Master's Project at ICT, KTH
Examensarbete vid ICT, KTH

Automated source-to-source translation from Java to C++
Automatisk källkodsöversättning från Java till C++

JACEK SIEKA
jacek@kth.se

Master's Thesis in Software Engineering
Examensarbete inom programvaruteknik

Supervisor and examiner: Thomas Sjöland
Handledare och examinator: Thomas Sjöland

Abstract

Reuse of Java libraries and interoperability with platform native components has traditionally been limited to the application programming interface offered by the reference implementation of Java, the Java Native Interface.

In this thesis the feasibility of another approach, automated source-to-source translation from Java to C++, is examined starting with a survey of the current research. Using the Java Language Specification as guide, translations for the constructs of the Java language are proposed, focusing on and taking advantage of the syntactic and semantic similarities between the two languages.

Based on these translations, a tool for automatically translating Java source code to C++ has been developed and is presented in the text. Experimentation shows that a simple application and the core Java libraries it depends on can automatically be translated, producing equal output when built and run. The resulting source code is readable and maintainable, and therefore suitable as a starting point for further development in C++.

With the fully automated process described, source-to-source translation becomes a viable alternative when facing a need for functionality already implemented in a Java library or application, saving considerable resources that would otherwise have to be spent rewriting the code manually.

Sammanfattning

Återanvändning av Java-bibliotek och interoperabilitet med plattformspecifika komponenter har traditionellt varit begränsat till det programmeringsgränssnitt som erbjuds av referensimplementationen av Java, Java Native Interface.

I detta examensarbete undersöks genomförbarheten av ett annat tillvägagångssätt, automatisk källkodsöversättning från Java till C++, med början i en genomgång av aktuell forskning. Därefter föreslås med Java-specifikationen som guide översättningar för de olika språkkonstruktionerna i Java, med fokus på utnyttjandet av de syntaktiska och semantiska likheterna mellan de två språken.

Baserat på dessa översättningar har ett verktyg för att automatiskt översätta källkod från Java till C++ utvecklats och detta presenteras i texten. Experiment visar att en enkel applikation och de Java-bibliotek den beror på kan översättas automatiskt, och att applikationen kan byggas och köras med ekvivalent utdata. Den översatta källkoden är möjlig att läsa och underhålla, och därför lämplig som en utgångspunkt för vidare utveckling i C++.

Med den automatiska process som beskrivs blir källkodsöversättning ett effektivt alternativ då man har behov av funktionalitet som redan implementerats i ett Java-bibliotek eller program, med signifikanta besparingar av de resurser man annars behövt lägga på att manuellt implementera om den existerande lösningen.

Acknowledgements

I would like to thank my supervisor Thomas Sjöland at ICT/SCS for his help, advice and patience over the years.

Thanks to Vladimir Vlassov, also at ICT/SCS, for taking time from his busy schedule to read and offer comments along the way.

Thanks to Erik Angelin all technical discussions on and off topic that allowed me to refine and improve what is presented here.

To Oskar and Milla, for reminding me the meaning of curiosity, and Dorota for patience and love.

Table of Contents

Chapter 1. Introduction

The Java ecosystem ranks as one of the most popular development platforms in 2012 [1]. Backed by large corporations and a vibrant open source community, there are hundreds of thousands libraries available solving tasks in environments spanning from mobile and embedded devices through desktop systems to large server halls.

The Java language has its roots in C and C++, but takes a more simple approach in its design goals [2]. Where C++ is seen as a multi-paradigm language, Java with its class based design is intended to be used in an object oriented setting.

The simplicity of the language in terms of syntax and features makes it easy to learn and understand, and to build custom tools for static analysis and source code transformation. The syntactic similarities between Java and C++ make for an attractive target for source-to-source translation. It becomes easy to trace the origins of the translated C++ code back to the source that produced it - an important characteristic assuming familiarity with the original Java code base.

The similarity between Java and C++ is not only syntactic. Java programs are typically written following the object oriented paradigm which is also supported by C++, improving the fit between translated and native code.

