s research on generic programming. While the goals of parameterized programming were essentially the same as those of generic programming - maximizing code correctness and re-use - the fundamental focus of the two theories were different. Gougen argues that algorithms should not only be independent of the data form upon which they act, but should in fact consist of a general, high-level interface which parameterizes all of the environmental properties which the interface will make use of. This includes not only data form but also all of the resources in a system that the algorithm requires. This model is significantly more far-reaching to implement, and harder to understand than Musser and Stepanov’s model, and as a result fell largely out of favor with the advent of generic programming. Today, Gougen’s model is not commonly known, and is used only in a small handful of esoteric, purely academic programming languages (most of which are descended from the OBJ language that Gougen designed to demonstrate his ideas) [9]. While Musser and Stepanov discuss this model in their 1989 paper outlining generic programming as a model, they claim that it is fundamentally different from their own, writing “Some of our goals are in common with the parameterized programming” approach advocated by J. Goguen, but the most fundamental difference is that Goguen mainly addresses metaissues - namely, how to manipulate theories - while our primary interest is in building useful theories to manipulate.” [12].

Generic programming techniques first moved from theoretical algorithms to actual implementation in 1987, when Musser and Stepanov released a library of generic data structures (specifically singly linked lists, which were used as the basis for other, similar data structures such as stacks and deques), and associated algorithms (such as generic merge and sort algorithms) for the Ada language. This implementation also marked the beginning of the association between what Stepanov refers to as the “algebraic structure” of data - that which is dissociated from generic algorithms - and data type. In the Musser and Stepanov paper which originally described this library, the authors point out that “In Ada, data abstractions can be written as packages which define a new type and procedures and functions on that type. Another degree of abstractness is achieved by using a generic package in which the type of elements being stored is a generic
formal parameter.” [13]. This form of data abstraction remained central to generic programming until the rise of modern, object-oriented generics, which instead considers classes to be the primary abstraction of data to which generic algorithms are applied.

Though Musser and Stepanov’s generics library provided Ada with a fairly robust (if basic) genericity mechanism, generic programming remained fairly little-known outside of academic and language-design circles (in large part due to the relatively weak market share of Ada and other languages, such as Eiffel, which included genericity mechanisms). Generic programming did not become well-known among programmers in industry until 1994, with the development of the C++ Standard Template Library, or STL. The STL, which was designed primarily by Meng Lee and Alexander Stepanov himself, provided a library of generic container classes (and associated methods) such as vectors, sets, and stacks built on C++’s template system (the details of which will be discussed later in this paper). Stepanov himself believed that the direct memory access allowed by pointers in C++ would allow for a more efficient implementation of his theories of generic programming, and (after working closely with Bjarne Stroustrup on the initial design of templates) built the STL as an attempt to demonstrate this [20]. Unlike Musser and Stepanov’s earlier Ada generic library, the STL was widely adopted among C++ programmers in industry (due in large part to many vendors advertising STL support in their compilers), and thanks to C++’s dominant market share, the STL led to generic programming practices becoming fairly popular among non-academic programmers.

While the STL managed to gain significant traction and popularized generic programming, it was fundamentally similar in design and execution to Stepanov’s earlier Ada library. This occurred largely because in the case of both Ada and C++, the different classes of algebraic structures on which generic algorithms act were thought of primarily as types. Though C++ templates are able to accept user-defined types as input (and thus work with objects of user-defined classes), the C++ approach to generic programming still focused primarily on treating algorithms as working independently of input type. In the early 2000s, however, the advent of object-oriented programming as a mainstream programming technique lead to a general trend towards thinking about basic algebraic structures as classes. Though Stepanov himself has argued against this trend, claiming that this is a fundamentally backwards way to think of data [11], thinking about generic algorithms in terms of classes rather than types has become a key part of modern genericity mechanisms (especially those used in pure object-oriented languages like Java and C#), and this does not seem likely to change in the near future.

Chapter 4: Further Background and Code Examples

At this point, readers with a background in dynamically-typed programming languages such as Python, Ruby, or Lisp may be wondering why, exactly, genericity mechanisms are important at all? At first glance, it seems that parametric polymorphism only provides a way to allow functions in statically-typed languages to operate on data of any type - something that dynamically typed languages are able to do by default. This is essentially the fundamental difference between statically and dynamically typed languages: in statically typed languages, all values are labeled with their type, and all procedures have annotations marking what type or types they operate on and return. This allows for robust error checking at compile-time, by effectively comparing the labels on values to the annotations of the procedures using those values. Dynamically typed languages lack these labels and annotations, allowing for greater flexibility but limiting the error-checking which can be done before run time. The important point is not, however, what static languages with genericity methods are able to do - it is what dynamically typed languages are not able do. Specifically, while dynamic languages allow by default for the flexibility and code re-use provided by good systems of parametric polymorphism, they lack the type-safety and (oftentimes) performance that comes from using statically-typed languages. Genericity mechanisms in statically-typed languages often serve to provide flexibility and code re-use while still maintaining type-safety and performance. In fact, the tension between these four factors is a key element of language design, and one which is a factor in the implementation of any language feature, not just generic programming. By discussing generic programming and its implementation across several languages, we will be able to explore how language designers resolve this tension in different ways (and with different priorities) - and what effects this process can have on the experience of using a finished language.
Hundreds (if not thousands) of programming languages have been designed and released over the last fifty years, and choosing a small handful of languages to examine, in any situation, can be a tremendous task. That being said, for this paper, we have chosen to focus our analysis on C++, Java, and C# (with a brief discussion of Haskell at the end). With the exception of Haskell, these languages are all statically-typed, object-oriented, mid-to-high-level compiled programming languages which are derived from C (some more than others). Though they differ in some key ways, these differences are generally not relevant to how they handle generic programming (or, in the cases which they are relevant, are relevant in interesting ways - such as the presence of a virtual machine in Java and C#). Additionally, while these languages are all very similar, they all handle generic programming in very different ways - and in some cases, these differences are informed by the implementation of generics in one of the other two. This means that these three languages provide an excellent basis for examining implementations of generic programming, as they will allow us to look past specific language differences and focus instead on the broader themes of how and why particular language features are designed across languages.

Over the course of this paper, we will refer to several functions and classes to provide examples to illustrate various concepts. We will now define those functions, and provide example source code for them in three of the languages which we will discuss (Haskell is difficult to compare directly to C++, Java, and C#, so our discussion of Haskell will not use these code examples).

Max(a, b)

Max(a, b) is an extremely simple function that takes two items of any type, and returns the larger of the two. We will use it as an example of a basic function which uses parametric polymorphism. It relies on the > operator; if the type passed in to Max does not support >, the function will return an error (either at compile-time or run-time, depending on the language used).

Note: In Java and C# this method must be part of a class, though the class need not be generic itself

<table>
<thead>
<tr>
<th>C++:</th>
<th>Java:</th>
</tr>
</thead>
</table>
| template <class T>
  T max (T a, T b) {
    return (a<b) ? b : a;
  }
| public static <T extends Comparable<T>>
  T max(T a, T b) {
    return a.compareTo(b) > 0 ? a : b;
  }

<table>
<thead>
<tr>
<th>C#:</th>
</tr>
</thead>
</table>
| public static T Max<T>(T a, T b) {
  return (Comparer<T>.Default.Compare(a, b) > 0) ? x : y;
}

Figure 4: Max(a, b) Implementations in C++, Java, and C#

List

List is a simple class that allows the user to create a list of items of any one type. The items in the list must still be of the same type; however which type that is may be defined by the user. List is provided
as an example of a basic generic class which is created with parametric polymorphism. Note that we will not define any of the methods of List, as generic methods are covered by other examples; we’re primarily interested in how a generic class is defined and structured.

Add(a, b)

Add(a, b) is a basic function that can take two elements of any type. If the elements are of type int, their sum is returned, and if the elements are of type String, a single String made up of a concatenated with b is returned. If the elements are of another type, an error is printed. This function is provided as an example of a basic (but still useful) function which relies on ad-hoc polymorphism. It also demonstrates some of the issues that come with particular implementations of generic programming; specifically those caused by type erasure in Java.

A note on C++: ad-hoc polymorphic functions such as add() can be written in C++ using one of two techniques: reflection or template specialization. We will demonstrate both techniques below. Note that reflection makes use of a feature called RTTI (or Real-Time Type Information) to examine the object’s type; RTTI is not standard, and different compilers implement it in different ways (if at all). Even if the compiler being used support RTTI, using RTTI to implement ad-hoc polymorphism is generally considered poor programming practice (programmers are generally encouraged to use overloading or template specialization instead).

A note on Java: this method will not work properly in Java due to a process known as type erasure, by which type information about the arguments to generic classes and methods is eliminated at compile time; we will discuss the reasoning behind the use of type erasure and its advantages and disadvantages in more detail in the chapter on Java generics. Implementing this method in Java correctly would require adding an Add(a, b) method separately to Integer and String which each performed differently (or overloading the existing + method for each).
Section 2: Implementations

Chapter 5: Generic Programming in C++

Introduction and History

C++ implements generic programming through a language feature known as templates. Bjarne Stroustrup, the creator of C++, considered them a fundamental feature necessary for effective object-oriented programming [18], and designed a robust templating system for the language [19]. C with Classes, the predecessor language to C++, allowed programmers to use C preprocessor macros to parametrize container classes, but this was considered a sloppy, interim solution at best (not least due to the fact that C macros cause serious issues for variable scope and type checking). An effective genericity mechanism was considered so important for C++, in fact, that the language standard mandated the inclusion of templates. Despite this outpouring of support for the feature, however, templates were not originally included in early C++ implementations due to concerns over the complexity of any template integration, as well as slowed compilation and linking times [20]. In fact, the earliest appearance of templates in C++ did not occur until 1989 when class templates (but not function templates) were added to the Cfront compiler, almost ten years after the creation of the language; templates were not fully supported by any C++ compiler until 1992 [20].

According to Bjarne Stroustrup, the design of genericity mechanisms in C++ hinged on three key points - notational convenience, run-time efficiency, and type-safety [20]. The first of these, notational convenience, was considered essential to making a fairly complex theoretical idea (generic programming) as clear as possible for programmers. The notation used for C++ templates needed to highlight that the type parameters
used in template construction were distinct from normal parameters. At the same time, in order to speed adoption and ensure language cohesion, templates needed to be as syntactically close to the rest of the language as possible; writing a templated function still needed to ‘feel’ like writing a function.

Run-time efficiency in templates was considered of paramount importance due to the need for programs using templates to actually be viable in production. Programmers would shy away from template use (and generic programming in general) if using generic techniques would cause noticeable ill effects for compiled software, so run-time optimization was considered a priority for template implementation. It is important to point out here that run-time efficiency was considered far more important than compile-time efficiency - while keeping compilation times down as much as possible was considered important for any new feature, run-time efficiency was seen as a necessary feature, while reasonable compile-time efficiency was viewed as a constraint on possible solutions [20]. The focus on run-time efficiency stemmed largely from negative experiences that Stroustrup and his colleagues had with the approach Smalltalk (an early object-oriented language that served as a precursor to C++) took towards container classes - namely, implementing them using dynamic types and inheritance. Though this implementation worked, it also led to major issues in run-time efficiency for code that used parametrized container classes [20].

Issues with Smalltalk’s implementation of container classes also gave rise to one of the major concerns for C++’s implementation of templates: type-safety and overall robustness. The dynamic typing used in Smalltalk made any sort of static type-checking impossible, and while other languages existed at the time which had both static type systems and generic programming support (Stroustrup specifically mentions Clu; Ada also fits this description) the implementation of generic programming in these languages was generally very complex (they were also considered to be inflexible, though Stroustrup attributes that perception more to their lack of inheritance mechanisms for generic classes than the implementation of the generics itself [20]). Thus, a major goal of C++ was to maintain complete type-safety for generic classes and functions while still allowing the flexibility present in Smalltalk-style genericity mechanisms.

Design and Implementation

Template syntax in C++ is fairly straightforward. Templates (both function and class) begin with the keyword template followed by either `<class T>` or `<typename T>` (either name can be used and both work exactly the same; originally only ‘class’ was used, and ‘typename’ was later added as an option to resolve confusion among programmers as to whether or not primitive types could be passed as template arguments). T, the name of the type can be any legal C++ identifier, however, T is used here because it is frequently used to represent a type parameter in documentation. Note that a template may accept any number of distinct type parameters using the syntax template `<typename T, typename U>`. This statement is then followed by a function or class definition in which the type variables provided can be used anywhere a type would be used (for an example, see Chapter 3).

This syntax neatly fulfills Stroustrup’s dual design criteria of highlighting the fact that type parameters are distinct from regular parameters while also making writing templated functions and classes feel as natural as possible. Template type declarations have a unique, easily-recognizable syntax: the only other situations in which angle-brackets are used in C++ are to represent greater-than and less-than operations and in import statements to represent non-local imports. Template instantiations are easily distinguishable from these other scenarios due to the presence of the template keyword, making it easier for programmers to recognize and differentiate type parametrization from normal parameters. That being said, once the template has been declared, class or function definitions can be written normally using the templated type just like any other type. This ensures that writing templates feels as natural as possible for programmers.

C++ templates are implemented in the compiler by way of a process that is sometimes referred to as “template unfolding”. In short, during compilation the compiler generates a copy of each templated class or function for each type that the class or function is called with. For example, if the $max()$ function described earlier were called with two int arguments at one point and two String arguments at another, template
unfolding would generate two max() functions; one with int arguments which returned an int and one with String arguments that returned a String. This leads to an issue which is sometimes referred to as the ‘template instantiation problem’: a compiler must guarantee that a templated function or class with a given set of types is declared once and only once in an executable if it is used, and is not declared at all if it is not used. The C++ standard does not declare how this should be accomplished, so compiler writers are left to determine their own solutions [7]. While different compilers over the history of C++ have dealt with this using several different techniques (some more efficient than others), a common element of such solutions is having template unfolding not occur until after the linker has linked together source files (which is fairly late in the compilation process) to prevent duplication, making it one of the last steps undergone during compilation prior to code generation.

While templates address parametric polymorphism in C++ and allow for full use of generic programming techniques, they do not seem to allow any mechanism for ad-hoc polymorphism. Such functionality is provided through one of two features - type reflection via a (non-standard) compiler feature called Real-Time Type Information (or RTTI), or by way of a feature known as template specialization. RTTI allows the compiler to determine type information about a variable at run-time, while template specialization allows for a programmer, after declaring and defining a template, to re-define that template differently if it is called with a particular type. Examples of both of these uses can be found in chapter 4, figure 6.

In 2011, the ISO released a new standard for the C++ language. This standard (colloquially known as “C++11”) consists of significant additions and changes to the C++ language and standard libraries, many of which address major concerns held by many C++ programmers. While the C++11 standard makes several additions and improvements to C++ templates, these additions are primarily matters of convenience, doing little to change how templates are actually compiled. There are two main additions to the template system - extern templates and variadic templates. Extern templates address a concern put forth by many expert C++ programmers: most C++ compilers compile templates within a single translation unit (a “translation unit” refers to a block of .cpp/.h files connected by the linker; a single program consists of one or more translation units). If identical templates are used in different translation units, each template will be compiled multiple times unnecessarily, which can create significant overheads in compile time and increased
binary sizes in sufficiently large projects. Templates can now be marked as extern templates, which essentially tells the compiler not to compile a template because it will be compiled in another translation unit. Variadic templates allow a template to be written to accept a variable number of type arguments. While these are major additions that are extremely useful to programmers using templates, they have little effect on how the compiler handles templates - extern templates are simply ignored, while each instance of a variadic template with different inputs is compiled to a different bytecode template, just as normal templates are [1].

Advantages and Disadvantages

Implementing generic programming via template unfolding provides one significant advantage over essentially any other genericity mechanisms: template unfolding provides extraordinary type-safety and robustness. By actually generating copies of a template class or function for every required call, the compiler is able to ensure that values of the type being passed to a given template can actually perform any methods required by that template, allowing for an entire class of errors to be caught at compile-time (instead of runtime, as they would be in a Smalltalk-style dynamically-typed genericity mechanism). Additionally, template unfolding allows for a guarantee that if the same container class is used for different types, the resulting classes will be treated as distinct by the compiler - a guarantee that is not present in some other languages (most notably Java).