The benefit of automatically translating source code cannot be underestimated. Rewriting code manually requires massive effort and means having to spend resources on solving a problem that has already been solved.

An automatic translator thus opens possibilities to reuse libraries that would otherwise not have been available for consideration, broadening the usefulness and extending the lifetime of existing code.

1.1 Questions, goals and methodology

The initial idea for this thesis was to investigate how the constructs of the Java programming language could be translated into C++, what differences need special treatment and what tradeoffs need to be made in order to be able to reuse such translated code in a C++ context or use it as a base for further development.

In short, it seeks to answer the question whether source-to-source translation from Java to C++ is a viable alternative for reusing existing Java code in a C++ environment, and what the limitations of such a translation would be.

As Terekhov and Verhoef state [3], the problem statement for source-to-source translation is deceptively simple: translate from one language to another without changing the external behavior of the application. To approach the problem, one needs to inventory the language constructs that need translation and provide definitions on how to translate each. This thesis will thus examine the language constructs of Java and see if these can be translated to C++.

Correctness of translation may seem like an absolute requirement of a source-to-source translator, but depending on the goals of the translation, that must not necessarily be true. Readability and maintainability of the translated code may be equally or more important goals and this thesis will examine the tradeoffs involved for particular language statements.

During the course of the thesis a Java to C++ converter, `j2c` [4], was developed to verify the proposed translations and experimentation results will be presented here.

The work has been based on The Java Language Specification, Third Edition by James Gosling, Bill Joy, Guy Steele and Gilad Bracha [2] that covers the Java language up to version 1.6. The translation targets C++ 2011, as specified by ISO/IEC 14882:2011 [5].

1.2 Outline

Chapter 2 starts with a discussion of the problem background and an outline of the scarce research done previously in the area.

Chapter 3 provides an overview section that presents the large picture of source-to-source translation in general and our solution in particular.

Chapter 4 is a reference chapter providing translations for the constructs of Java that need special attention. Where motivated, the relevant parts of the Java Language Specification are quoted.

Chapter 5 contains a presentation of the implemented converter

Finally, Chapter 6 contains conclusion and thoughts on future research.

Appendixes A and B cover bibliography and extended code listings.

Throughout familiarity with both the Java and C++ languages is assumed.

Chapter 2. Background

Code reuse has been a topic of research since before the seventies - it forms the basis for modern software engineering practice [6]. Regardless if the reused code remains external to an application or the code of an old application can be used to create a new one, the gains are obvious. By reusing existing components, software development resources can be redirected to inventing new features and improving existing ones, instead of reinventing the wheel.

Translating the source code to a high level language such as C++ offers the distinct advantage that the translated code can be read, modified and tightly integrated with the rest of the application. Use of a high level language comes at a cost however - the abstraction penalty for using complex language constructs and features can be significant. We therefore begin by examining the various techniques for accessing Java from C++.

2.1 Code reuse strategies

There are several strategies to follow when facing a requirement to reuse a Java software component in a C++ application, each with its own tradeoffs. We will briefly describe some of the alternatives to source-to-source translation.

2.1.1 Java Native Interface

The Java Native Interface (JNI) allows C and C++ applications to embed a Java Virtual Machine (JVM) and run Java code directly through the use of a well defined application programming interface (API) [7]. The API allows the calling application to interact with Java by enabling the creation of class instances, calling of methods and interpreting of results. The same API also allows Java code to call native code, providing a means for calling existing C++ code from Java.

This approach guarantees that the Java code will run according to the Java specification, but becomes impractical for large scale interaction between C++ and Java due to verboseness of the bridging code and limited access to common language features such as inheritance and compile-time error checking. This solution also carries a large overhead in terms of memory use which may be impractical if the required component only makes up a small part of the application.

SWIG, the Simplified Wrapper and Interface Generator [8], is an application that can reduce the amount of work needed to bridge Java and C++ code. It works by automatically generating the JNI glue code and in some cases Java code needed for interaction between the two languages based on the content of C and C++ header files.