Unfortunately, this robustness comes with a significant cost. Since the compiler must generate new classes or functions for each use of a template, heavy use of templates can lead to both noticeable, significant slowdowns in the time required for compilation, and significant increases in the size of compiled binaries. It should be pointed out that in the original design of C++ templates this was considered an ‘acceptable’ flaw - so long as the execution of programs was not significantly slowed, slower compilation times were considered a legitimate trade-off [20]. Additionally, since template unfolding takes place late in the compilation process, after separate compilation units have been linked together, compiler errors caused by issues in templates are often difficult to understand and track down. This is compounded by the fact that it is difficult to effectively use debuggers to trace issues in template definition, since the base templates are no longer present at runtime. Template unfolding can also lead to significant increases in the size of compiled binaries, since the code itself is being duplicated for each type of use. This increase in compilation time and program size grows quickly the more templates are used, and can even lead to massive increases in the rare case that a template argument is itself a templated class.

Testing Template Code Expansion

That being said, no research currently exists testing exactly how large this increase, or if it is seen at any realistic level of template nesting. In order to rectify this, we have conducted a series of small experiments to determine if we could get the size of a C++ binary to grow at a faster rate than that of the source code. In each experiment, we created a series of class templates, the constructors of which created other templates in a sort of cascade. At each iteration, another layer of templates was added to the program, itself creating more templates down the chain. In each experiment, iteration 1 consisted of a simple wrapper template (named ‘tempA’, for Template A) which took a single value of a given type, and iteration 2 added a second template (tempB) which took two arguments of two different types and created an instance of tempA for each.

In Experiment 1, each iteration after 2 added a template which took two arguments and created two instances of the template added in the iteration before it (one for each permutation of the two arguments). Thus, in iteration 3, an instance of tempC<

\texttt{int, float}> was created, which then generated instances of tempB<

\texttt{int, float}> and tempB<

\texttt{float, int}>, which then generated tempA<

\texttt{int}> and tempB<

\texttt{float}>. In iteration 4, an instance of tempD<

\texttt{int, float}> was created, which generated instances of tempC<

\texttt{int, float}> and tempC<

\texttt{float, int}>, and so on. This was repeated for seven iterations. Seven was chosen because there are seven primitive (non architecture-dependent) types in native C++ (\texttt{int}, \texttt{short}, \texttt{float}, \texttt{long}, \texttt{char}, \texttt{double}, and \texttt{bool}). While this was not relevant in this particular experiment, it became relevant in the next two (as it allowed us to use basic types without resorting to libraries or custom types), so we used seven iterations.
for Experiment 1 as well in order to keep our experiments consistent.

Experiment 2 was structured similarly to Experiment 1, with one major difference: while each template still only took two arguments, each template generated six instances of the prior template - in each case, creating a template with each of the permutations of the two argument types and a third, hardcoded type which differed for each template (this led to six instances instead of nine because permutations with repeating elements were not used). Like Experiment 1, seven iterations were run, since each added a new primitive type to the template cascade.

Experiment 3 was similar to Experiments 1 and 2, but with one key difference: each template used took one more argument than the template before it, and generated a number of new templates equal to the permutations of the arguments to that level of template (meaning that one iteration created six new templates, the next 24, and the next 126). This meant that Experiment 3 had a quadratic rate of source code growth, as opposed to 1 and 2's linear growth. We originally intended for Experiment 3 to continue for seven iterations, but due to difficulties creating the test code we were only able to run it for five.

Our hypotheses for these experiments were fairly straightforward. In Experiment 1, it was assumed that the size of generated binary code would increase quadratically, since the number of templates created doubles with each iteration. Our hypothesis for Experiment 2 was similar: while the source code size still increased at a linear rate, we suspected that the creation of more levels of templates at each step would create a much larger (but still quadratically-growing) expansion. For Experiment 3 we assumed that the rate of growth in binary size would match that of the source code - IE, since the source size grew at a quadratic rate, we expected the binary size to grow at a quadratic rate as well. This experiment was included to test if unexpected behaviors begin to occur at such high rates of source code growth (in other words, if we would actually see exponential, rather than quadratic, growth).

Our findings for Experiments 2 and 3 essentially mirror our hypotheses, as both displayed quadratic growth (see Appendix A for specific results). Experiment 1, however, provided more unexpected results: both the binary and source code size growth appeared to be linear. The difference between the findings seen in Experiment 1 and 2 warrant further exploration. The fact that binary size grows quadratically in Experiment 2 makes intuitive sense: each template creates six other templates, which themselves create six other templates, so the addition of a single additional iteration multiplies the number of created templates by six. It is less clear, however, why the same pattern would not be seen in Experiment 1, as that followed the same pattern of growth, with the number of new templates created at each step reduced by only a constant factor. While an obvious answer might be that seven iterations was simply not enough to see a pattern indicating quadratic growth, this seems unlikely: if the pattern of growth was quadratic we would expect to see that the changes in binary size at later iterations would be at least somewhat smaller than those in earlier iterations, but the growth in binary size at each step is consistently 200-300 bytes. Another possible answer may be that the compiler performs some sort of optimization in the case of Experiment 1 that is not performed in Experiment 2; further research is necessary to test this hypothesis.
While the fact that a linear code size increase can lead to a quadratic expansion in binary size is worrying, several factors keep it from being a major issue in actual use of the language. Firstly, it is very rare that the kinds of cases used in this experiment would ever appear in production code - there are few if any circumstances in which a user would ever want to have more than (at most) two or three layers worth of templates creating other templates. Secondly, even if such code did appear in production, the fact that a template is only expanded once for each type means that it is unlikely that so many templates would need to be expanded. The code used in this experiment was specifically designed to instantiate as many typed template classes as possible; actual production code would be far more likely to contain calls to overlapping templates, which would only need to be expanded once.

Chapter 6: Generic Programming in Java

Introduction and History

Moving on from C++, we come to Java. Java’s implementation of generic programming (via a feature known as ‘generics’) provides an intriguing case study for a variety of reasons. Firstly, Java was arguably the first extremely popular entirely object-oriented programming language, and is the language which first introduced many non-academic programmers to object oriented programming. As such, Java provides an excellent opportunity to examine how generic programming can be expressed within a purely object-oriented framework. Secondly, and perhaps more interestingly Java (like C++ before it) did not initially offer any kind of support for generic programming. Unlike C++, however, no genericity mechanism was written into the Java language standard - generics were not added to Java until 2004, nine years after the original release of the language, and even then were only added after both tremendous programmer demand and the creation by a researcher of a separate form of Java which included support for generic programming. Because of this late addition, the design of Java’s generics was affected by significant constraints, most notably the requirement to maintain backwards compatibility with code written for earlier versions of the Java Virtual Machine - leading to some unique elements present in their design.

Originally, Java did not contain any features specifically designed to support generic programming. Programmers were expected to create container classes by creating classes which accepted as a parameter one
or more objects of type Object (which, since all non-primitive Java classes inherit from Object, means that data of any type may be passed in), and use casts to convert these elements to the proper type when necessary. While this solution worked, it was highly type-unsafe, and (by relying on programmers to correctly cast objects retrieved from container classes) tended to lead to highly buggy, unreliable code [15]. In 1997, Odersky and Wadler published the first paper on a language which they called Pizza. Pizza was a strict superset of Java that compiled directly to Java code, which could then be compiled to Java bytecode and run on the JVM (much the same way C With Classes originally compiled directly to C), and was designed to introduce several advances made by so-called “academic” programming languages to Java (including higher-order functions, algebraic data types, and, of course, generic programming) [15].

Pizza’s genericity mechanism, which Odersky and Wadler referred to simply as parametric polymorphism, was very similar to C++ templates both syntactically and in terms of use. Interestingly, Pizza provided two separate implementations of parametric programming: the so-called heterogeneous implementation, in which separate versions of generic classes were created at compile-time for each type, C++ style, and the homogeneous implementation, in which generic classes were compiled into single classes which took as their arguments objects of type “Object” and used casts to change them to the correct type when necessary (much as Java programmers had been doing by hand). Pizza’s compiler allowed users to choose which implementation they wanted to use - Odersky and Wadler argued that generally the homogeneous translation would produce smaller binary files and more compact code, while the heterogeneous translation would produce faster code, and that programmers should be able to choose which style of translation was most appropriate for a given project.