2.1.2 Compile-to-native

GCJ is a native compiler for Java [9]. It is able to compile Java source code into native libraries which then can be reused by C++ code. GCJ provides special means to interface with the generated machine code - it provides natural C++ access to classes, methods, object allocation, exceptions. There are several limitations as well - classes that interact with Java may not have non-java members and the support for interfaces is very limited. Also, GCJ does not provide the full Java platform library, thus incompatibilities arise if the Java code interfaces with unsupported parts of the Java platform.

One instance of abstraction penalty in the solution presented in the following chapters is the use of virtual inheritance and the relatively expensive `dynamic_cast` operator. As an example of reduced abstraction penalty due to a lower level translation, GCJ is able to use a more efficient representation of virtual method call tables and by exploiting assumptions about the type of casts that will be made, GCJ can avoid some of the overhead associated with dynamic casting in C++.

2.1.3 Rewrite the code manually

Some projects, for example log4cplus [10] and CppUnit [11], opt to reuse the concepts and architecture of existing Java libraries but rewrite the source code by hand. This can be advantageous as it allows for rewriting the code using native idioms and language features. It is also a very labor intensive approach prone to human mistakes. Any updates to the original library must be applied manually, making the approach impractical if the Java source code changes frequently.

2.2 Prior art

The idea of translating between programming languages is not new. Boyle and Muralidharan [12] showed how translating between LISP and Fortran not only allowed the reuse of existing application code in a new environment, but also how the existing code could be made more efficient as part of the transformation process.

Varma [13] describes how translating Java to C can be beneficial when seeking to use existing code on embedded platforms, offering small code size compared to other native code generation strategies and possibility to execute Java code natively on systems where no Java Virtual Machine is available. His work is based on Toba [14] which provides Java-to-C translation for early Java versions. However, the semantic leap between Java and C is great - many core Java features such as classes, inheritance and exceptions have no native counterpart in C and must thus be simulated leading to code that is difficult to read and even more difficult to maintain. Such an approach is therefore only useful when the translation result will only be used as an intermediate format for further machine translation.

Peterson, Downing and Rockhold [15] provided an overview of a Java to C++ translator in 1998. Many of the points they make remain valid today, but much has also been outdated by advances in both Java and C++. Most importantly, they are successful at producing working C++ translations of several Java programs showing that the problem is tractable.

In the context of Java translation, it is interesting to look at efforts to convert between Java and other languages. Trudel et al. investigate in their paper from 2011 the translation of Java to Eiffel [16]. Just like Java, Eiffel is an object oriented language featuring classes, objects, methods and exceptions. With `j2eif`, the translator implemented as part of the research, they are able to successfully translate and run both simple and GUI applications. Nonetheless, the authors note, differences in semantics to these core concepts require careful analysis in order to produce a successful translator. Dynamic loading, serialization, readability and resulting binary sizes are cited as problematic areas needing further research.

An interesting aside is that Eiffel compilers often use C as an intermediate language and delegate the generation of machine code to C compilers. Thus `j2eif` can be used to produce a C representation of a Java program with the help of a suitable Eiffel compiler.

On the commercial side, Tangible Software Solutions [17] offers a Java to C++ converter labeled as "Accurate and comprehensive" but lacks support for several key Java features such as anonymous and nested classes, static initialization blocks and certain constructors and finally blocks. Some attempts are made at memory management by inserting delete expressions using heuristics, but support is incomplete at best. Where manual intervention is required, the translator inserts comments noting what must be done. The manual notes that there is also limited support for API conversion where Java `String`:s are converted to C++ `string`:s and arrays to `vector`:s, but offers no details on the limitations of the feature.

The approach of this work differs from the Tangible converter by concentrating on providing extensive language support in order to be able to reuse as much existing Java code as possible without manual intervention, including available implementations of the core Java classes.

The Tangible converter instead takes a more pragmatic approach where difficult cases are left to the user to convert and correct by hand. Heurestics and guesses are used in an effort to solve some of the memory management and runtime dependency issues, succeeding in some cases but generating incorrect code in others.