While Pizza received a fair amount of notice in academic and language-design circles, it was not widely adapted by programmers in industry. Nevertheless, Pizza’s treatment of generic programming was well-enough received that Odersky, now working with Gilad Bracha, expanded the language into a project he called Generic Java, or GJ. GJ stripped away the non-generic programming related features added to Pizza, and, rather than being presented as a separate and distinct language, was designed to be an extension of basic Java, with full backwards compatibility for code written in earlier forms of Java (it should be noted that while what Gilad and Bracha refer to as ‘backwards compatibility’ might be more properly referred to simply as ‘compatibility’, ‘backwards compatibility’ is the phrase Gilad and Bracha use to describe how GJ solves what they refer to as “the legacy problem”, and so the term ‘backwards compatibility’ will be used here as well). In order to aid backwards compatibility and simplify the cases in which it would be necessary to retrofit legacy code for container classes, Generic Java dropped Pizza’s optional heterogeneous implementation of generic programming, in which new classes were generated for each type used, instead focusing entirely on homogeneous translation [3]. Unlike Pizza, Generic Java was widely circulated and discussed (especially among C++ programmers learning Java, who were disappointed with the lack of anything resembling templates in Java). This support led Bracha and Odersky to formally present the Java Standards Committee with a proposal (modeled closely on GJ) to add support for generics to Java [2]. This proposal was adopted in 2004, and generics were finally included in version 1.5 of the JDK.

Design and Implementation

When designing Java generics, Bracha and Odersky had one primary concern that trumped all others: since generics were added to the language so late, and since so much production code had been written in Java in the years since its original release, it was considered absolutely vital that generics be entirely backwards-compatible with older versions of Java. In this case, backwards compatibility was taken to mean both that code written without generics needed to be able to run alongside code written with generics without any issues, and that it was necessary to be able to use older library classes which faked genericity via casts from Object without re-writing older libraries [3]. This point is a subtle one, and it is best illustrated by the following example, presented by Bracha and Odersky: say a library written before the addition of generics to Java contains a generic class Collection, written using the standard Object-type-and-cast method for faking generic programming in Java (which Bracha and Odersky refer to as “the generic idiom”). It was considered vital for the interoperability of new and legacy code that this class be compatible with a generic class Collection<T>, since if it is not users would be required to either convert all references to Collection...
in their code to Collection<T>, keep track of two different forms of generics in their code, or write code to automatically convert one to the other - all of which were considered unacceptable options for maintainers of large codebases. This backwards-compatibility proved to be the most significant constraint in the design of Java generics, and played a major role in determining how generics would ultimately work.

Another constraint in the design of Java generics came from Java’s philosophy. Java is a purely object-oriented language; all code in Java is contained within classes and nearly everything in the language is an object with the abstract class Object at the top of its inheritance tree. Even functions must be treated as objects: since Java does not have support for first-class functions, passing functions around requires wrapping them in a class with the function in question present as a method of the class. The only exception is a small handful of primitive types, such as int, float, byte, and char which are placed on the stack (instead of in the heap like most objects) and can be used as values without classes - and even these can be automatically converted to an object of an associated class (such as Integer for int) when necessary through a process called boxing. Any generic addition to Java thus needed to work within the context of pure object-orientation. While it may seem that this is a fairly straightforward constraint (which can be met simply by making it impossible to have a generic function that is not bound to a class, the way function templates allow in C++), it does lead to some subtle issues in how generics are designed and implemented.