Chapter 3. Overview

This chapter contains an overview of the general problem of source-to-source translation, and highlights some of the high-level problems that need solving when translating from Java to C++.

3.1 Translation steps

Migrating a code base from one platform to another is a multifaceted problem. There are many things to consider for a successful translation, such as overall design paradigms used, documentation, idiomatic use of the source and target languages and API availability.

Terekhov and Verhoef outline many of the difficulties encountered when translating from COBOL to C and suggest a three-step approach to language migration [3]. First, the source code is restructured to minimize friction between source and target languages. Then syntax between source and target language is swapped and finally the target code is restructured to better fit with its native idioms.

In the case of Java to C++ conversion, the first and the last step become less important as many of the idioms of Java naturally carry over to C++ with little friction. We can thus concentrate on the actual translation step, producing code that fits as tightly with C++ idioms as is possible, already here.

Nonetheless, Peterson et al. suggest that certain aspects of Java to C++ conversion are better carried out beforehand, for example to avoid name conflicts due to differences in name resolution. There are weaknesses to this approach however. It may not always be practical to carry out refactoring of a source library for the purpose of translation, especially when the library has been developed or continues to be developed externally. Thus, the better the translator is able to handle the corner cases of the source language, the more useful it becomes as fewer pre and post translation modifications are needed.

In the last step, knowledge and assumptions about the code being translated could be used to rewrite the translated code to fit better with the intentions of the original implementation, but as a general-purpose translator is being treated here, the assumption is that such knowledge is not available.

3.2 Intermediate language

The task of a compiler is typically to transform source code written in a high level language to a lower level language, often machine code for a particular environment. For example, a C++ compiler will translate C++ statements and expressions into assembly code representing the machine instructions of a particular hardware architecture and a Java compiler translates Java source code into bytecode, a stack based instruction set suitable for execution on a Java Virtual Machine.

Modern compilers are often divided into front and back ends. The front end is responsible for translating the particulars of a language into an intermediate format while the back end translates the intermediate format into machine code. To add support for another input language, only a new front end is needed, and by adding a new back end, all existing front ends can be used on a new architecture. In fact, one could see Java bytecode as such an intermediate format - apart from Java, several other languages have been compiled to Java byte code such as Python (through Jython [18]) and Scala [19].

Taking the same approach with a source-to-source translator is problematic. In the case of Java and C++, it is the exploitation of the similarities of the languages that makes the resulting C++ code useful on its own and not only a vessel for further translation. The purpose of an intermediate format is to bring language complexities down and to provide a nucleus of features that are easy for the back ends to consume. For meaningful source-to-source translation, an intermediate format would necessarily have to be expressive enough to carry the nuances of each language it supports, and thus become more complicated than the source language itself.

Toba [14], the Java to C translator mentioned previously, takes the intermediate language approach by translating Java bytecode to C, but the generated code becomes unreadable and unmaintainable as the bytecode instructions are translated directly to C without analyzing their meaning in context. This leads to code that loses all the advantages a higher level language has to offer, as only the most basic building blocks of the language are used. Looking for example at Table 1, a sample presented in the paper on Toba [14], the translated code produces equivalent results, but the intention and clarity of the original Java code is lost in translation.

Java	**Toba (C)**
class d	Int div_ii_3WIeN(Int p1,Int p2)
{	{
static int div(int i, int j)	Int i0, i1, i2;
{	Int iv0, iv1;
i = i / j;	iv0 = p1;
return i;	iv1 = p2;
}	L0:
}	i1 = iv0;
	i2 = iv1;
Method int div(int, int)	if(!i2)
0 iload_0	
1 iload_1	throwDivisionByZeroException();
2 idiv	i1 = i1 / i2;
3 istore_0	iv0 = i1;
4 iload_0	i1 = iv0;
5 ireturn	return i1;
	}

Table 1: Java program, Java bytecode and corresponding Toba output in C [14]

3.3 Runtime support

Java comes with an extensive standard library, the Java Platform. C++ also has a standard library but it is comparatively small and lacks support for many commonly used technologies and tools such as database access, XML processing, GUI programming and logging. Thus, it is not possible to provide full native API migration, even should the Java code only use standard components.