Guided by these constraints, the implementation of generics in Java builds fairly straightforwardly from the design for Generic Java proposed by Bracha and Odersky. Syntactically, Java generics are almost identical to C++ templates. It should be noted here that unlike C++ templates, generic types can be 'bounded' - in other words, the compiler can be told that the type passed to a generic must be a subclass of a particular class. If the object passed to the generic does not fit these bounds, the compiler will return an error. Since all classes in Java inherit from class Object, any generic for which a bound is not provided is implicitly considered to be bounded by Object. The syntax for a bounded generic is as follows:

```java<br>&lt;type extends Superclass&lt;type&gt;&gt;<br>```

Generics are defined either as generic classes or generic methods. Like all Java methods, generic methods must be bound to a class (though that class need not itself be generic). The use of generics also mirrors that of templates - types are passed into a generic class or function when it is instantiated, and the type passed is then used throughout the rest of the generic.

The key to understanding how generics work in Java is understanding a construct referred to as raw types. In short, a raw type is a generic type without any type information; for instance if a library provides a generic list class List<T>, the raw type of List<T> is List. Raw types then work on objects of type Object, and code making use of raw types are expected to use the appropriate casts for objects retrieved from raw types. By supporting interoperability with raw types, generics are able to support legacy code which uses the generic idiom, since such code already uses raw types by default. This requirement to support raw types effectively blocked Java from providing so-called heterogeneous instantiation of generics (the C++ style, in which a copy of a generic function or class is created for each type with which it is instantiated). Instead, Java provides a homogeneous implementation, in which generics are converted to raw types at compile-time via a process called type erasure. Type erasure refers to the removal of type information from arguments to generics (in other words, the conversion of all arguments to type Object, or the type of the first bound class if one is provided) and the insertion of relevant casts for any values returned from generics.
To clarify this process, we will now walk through the implementation of a Java generic. Assume that we have a generic class `List<T>`, which we have called with an object of class `String` at some point in our code. The compiler proceeds as normal until it is time to generate JVM bytecode for our Java code. At this point, any use of our `List<String>` class will undergo type erasure. In this process, any reference to the type `T` within the `List` class will be replaced with `List`’s first bound class. Since `List<T>` is not bounded, the first bound is `Object`, and all references to `T` will be replaced by references to `Object`. Had `T` been bounded, it would have been replaced with that bound instead. This creates obvious issues for primitive types - since primitives do not inherit from `Object`, they have no first bound and thus cannot be replaced by any type. To prevent this, the compiler forbids the instantiation of generics with primitives. Instead, if a generic is passed a primitive, that primitive is automatically boxed to its relevant class (so, for instance, an `int` is automatically converted to an `Integer`). Since boxed types are all subclasses of `Object`, they have a first bound and may be replaced as such. The compiler notes each use of type erasure, and any time code makes use of a field of a type-erased object or a return from a type-erased method, an appropriate cast is inserted.

Advantages and Disadvantages

Java’s genericity system has several distinct advantages. Most notably, the designers’ goal of maintaining backwards compatibility with older library code without requiring rewriting that code works flawlessly. Since the concept of raw types is (by design) essentially identical to generic-style code written prior to Java 1.5 using the generic idiom, full interoperability between older-style generic code and code using modern generics is guaranteed, since the modern code is more or less translated to the older style. Additionally, since Java generics are homogeneous, they are only translated to bytecode once as their raw type rather than once per type use, as in the case of heterogeneous implementations such as C++ templates. This avoids many of the significant drawbacks associated with heterogeneous implementations - most notably, slowed compilation times and larger binary files.

Unfortunately, Java’s generics system also has significant disadvantages, mostly due to the required use of type erasure in their implementation. Generic classes and methods are not able to reason about the type of the object which they have been passed outside of a few limited avenues (such as setting a bound for template arguments). This limits what programmers are able to achieve with generics (though it could be argued that by forcing the programmer to abstract away the representation of the data handled by generics,
they are ‘limited’ to good generic programming practice). More problematically, in some cases type erasure can have dramatic effects on type-safety within Java programs. Since all generics are considered to be the same type as their raw type after type erasure, it is impossible to tell at runtime if two generic objects are of the same class or not - for instance, at runtime an object of List<String> and one of List<Integer> will both only be considered objects of List. Thus, any operation that relies on testing whether or not two generic objects were instantiated with the same types will fail, as any such test will always return true. While there are ways around this (most notably, retrieving an object from the container class and testing that object’s type), they are clunky, and most programmers do not know enough about the specifics of how generics are implemented to be able to catch the kind of subtle errors that can appear when generic objects are compared [14]. Additionally, Java generics cannot perform any sort of type reflection on the objects which they have been instantiated with. For example, the Add(a, b) code shown in chapter 3 will not work properly in Java - it will always return an error, as type erasure means that no matter what type the objects passed into Add(a, b) are, they will be considered objects of class Object when the code executes. Finally, the fact that any use of a generic class is folded into the same class limits what exactly can be done within a generic - for example, any static variables within a generic class will be shared by all uses of that class regardless of the type parameters, so they cannot make use of the type parameters themselves.

Chapter 7: Generic Programming in C#

Introduction and History

Finally, we will move on to the third language this paper will cover: C#. Like Java, C# was not initially designed with any kind of generic mechanism at all. Unlike Java, however, generics were proposed as an addition very early in the life of the language (with the formal proposal to extend C# with generics published one year after the language’s release, versus six years later for Java). This freed the designers of C#’s generic mechanism from any constraints of backwards-compatibility, allowing them to focus on designing a robust and effective genericity mechanism. Intriguingly, they did so by implementing generics at the level of the virtual machine using the .NET virtual machine’s just-in-time compiler rather than the standard C#-to-CLR-bytecode compiler, providing a fascinating divergence from the strategies implemented by the designers of C++ and Java. It should be noted that since C#’s genericity mechanism is implemented at the virtual machine level, it is not totally accurate to refer to it as "C#’s" mechanism specifically, as it is the same mechanism shared by every language that uses the .NET Common Language Runtime. That being said, for conveniences’ sake we will refer to the genericity mechanism implemented by the .NET CLR as "C#’s genericity mechanism" specifically, and all code examples in this chapter will be written in C#.

The first proposed addition of generics to C# came in a 2001 paper by Andrew Kennedy and Don Syme, two researchers working at Microsoft. Studying the state of generic programming at the time, the two found that many of the factors that they considered flaws in other implementations of generic programming (such as the inefficiency of C++ style templates and the non-type-safety of what was then known as Generic Java) were a result of attempting to handle generics with the compiler itself. They argued that by implementing generics at the level of the VM and using C#’s JIT compiler to generate type bindings dynamically, it would be possible to combine the efficiency of homogeneous implementations of generic programming with the type-safety of heterogenous implementations [10]. Their suggestions were adopted for addition to the CLI by ECMA, and in 2005 C# 2.0 was released. Version 2.0 was seen as a major step forward for the language, and involved the release of an entirely new .NET CLR virtual machine to implement several new features - including VM-level generics.

Kennedy and Syme first approached the question of adding generics to the CLR with two unique advantages - hindsight and a very young language. By the time Kennedy and Syme had started designing what would become C# generics, C++ templates had been in widespread use for many years, and Java generics had evolved from their roots in Pizza to the GJ proposal that would eventually lead to Java’s implementation of generics. This gave Kennedy and Syme a clear idea of the forms taken by ‘modern’ genericity mechanisms, and the unique drawbacks that came with both heterogenous and homogeneous implementations of generics. Kennedy and Syme argued that these issues were inherent in any generic design that relied on the compiler to translate generic code into ‘standard’ code, since that level of translation would need to rely on either
translating generics into a large volume of code (as in heterogenous implementations) which will generally lead to inefficiency, or translating generics into vague code (as in homogeneous implementations) which will generally lead to a lack of type-safety. Instead, they argued that the most effective way to implement generics would be to do so beneath the level of the compiler. More specifically, they argued that the presence of a VM and JIT compilation in the .NET runtime would allow for a hybrid homogeneous-heterogenous implementation, by having the compiler perform type-checks on generic use but only compile to one implementation of a generic, and then having the JIT compiler use type information saved at the VM level to expand generics when necessary at runtime [10]. This approach manages to avoid many of the pitfalls of older genericity systems: it allows for strong compile-time type safety by checking generic types at compile-time; provides efficient compilation and small binary sizes by only compiling generics once, and guarantees type-safety of object comparison (unlike Java) by instantiating generics as independent classes at runtime.

Design and Implementation

We will not spend time focusing on the syntax used for C# generics, as it is essentially identical to the syntax used for Java generics - the interesting differences between the two are entirely semantic and lie in their implementations. The reliance on the virtual machine and JIT compiler makes the implementation of generics in C# significantly more complex than those present in Java or C++. In order to clarify exactly how C# handles generics, we will walk through compiling and executing a generic class List<T> in C#.

At compile-time, List<T> is checked for method compatibility - in other words, the compiler checks that if List<T> calls any methods of an object of type T, those methods exist on whatever the upper bound class of T is (so if no inheritance constraint is provided, any method called on an object T must be a method of Object, which, like in Java, is the universal superclass). After these checks, List<T> is compiled to .NET bytecode (which is often referred to as Intermediate Language, or IL) for a single class List<T> that is specifically flagged as being a generic class.

At run-time, when an application makes a call to a specific use of a generic (such as creating an object of type List<String>), the .NET runtime checks to see if the version of the class being called has been created yet, and if it has not been it is then created and used (if it has been created previously than that version is used instead). This is done through the use of vtables: the .NET runtime automatically creates a vtable for each class used in an application (both generic and non-generic), so when a new instantiation of a generic class is called the current vtables are checked to see if that type-class combination has a vtable yet and creates one if it does not. This step is what differentiates this implementation from pure homogeneous implementations such as Java's: the fact that different generic types are unfolded into distinct classes at run-time ensures run-time type safety when comparing generic objects, unlike in Java. Interestingly enough, in order to cut down on how much unnecessary code is generated, generic classes are lazily instantiated: the vtables for a generic class only include stubs for the class's methods (to allow for type-checking and appropriate data hiding based on the method signature) and those stubs are only expanded when the method itself is called [10].
Advantages and Disadvantages

Clearly, there are several advantages to this style of generic implementation. Some of the advantages are obvious: C#’s hybrid homogeneous/heterogenous implementation manages to ensure both reduced compilation times and binary sizes, while providing the run-time type safety lacking in Java. While this is the main benefit of the approach, some of the others are much more subtle. One such advantage is the fact that run-time unfolding of generics means that if program execution does not reach any use of a specific generic, it will not be unfolded. This is a sharp contrast from C++ and other languages with pure heterogenous implementation of generics, in which every possible generic type is generated at compile-time and is present in the binary regardless of whether or not it is actually used, and significantly reduces the size, running time, and complexity of the resulting code.