Even for simple cases where classes in the Java and C++ standard libraries match conceptually, such as `ArrayList` in Java and `vector` in C++, the gap between operations supported and idiomatic use of the class is significant, and translation becomes possible only for limited cases where only a subset of the features are used.

One obstacle is the fact that all classes in Java inherit from the common `Object` class - collections and strings included. Replacing Java `String` with C++ STL `string`:s would require converting the C++ `string` instance to a Java-like `Object` reference whenever code depends on the inheritance properties of the Java `String`, for example when storing a reference to the string in a collection. Such a conversion would also need to make sure that a single reference is reused to preserve reference equality semantics.

In short, what seems a simple conversion has many subtle issues that are not easily resolved. We must find another option to provide runtime support - three alternatives present themselves. Which strategy is the best depends largely on the application or library being translated – the relative merits of each must be considered in a larger context.

3.3.1 Implement dependencies manually

The first strategy is to analyze the dependencies of the code and implement them natively in C++. As the examples in Table 2 show, most Java applications directly use only a small subset of the ca 12000 classes that the OpenJDK [20] implementation of the Java Platform consists of.

Library	**Top level classes**	**Java Platform dependencies**
SWT 3.7.2, GTK 64-bit edition	532	103
H2 database, 1.3.168	394	266
logback core, 1.0.7	225	143
itextpdf, 5.3.3	414	204

Table 2: Dependency statistics

An important advantage of this method is that it can be applied to any dependency where the source code is not available. The class file of a compiled Java dependency contains enough information to reconstruct a C++ header with a class declaration. Class, method and field signatures are all present - this is precisely the information contained in a typical C++ header. This is also the same information that the Java compiler itself requires and uses when verifying that the dependency is correctly referenced. In fact, the Java Development kit itself comes with a tool that extracts such information from a Java class file, `javap`.

From the method signatures, stub files can be generated that contain minimal implementations of the dependency - methods with no return type can be left empty, and those that return something can return the default constructed value of the return type. Table 3 shows an example of such a generated header and stub file, based on information easily retrievable from a Java class file.

This strategy is most beneficial when there are few dependencies in the code being converted. An example where this strategy applies could be the implementation of an advanced algorithm, where complicated logic needs translation but external dependencies are scarce.

Java source	**C++ header**

`class Point`
`{`
`    public int x;`
`    public int y;`
 	
`    public Point add(Point rhs) {`
`        // ???`
`    }`
`}`
	`class Point : public virtual`
	`::java::lang::Object`
	`{`
	`public:`
	`    int x;`
	`    int y;`
	`    Point();`
	`    Point *add(Point *rhs);`
	`};`

`javap` output based on the class file	**C++ stub**

`public class Point`
`{`
`    public int x;`
`    public int y;`
`    public Point();`
`    public Point add(Point);`
`}`
	`Point() : x(), y() { }`
	
	`Point *Point::add(Point *rhs)`
	`{`
	`    return nullptr;`
	`}`

Table 3: Generating a stub from a dependency without source.

3.3.2 Convert dependencies

At the other end of the spectrum lies the second alternative. With a Java converter in hand, it becomes possible to convert an existing implementation of the Java Platform to C++ and use the converted code.

The obvious advantage is guaranteed compatibility as the exact same implementation of the dependency is used. This approach can also be extended to dependencies on libraries other than the platform library, for which the source code is available.

The approach however does not come for free. For example, a single dependency on the `String` class in OpenJDK pulls in ca. 1000 other classes as dependencies of dependencies are pulled in recursively making a small application increase its binary size and load times significantly.

Also, certain parts of the JDK are implemented as native methods that depend on a particular Java Virtual Machine being present, and such methods must still be implemented manually. In OpenJDK, the ca. 1000 classes that `String` depends on contain ca. 480 such native methods, but depending on the application being translated, only a handful of those are likely to be called.