One might assume that this style of implementation, by requiring run-time action at the virtual machine-level, would involve significant decreases in execution speed. Though this is a fairly obvious conclusion to make, it turns out not to be true: according to Kennedy and Syme, code that uses the .NET CLR’s implementation of generic programming is not significantly slower than equivalent C# code without generics, and in fact can be significantly faster than generic code implemented in Java. This is true due to several optimizations throughout the code. Code re-use for expanded generics, coupled with optimizations in the .NET CLR targeted at rapidly creating vtables for classes, ensure that instantiation takes as little time as possible, and techniques such as lazy method instantiation ensure that potentially inefficient operations, such as generating new methods, are spread throughout program execution rather than piling up at class creation causing bottlenecks. Additionally, a major speed advantage comes from the fact that C# does not need to perform boxing of primitive values. Since Java generics are ultimately reduced to use the Object class, any type argument to a generic must be a subclass of Object - requiring that primitive types be automatically boxed, a process which can be comparatively very slow. Since C# generics can be expanded with primitive
types just as easily as with classes, there is no need to box primitives. This gives C# generics a significant speed advantage over Java in the case of primitive types (although, obviously, shows no difference against heterogeneous generics such as those seen in C++). That being said, while it makes intuitive sense that instantiating C# generics on primitive types, by not requiring boxing, will be faster than performing the same operation in Java generics, these claims should be regarded as potentially biased, coming as they are from the designers of C# generics themselves. Unfortunately, very few rigorous, unbiased comparisons of generic programming in the two languages have been carried out, and those which have are generally not useful for any of a variety of reasons (for instance, Dragan and Watt (2005) find that when running a scientific computing benchmark, converted to use generics, C# actually performs worse than Java - but they also point out that they performed these experiments with an early, pre-release experimental version of the C# 2.0 compiler, and acknowledge that C# performance could be significantly improved with some changes to the JIT compiler) [6]. The only other cross-language generics comparison available was written by Dragan in 2007, and even then he does not compare generics performance between languages directly. Instead, he measures the speed difference between generics and specialized code running the same scientific computing benchmark in C++, Java, and C# (as well as a language called Aldor). He reports that (depending on the size of the dataset) in C++ specialized code is 1.3 to 1.7 times faster than templates, in C# specialized code is 2.7 to 5.7 times faster than generics, and in Java specialized code is 8.8 to 11.3 times faster than generics [5]. While this seems to support the claim that C# generics are faster than Java generics, it is not a direct comparison, and the lack of studies performing such comparisons makes it difficult to evaluate comparative claims of generics performance effectively.

While C# generics have several significant advantages, they are not without their drawbacks. The first is type-safety: while C# is able to provide the run-time type-safe generic object comparison that Java cannot, and although some checks (such as method legality) are performed at compile-time, C# is unable to guarantee the same kind of compile-time total type-checking that full heterogeneous generics such as C++ templates can. This is an inherent limitation of C#’s style of genericity, as it is impossible to gain C++-style full type checking without providing full heterogeneous generic implementation. Another significant drawback (though one that is irrelevant to most end-users) is the difficulty present in implementing such a technique. Unlike full compiler-based strategies for generic implementation (both heterogeneous and homogeneous) which can be implemented fairly easily by compiler designers, creating this style of generics requires significant domain knowledge and work on multiple levels of language implementation (compiler, virtual machine, and JIT compilation). Though this is not an issue for a company like Microsoft, which provides Virtual Studio (the reference compiler for C#) and the .NET CLR (and which is able to pour nearly unlimited funds and technical expertise into creating and maintaining C# and its related languages), it can have a chilling effect on third-party attempts to create alternative language implementations (such as the open-source Mono project).

Part 3: Final Thoughts and Conclusion

Chapter 8: Comparing Implementations of Generic Programming

In chapter 4, we discussed the four tensions that language designers must balance when designing new language features: flexibility (or the ability to use a single piece of code for as much functionality as possible), re-use (the ability, closely related to flexibility, to write as little repetitive code as possible), type-safety (the ability for code to generate useful errors when incorrect types are used in an operation, either at run-time or, better, compile-time) and performance (the ability for a program to execute correctly as quickly as possible). Language design is often a balancing act between the four, and the decisions made by designers often involve making sacrifices in one area for the sake of the other three. Having examined in detail how generic programming is implemented across several languages, we can now discuss how the specifics of balancing these four issues played a role in the design of each language’s genericity mechanisms - and what larger patterns about language design can be gleaned from such an examination.

Before we turn to genericity mechanisms, we will first talk about the most common alternative to generic programming seen in programming languages - that of using a dynamic type system, such as the ones seen
in most scripting languages (most notably Python and Ruby). As mentioned before, these systems are extremely flexible (as any one function can work on any value passed to it, without any special notation or consideration required), and require next to no code re-use for the same reason. That being said, these languages require significant trade-offs. Type-safety in dynamically-typed languages is next to nonexistent; it is impossible to catch any kind of type mismatches at compile time for the simple reason that there are no types to mismatch. This means that errors related to types are caught during runtime crashes at best, and at worst are not caught at all - instead, such operations produce unexpected, often strange results, that then affect the rest of the program’s execution. Such errors can be nearly impossible to catch. Finally, there is a hit to performance as well - scripting languages (which are most often dynamically typed) tend not to perform as well as compiled languages (which are most often statically typed) for a variety of reasons. While most of these reasons are unrelated to the languages’ type systems, some are - it is essentially impossible to optimize code based on types in a Python program, since there are no types to optimize on. That being said, statically typed programming languages without genericity mechanisms run into the opposite issue - while they tend to have extremely powerful compile-time type-checking, this comes at the expense of flexibility (as code must be written to work on only one kind of type), and code re-use (as a function can be exactly the same when performed on integers or characters, but it must be re-written for each type it will work on).

With the addition of templates, C++ became one of a handful of statically typed languages to attempt to bridge the gap between these four issues. Like most statically typed languages, C++ without templates provided strong facilities for type-safety and optimization, but offered little flexibility and required a good deal of boilerplate, redundant code. Templates were notable for providing flexibility and code re-use without compromising type-safety. By expanding templates at compile-time, templates allowed users to write flexible code that could be re-used easily across many types, while still providing full compile-time type-checking of code.

Unfortunately, while C++’s template system allowed for flexibility, code re-use, and type-safety, it did so at the expense of considerable hits to performance in several areas. Expanding templates at compile time slowed compilation and dramatically increased the size of compiled program binaries. While compile-time expanding does allow for programmers to use generics to fine-tune how their programs operate on data of specific types, allowing for users to manually extract high performance from code using templates, doing so is generally difficult and requires an extremely high level of skill. Ultimately no matter how well a programmer optimizes his or her code using templates, certain constraints of the templates themselves (such as large binary files) are essentially unavoidable.

While at the surface Java’s generics appear similar to C++’s templates, it should be clear at this point that there are strong differences in how each balance the four concerns we have discussed. Through the process of type erasure, Java essentially implements generic programming as a sort of de facto dynamic typing; in which function arguments have no types and thus a function can be used on any type of data. This means that Java balances these issues in much the same way as dynamically typed languages - it provides extreme flexibility and code re-use (arguably more so than C++, since it could be said that C++ templates involve significant code repetition and redundancy that simply happens at a level at which the programmer does not need to be concerned with, while Java generics do not involve repeating code at all), while providing little in the way of type-safety or performance. Automatic boxing of primitive types can lead to significant performance degradation in Java programs, and though some limited type-checking occurs at compile-time, it is for the most part very difficult to find generics-related type errors in Java programs until they are encountered at runtime. Even worse, in the case of some of the more subtle bugs that type erasure can introduce, it is possible for type errors to lead to unexpected program behavior that is not caught at all.

Of course, despite the many drawbacks of Java’s generics, there is one significant advantage present: that of backwards-compatibility, or the fact that the tremendous amount of quasi-generic Java code written before the introduction of generics can be used alongside generics without having to modify any code. While this is difficult to map directly to any of the four points we have been exploring, it could be said to represent a specific edge case of code re-use: that of being able to re-use legacy code with minimal modification. Taken this way, the tradeoffs made by Java make quite a bit more sense - by sacrificing type-safety and performance,
Java’s designers ensured that code re-use was central to the design of generics.

As arguably the most advanced (and the most complex) of the genericity mechanisms that we have discussed, C# generics provide an interesting test bed for further exploring these four points. Like both C++ and Java, generic programming in C# allows for significant flexibility and code re-use (and like Java, C# generic code is actually re-used rather than simply hiding redundant code from the user). However, this is essentially where the similarities between the two end. While C#’s type-checking is not as strong as C++’s (since it occurs primarily at run-time) it is still much stronger than Java’s, especially considering that an entire class of errors caused by type erasure are not present in C#. Additionally, by not requiring boxing of primitive values and providing lazy instantiation of generic classes and methods, C# provides significantly better performance than Java, although programmers are not able to perform the same kind of bleeding-edge manual fine tuning possible in C++.

Ultimately it is clear that while C#, like all of the languages we have discussed, requires tradeoffs to be made in feature implementation, it manages to minimize these tradeoffs a surprising amount. While C# lacks the strong type-checking of C++, or the legacy code re-usability of Java, it also lacks any of the glaring flaws that are present in any of the other implementations of generic programming which we have explored. This occurs in large part because of the fact that C# leverages advanced language design technologies (specifically, a virtual machine and just-in-time compiler) that other languages cannot (such as C++, which has no virtual machine) or do not (such as Java, the designers of which were not willing to modify the Java Virtual Machine to allow for generics). By delegating specific functions to the VM, C# is able to avoid many of the drawbacks generally associated with the homogeneous style of generic implementation which they ultimately used.

By examining these languages, we can start to see a clear pattern emerging in how language designers balance flexibility, code re-use, type-safety and performance. More specifically, we can see that it is often necessary to make significant sacrifices in one domain (such as efficiency for C++, or type-safety in Java and dynamically typed languages) for the sake of the other three. However, we can also see that this need not be the case; as C# is able to provide a full genericity mechanism with few significant drawbacks. We can see in the case of C#, however, that the differentiating factor is technology: C# minimizes its drawbacks by using a more technically sophisticated implementation than any other language. In fact, it should be noted that Java, which has the same features (a VM and JIT compiler) available, could implement a system very similar to C#’s - Java’s drawbacks are the result of specific design considerations, not a lack of technology. It may be the case, then, that sacrifices are not necessary a universally necessary consideration in language design. Instead, it could be argued that advancing technology allows for the design of language features which require fewer tradeoffs. Perhaps future advances in language theory and design will some day allow for the design of a ‘perfect’ genericity mechanism - one which combines the strong type-safety of C++ with the flexibility of dynamically typed languages and the code re-use of Java.