This approach is most useful in cases where the converted code has many external dependencies, specially such that have no clear replacement in C++. One example would be an application making heavy use of complicated internet standards such SOAP and its companion protocols, where reference implementations exist for Java but not necessarily for C++.

3.3.3 Mixed approach

The third way lies in the middle ground. Of the ca. 100 classes that SWT depends on, most come from the `java.lang` and `java.util` packages that cover core language features and collections. The classes of these two packages are used by most Java applications, so these are the classes that carry the largest benefit of a native implementation. For example, further examination of SWT and H2 shows that 90 of the dependent classes are shared between the two libraries. The strategy thus becomes to concentrate on the core classes such as `Object`, `String` and `ArrayList`, implementing those natively while taking the rest from an existing platform implementation.

A study on API usage by Lämmel, Pek and Starek [21] that found that out of 1476 projects, 1374 used the Java collection classes compared to Comm.Logging used only by 151 projects.

By also comparing the number of distinct methods called with the number of calls to this method for each API category, an initial prioritization for the native implementation effort can be obtained.

For example, in the above libraries, 392 639 calls were made to 406 distinct methods of the collection classes giving a ratio of ca 1000 calls per method, compared to the usage of JUnit where 71 481 calls were made to 1011 methods, averaging ca 70 calls per method. Such numbers suggest that a conversion of the collection classes would have larger impact for the same development effort, assuming comparable average effort per method required.

This approach is best used when the natively implemented code can be reused across multiple projects, maximizing the benefit of a manual conversion.

3.4 Java Native Interface

The Java Native Interface (JNI) provides an application programming interface (API) that applications can use to allow Java code interface with native code and vice versa. The use of JNI is discouraged as it breaks platform independence, one of the main goals of the Java environment.

In the OpenJDK, native calls are used for several reasons:

- Implementing classes that need to make use of operating system services, as seen in the file I/O classes.
- Interaction with the Java Virtual Machine (JVM) - the `wait` and `notify` methods on the `Object` class are `native` as they require interacting with locks that are taken by language primitives and implemented in the JVM.
- Circumvent limitations of the Java language - for example, `System.out` is a final field that represents the standard console output stream and may per its final modifier not be assigned after the static initializers have been run. To allow users to replace it with another stream and maintain binary compatibility with older Java versions, a native method `setOut` is provided that circumvents the protection mandated by the `final` keyword.
- Enable hardware or platform specific optimizations, such as efficient interlocked memory access that is used to implement for example atomic counters.

Typically, when using JNI to interface with existing code, bridge code is written that interacts with Java using a reflection-like API where methods and fields are looked up by name using string literals. Apart from being cumbersome, it is also not very performant, thus it makes little sense to reuse it directly in a native translation as methods and fields are directly accessible in the translated code without the use of string literals.

The use of native code is discouraged in Java as one of its objectives is to maintain platform independence which is not possible with native code. As a consequence, JNI is not widely used thus rewriting JNI calls manually is likely to require little effort.

3.5 Execution and threads

Program execution in Java begins with the virtual machine initializing itself and the core Java classes needed for loading Java byte code. Then, similar to C++, a `main` method is executed in the class that the user specifies. For each `main` method encountered in the original code, we can generate a special stub file that runs a runtime initialization routine and translates command line arguments to a Java `String` array.

Thread support in Java is split between the runtime and the language itself. The language provides primitives for synchronization and guarantees about the execution environment while the actual management of threads is delegated to the runtime, which consists of a virtual machine and a platform implementation.

Synchronization primitives in Java are an implementation of the monitor model [22]. Methods and blocks may be declared as synchronized meaning that a mutually exclusive lock is taken for the duration of the block. Inside a synchronized block, there is support to temporarily release the lock while waiting for notification from another thread, but this support is implemented as part of the `Object` class, not as a language feature.

Conceptually, the `synchronized` keyword is similar to C++ standard library's `std::unique_lock` class template when used with an instance of the `std::recursive_mutex` class, while the notification support in `Object` can be implemented using a `std::condition_variable`.