Of course, it is a bit presumptuous to claim at this point that we have found some sort of universal truth of programming language design. We have only examined a particular, very small subset of programming languages - C-derived, statically-typed, object-oriented compiled languages, to be precise. Though we have touched briefly on dynamically-typed programming languages, even these share key similarities with the languages that we have focused on - Python and Ruby, for instance, are both themselves object-oriented programming languages that share a similar model of programming as C++, Java, and C#. It may be the case that the conclusions we have reached - that sacrifices in the four domains of language design which we have focused on are a necessary part of language design which are minimized over time by advancing technology - are unique to this type of programming language, or even this particular subset of this type of language. In order to test this claim, we will now turn our attention to a radically different type of language, which is fundamentally distinct from the languages we have explored so far: the well-known pure functional programming language Haskell.
Chapter 9: Generic Programming in Haskell

Haskell is a strong statically-typed, pure functional programming language first released in 1990. The ‘pure functional’ aspects of Haskell distinguish it from most other mainstream programming languages, which are nearly all (with the notable exception of Lisp) either object-oriented or multiparadigm languages with support for a wide variety of programming styles. Haskell, by contrast, seems (at least at first) to be extremely limiting. Variables in Haskell cannot be re-assigned (once a name has been bound to a value, the name cannot be re-bound), and functions in Haskell can have no ‘side-effects’ - they can only take one or more values, perform some operation on them, and then return a value. Haskell also has no looping constructs; programmers instead use recursion or list iteration constructs. Complex operations are generally built up from smaller, simpler functions which, along with the previously-mentioned fact that functions cannot have side effects, make Haskell programs very easy to reason about mathematically (and in fact, Haskell is extremely popular among programmers with strong backgrounds in mathematics and theoretical computer science).

There are several factors of the language which make it an interesting test case for study as a counterpoint to C++, Java, and C#. Unlike the other languages we have discussed, Haskell provides no support at all for object-oriented programming - and in fact any such support would run contrary to the entire philosophy of the language, which claims that “functions are first-class citizens” and treats program construction as fundamentally being a process of designing and combining functions. This also means that a great deal of focus in Haskell programming is placed on designing algorithms, which makes the language a fascinating place to explore Stepanov’s original notion of generic programming as being a way of ‘programming in algorithms’. Additionally, the genericity systems we have seen in the other three languages we have discussed are all notable in that they all (some to a greater degree than others) hinge on the languages’ class systems, treating classes as the basic generic argument and, in Java’s case, only allowing generic programming to be performed on objects of classes, rather than primitive types. Haskell, being a language without classes (though it has some constructs, namely typeclasses, which could be seen as analogous), could be expected to handle generics very differently.

This is especially true due to Haskell’s unique type system, which is markedly different from the type systems used in any of the other languages we have discussed. Haskell’s type system has its roots in a mathematical notation for functions on types known as the typed lambda calculus. Like the other languages that we have discussed, every variable in Haskell must have a type, and functions must be defined to take input of specific types and return output of specific types. Haskell provides full, complete compile-time type checking; the types of all operations are checked during compilation and code will not compile if type issues are present. This means that a Haskell program will never crash at runtime due to type-related errors, as such errors are guaranteed to have been caught at compile time if they exist. Unlike the languages that we have discussed, however, Haskell provides a feature known as type inference, using an algorithm called the Hindley-Milner type inference algorithm - while it is possible for programmers to provide type annotations for variables like in the other languages we have discussed, it is not necessary (which the exception of a few complex cases in which the compiler is not able to perform full inference and type annotation is required) [8]. The compiler is able to infer the type of a variable based on its value. This can be seen as the opposite of type erasure - where type erasure removes type information from a variable, type inference adds it. While at the surface level this appears to be fundamentally the same as dynamically typed languages, the difference is significant: Haskell will still return a compiler error if an integer is passed to a function expecting a string, the programmer simply does not have to specifically mark the integer as being an integer.

While type inference may seem like an interesting feature that has no real relevance to our topic, the opposite is true: in fact, type inference is central to how Haskell implements parametric polymorphism. As in most statically typed programming languages, Haskell functions have what are known as type signatures, information accompanying the function which specifies what types the input and output to the function will be (as an example, a function max that returns the higher of two integers would have the type signature max :: Int -> Int -> Int). Elements in a type signature can take one of two forms: actual types, or type variables, which are variables which represent types. The two are distinguished by capitalization - actual
types always begin with capital letters, while type variables always begin with lowercase ones. Type variables in a signature mean that the element represented by that item in the signature may be of any type (though it is possible to constrain type variables by specifying that they must belong to specific typeclasses; while the specifics of typeclasses are too complex to discuss here they can be thought of as defining a set of behaviors to which a type must conform, like an interface or pure abstract superclass in object-oriented programming). Thus, a version of the max function which could take two elements of any type and return the highest would have the type signature max :: t -> t -> t. Using type variables in the function signature is all that is needed to specify that a function is generic; no other special annotations (such as those seen in C++, Java, and C#) are necessary [8, 16].

Of all of the generics systems we have studied so far, Haskell has both one of the most straightforward and the most complex. Generic functions in Haskell are not expanded in any way at compile-time, nor are they handled by any sort of virtual machine. Instead, one copy of a generic function is created at compile-time, and when it is called the values used are passed to it as-is, just like in Java. Unlike Java, however, type inference ensures that full type-safety is maintained - the Haskell compiler is able to determine the types of all variables at all times, protecting the programmer from any of the sorts of errors that can arise in Java generics. At first glance, Haskell’s genericity mechanism appears to be essentially ‘perfect’ - it provides the full compile-time type-safety of C++ templates with the flexibility of Java generics, without the run-time overhead necessitated by C#. Based on this understanding, it’s tempting to claim that Haskell’s genericity mechanism is clearly the superior of the four. This is not entirely true, however - Haskell generics are in fact extremely complex, and not necessarily well-suited to any other programming languages. As we have mentioned, type inference (via the Hindley-Milner algorithm) is key to the effectiveness of Haskell’s genericity system. That being said, the Hindley-Milner algorithm is extremely complex, and while the specifics of the algorithm are far outside of the scope of this paper, in order for it to be used on a language that language must have an extremely strict typing system - much stricter than that used in C++, Java, and C#. This means that while type inference based generics may be extremely effective, they are limited to use in a small subset of (generally pure functional) programming languages - as opposed to the systems seen in the other three languages we have examined, which despite being less efficient are far more flexible.

Chapter 10: Conclusion

Now that we have examined Haskell, we can turn back the question raised in Chapter 8: is it a general fact of language design that more advanced design techniques and technology allow for trade-offs into be minimized? Or is this only true in some subset of languages? Unfortunately, Haskell provides no clear answers to this question. On the one hand, it could be argued that Haskell clearly fits this pattern: a specific advanced compilation technique (type inference) allows Haskell generics to avoid having to make any of the tradeoffs seen in languages that do not have such a feature present. On the other hand, it could be argued (especially by programmers with a background in imperative or object-oriented programming) that Haskell’s system does in fact require significant tradeoffs: using Hindley-Milner type inference limits Haskell’s type system to an extremely strict form that renders illegal many constructs seen in other programming languages with weaker type systems, creating a trade-off by sacrificing usability. Of course, defenders of Haskell would likely claim that this limited type system is in fact an advantage in and of itself, as it provides formality not possible in other programming languages - and in fact, programmers skilled in Haskell often regard the type systems seen in most other languages as being inferior to Haskell’s.

Even if we accept the claim that Haskell’s limited type system is not itself a trade-off, Haskell illustrates another flaw in our argument that advancing language technology minimizes tradeoffs in design: type inference is not by any measure a ‘new’ or ‘advanced’ technique the way that the virtual machines and JIT compilation that power C# generics are. The first version of Haskell was defined in 1990, earlier than either Java or C#, and the Hindley-Milner type inference algorithm was first described in a paper published in 1982. Clearly, the implementation of generics in Haskell demonstrates that it is inaccurate to claim that as time goes on, more advanced techniques allow language designers to minimize tradeoffs.

If that is the case, then, we must turn to the inevitable next question: can we draw any conclusions
about tradeoffs in language design from this set of languages at all? While the pattern present may not be as straightforward as the one seen in the set of C-derived languages discussed earlier, one clear common thread can be found: more sophisticated language design techniques tend to minimize the tradeoffs necessary for programmers using a language. Here, “sophisticated” refers to features and techniques that are not straightforward, and which rely on particular methods above and beyond those which a standard language designer might use. As an example, the raw heterogeneous and homogeneous systems used in C++ and Java respectively are fairly unsophisticated; they do not rely on any particular advanced features or algorithmic subtlety. As a result, they require that certain tradeoffs be considered when the language is used: programmers considering generics in C++ must be willing to sacrifice code re-use and efficient compilation for flexibility and type-safety, while programmers using Java must trade type-safety for efficiency and compatibility. The use of more elaborate systems, such as virtual machines and complicated type inference algorithms, in C# and Haskell minimize these tradeoffs for end-users. With this definition in mind, we can see why it might appear that advances in compiler technology over time are what minimizes tradeoffs: over time, as theoretical research in language design advances, more and more complex systems can be designed and used. That being said, the comparative sophistication of a technique need not necessarily be tied to the passing of time and advancement of the field: the Hindley-Milner algorithm can be said to be an extremely complex technique in language design, despite being over thirty years old. What is clear, however, is that some level of sophistication, whether it comes from advances in research over time (as is seen in the genericity mechanism used in C#) or from finding new ways to leverage older research (as Haskell does), is a necessary for the creation of robust, effective genericity mechanisms - and, quite possibly, for language design as a whole.

References


Appendix A: C++ Template Experiment Data

Experiment 1 (All Values in Bytes)

<table>
<thead>
<tr>
<th>Source</th>
<th>Binary 1</th>
<th>Binary 2</th>
<th>Binary 3</th>
<th>Binary 4</th>
<th>Binary 5</th>
<th>Binary 6</th>
<th>Binary 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 1</td>
<td>168</td>
<td>8200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 2</td>
<td>362</td>
<td>8370</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 3</td>
<td>487</td>
<td>8540</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 4</td>
<td>652</td>
<td>8700</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 5</td>
<td>816</td>
<td>8870</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 6</td>
<td>980</td>
<td>9040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source 7</td>
<td>1110</td>
<td>9210</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experiment 2 (All Values in Bytes)

<table>
<thead>
<tr>
<th>Source 1</th>
<th>168</th>
<th>Binary 1</th>
<th>8200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 2</td>
<td>324</td>
<td>Binary 2</td>
<td>8370</td>
</tr>
<tr>
<td>Source 3</td>
<td>617</td>
<td>Binary 3</td>
<td>8950</td>
</tr>
<tr>
<td>Source 4</td>
<td>916</td>
<td>Binary 4</td>
<td>14000</td>
</tr>
<tr>
<td>Source 5</td>
<td>1180</td>
<td>Binary 5</td>
<td>15800</td>
</tr>
<tr>
<td>Source 6</td>
<td>1470</td>
<td>Binary 6</td>
<td>26400</td>
</tr>
<tr>
<td>Source 7</td>
<td>1750</td>
<td>Binary 7</td>
<td>34000</td>
</tr>
</tbody>
</table>
Experiment 3 (All Values in Bytes)

<table>
<thead>
<tr>
<th>Source</th>
<th>Value</th>
<th>Binary</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 1</td>
<td>168</td>
<td>Binary 1</td>
<td>8200</td>
</tr>
<tr>
<td>Source 2</td>
<td>324</td>
<td>Binary 2</td>
<td>8370</td>
</tr>
<tr>
<td>Source 3</td>
<td>648</td>
<td>Binary 3</td>
<td>8960</td>
</tr>
<tr>
<td>Source 4</td>
<td>1560</td>
<td>Binary 4</td>
<td>19600</td>
</tr>
<tr>
<td>Source 5</td>
<td>6740</td>
<td>Binary 5</td>
<td>131000</td>
</tr>
</tbody>
</table>
Appendix B: C++ Template Experiment Code

**Experiment 1:**

```cpp
using namespace std;

template <class T> class tempA {
  public:
    T a;
    tempA (T first) 
    {a=first ;}
};

int main () {
  tempA <int> tempA (5);
  return 0;
}
```

![Graph showing experiment 3 source size and binary size](image-url)
Algorithm 2 Code for Experiment 1, N = 2

using namespace std;
template <class T> class tempA {
    public:
    T a;
    tempA (T first) {
        a = first;
    }
};
template <class U1, class U2> class tempB {
    public:
    tempB (U1 first, U2 second) {
        tempA<U1> a(first);
        tempA<U2> b(second);
    }
};
int main () {
    tempB<int, float> newtemp (5, 3.0);
    return 0;
}

Algorithm 3 Code for Experiment 1, N = 3

using namespace std;
template <class T> class tempA {
    public:
    T a;
    tempA (T first) {
        a = first;
    }
};
template <class B1, class B2> class tempB {
    public:
    tempB (B1 first, B2 second) {
        tempA<B1> a(first);
        tempA<B2> b(second);
    }
};
template <class C1, class C2> class tempC {
    public:
    tempC (C1 first, C2 second) {
        tempB<C1, C2> a(first, second);
        tempB<C2, C1> b(second, first);
    }
};
int main () {
    tempC<int, float> newtemp (5, 3.0);
    return 0;
}
Experiment 2:

**Algorithm 4** Code for Experiment 2, $N = 1$

using namespace std;
template <class T> class tempA {
    public:
        T a;
        tempA (T first)
            {a=first;}
};
int main () {
    tempA <int> tempA (5);
    return 0;
}

**Algorithm 5** Code for Experiment 2, $N = 2$

using namespace std;
template <class T> class tempA {
    public:
        T a;
        tempA (T first)
            {a=first;}
};
template <class U1, class U2> class tempB {
    public:
        tempB (U1 first, U2 second) {
            tempA<U1> a(first);
            tempA<U2> b(second);
        }
};
int main () {
    tempB<int, float> newtemp (5, 3.0);
    return 0;
}
**Algorithm 6** Code for Experiment 2, N = 3

using namespace std;
template <class T> class tempA {
  public:
    T a;
    tempA (T first)
    { a = first; }
};
template <class B1, class B2> class tempB {
  public:
    tempB (B1 first, B2 second)
    { tempA<B1> a(first); tempA<B2> b(second); }
};
template <class C1, class C2> class tempC {
  public:
    tempC (C1 first, C2 second)
    { tempB<C1, C2> a(first, second);
      tempB<C2, C1> b(second, first);
      tempB<C1, char> c(first, 'a');
      tempB<char, C1> d('a', first);
      tempB<C2, char> e(second, 'a');
      tempB<char, C2> f('a', second); }
};
int main () {
  tempC<int, float> newtemp (5, 3.0);
  return 0;
}

**Experiment 3:**

**Algorithm 7** Code for Experiment 3, N = 1

using namespace std;
template <class T> class tempA {
  public:
    T a;
    tempA (T first)
    { a = first; }
};
int main () {
  tempA<int> tempA (5);
  return 0;
}
Algorithm 8 Code for Experiment 3, N = 2

using namespace std;
template <class T> class tempA {
    public:
        T a;
        tempA (T first) {
            a = first;
        }
};
template <class U1, class U2> class tempB {
    public:
        tempB (U1 first, U2 second) {
            tempA<U1> a(first);
            tempA<U2> b(second);
        }
};
int main () {
    tempB<int, float> newtemp (5, 3.0);
    return 0;
}

Algorithm 9 Code for Experiment 3, N = 3

using namespace std;
template <class T> class tempA {
    public:
        T a;
        tempA (T first) {
            a = first;
        }
};
template <class B1, class B2> class tempB {
    public:
        tempB (B1 first, B2 second) {
            tempA<B1> a(first);
            tempA<B2> b(second);
        }
};
Template <class C1, class C2, class C3> class tempC {
    public:
        tempC (C1 first, C2 second, C3 third) {
            tempB<C1, C2> a(first, second);
            tempB<C2, C1> b(second, first);
            tempB<C1, C3> c(first, third);
            tempB<C3, C1> d(third, first);
            tempB<C2, C3> e(second, third);
            tempB<C3, C2> f(third, second);
        }
};
int main () {
    tempC<int, float, char> newtemp (5, 3.1, 'a');
    return 0;
}