It is not possible to take this approach directly however as in C++, an instance of a separate `std::recursive_mutex` class is required whereas in Java, all `Object` instances can serve as arguments to the synchronized statement. Since most object instances are not used for locking, it would be wasteful to include a mutex instance in every object. Instead, when translating synchronized statements, calls to unimplemented lock and unlock functions are inserted where needed, and an appropriate implementation can then be chosen based on locking usage patterns in the application or library. This is similar to how a Java compiler outputs lock and unlock bytecode instructions as appropriate.

3.6 Memory and other system resources

In contrast to C++ where memory resources must be explicitly released, Java has automatic memory management in the form of garbage collection. It is also possible to write special code that will be executed when an instance is about to be deallocated in the form of a finalizer. The language provides no means to deterministically release memory - in fact, it is not guaranteed that memory will be released at all, also meaning that finalizers will not necessarily be run prior to program termination.

Thus a correct implementation never has to release heap allocated memory, and we leave it to a future study to examine solutions where memory is reclaimed. Possible routes forward would be to use an existing collector such as the Boehm-Demers-Weiser conservative garbage collector [23] or implement reference counting with cycle detection, as is used by the reference implementation of Python. We also note that the Boehm-Demers-Weiser collector supports finalizers which are necessary to provide emulation of Java garbage collection.

Heap allocation and thus garbage collection can be avoided altogether in certain cases. Through the use of interprocedural escape analysis, Choi et al.[24] show how in a particular set of Java benchmarks, a median of 20% of all heap memory allocations can be avoided. If the lifetime of a reference type instance can be proven to be limited to a particular method, it may safely be stack allocated and automatically deallocated as the method ends, lessening the pressure on the garbage collector, and in the case of our C++ code, simplifying the generated code. Similar analysis for the locking mechanisms of the benchmark code shows that a median of 51% of all locking can be avoided, as the locks are being taken where it can be proven that only one thread has access to the locked resource.

The lack of explicit memory management has a profound effect on idiomatic use of the language, specially when interacting with other system resources such as files, network connections and user interface elements.

In C++, it is common practice to release such resources as the lifetime of an object ends, by placing cleanup code in the destructor. The ownership of a system resource thus follows the lifetime of the instance that acquired the resource, a design principle known as “resource acquisition is initialization”, or RAII [25]. Table 4 shows a typical C++ class that owns a database connection that is released when the instance of the class goes out of scope.

```
class database
{
    public:
        database(connection *c) : c(c) { c->connect()); }
        ~database() { c->close(); }

     // …

     private:
          connection *c;
};

void f(connection *c)
{
    database db(c);
// use db object
// ...
// Here, connection is closed by the destructor
}
```

Table 4: C++ resource management

In Java, when a resource has been acquired, it must explicitly be released, just as memory has to be released in C++. There is no natural place for such cleanup code in Java, thus it is often spread out in an application. One common technique is to place it in finally blocks in every place where the resource is used, to ensure cleanup even in the face of abrupt termination, as shown in Table 5. Using the database class as example, there is however no way for the translator to know that close should be called to do cleanup based on the local information it has when processing the class.

Also, if the translator was able to determine that the close function in fact performs destruction akin to that of the C++ destructor, it still could not simply call it from the C++ destructor without introducing unsafe code that either terminates in the face of an exception or silently swallows it.

```
class Database
{
     public Database(Connection c) { this.c = c; c.connect(); }
     public void close() { c.close(); }
     private Connection c;
}

...

void f(Connection c)
{
     Database db = null;
     try {
         db = new Database(c);
    } finally {
        // Explicitly have to close database
        if(db != null) db.close();
    }
}

...
```

Table 5: Java resource management

Our translation follows Java semantics by simulating finally using C++ constructs, and makes no attempt at providing destructors which would more naturally fit with C++ idioms. This approach follows naturally from the decision not to manage memory explicitly, but to rely on a library provided garbage collector such as Boehm-Demers-Weiser.

Chapter 4. Language migration

In this section, the details of language migration from Java to C++ will be covered. The chapter is organized using the Java Language Specification as a model, and covers the parts relevant to translation that are not trivially carried over to C++. Throughout, excerpts from the Java Language Specification appear in *italics*.

4.1 Base assumptions

It is assumed that we have the means to create an accurate representation of the Java source code in the form of an abstract syntax tree, where types, fields and method calls have been resolved. While an interesting problem, parsing the Java source code in accordance with the full specification is not the focus of this work.

The output of a translator must obviously be valid C++ code, and at the lowest level that means that it must be encoded in way that conforms to the rules of C++ parsing. Digraphs and trigraphs need to be escaped, Unicode characters escaped and so forth. Just like we assume that we are able to parse Java code we will assume that we are able to output syntactically valid C++ code.

As Terekhov and Verhoef describe [3], each language construct of the source language can either have a native counterpart in the target language, be easily simulated or remain beyond the grasp of a simple translation. In some cases, compound constructs in the source language may also have a native counterpart in the target language - such conversions improve the quality of the translation but are not necessary for correctness assuming that trivial translations exist.

4.2 Lexical structure

The grammar of a language helps decomposing valid source code into logical units suitable for analysis. The grammar of both Java and C++ is defined in terms of tokens, valid sequences of characters, that make up a valid program. Tokens come in the form of identifiers, keywords, literals, operators and separators. Whitespace in both languages is largely ignored but significant in that it separates other tokens.

Both Java and C++ programs are interpreted using the Unicode character set. Regardless of the encoding of the source file and use of Unicode escape sequences and other representation tricks, the internal representation of names and identifiers in the translator is assumed to follow the Unicode standard.

Comments in Java and C++ are equal in their definitions and can thus be copied directly when translating. In both Java and C++ they are ignored by the compiler and thus do not affect the correct execution of the program, but are highly relevant for a complete translation.

Identifiers in Java are similar in spirit to those of C++. Both languages essentially allow any sequence of letters and numbers to be used as an identifier, excepting those that start with a number and those that form a reserved keyword in the language. '$' is allowed as an identifier in Java, and although it is not so in C++, many compilers accept it anyway. In C++, identifiers starting with two underscore characters, one underscore and a capital letter or one underscore and any letter when in the global namespace are reserved for the system. A translator will have to provide an encoding for those identifiers in Java that would be invalid in C++ due to keyword conflict or system use.

4.3 Code organization

The unit-of-work for a Java compiler is the compilation unit, typically stored in a single source file. The compilation unit defines the basic scope for name lookup, symbol visibility and access control. In similar fashion, C++ compilers operate on a translation unit that provides name lookup and symbol visibility scope.

The Java compiler can make use of class files produced in previous compilations when resolving references external to the current compilation unit and places no restrictions on the order in which declarations within a compilation unit appear.

In contrast, C++ compilers have no provision for using symbols from object files, the intermediate output of a C++ compiler. Instead, the declarations of functions, variables and classes must be repeated for each translation unit in source code form. As a matter of convenience, such repeated declarations are stored in header files which can be reused by multiple source files.

When resolving type references, the C++ compiler may need either a forward declaration that declares the name of the type only or a full declaration depending on the context of the resolution. It is therefore practical to split class definitions into three parts - forward declaration, declaration and definition, each residing in a separate file, repeating the process for each distinct type defined in the Java compilation unit. The C++ preprocessor will then, among other things, join the files back into a single translation unit before passing them on to the compiler.

4.3.1 Packages

To prevent name conflicts, Java programs are divided into packages. If the code is stored on a file system, the package name also dictates the location of the class file. Package names are hierarchical, but when referenced in code, the full name is always used.

We will translate packages to C++ namespaces, and when qualifying type names with a namespace, we will always use the full name and the global qualifier as shown in Table 6. This is similar to how package references are used in Java, and necessary as unqualified namespace lookup in C++ begins in the current namespace and works itself up the hierarchy. Without the global qualifier, a match deep in the hierarchy would have precedence over a root namespace with the same name.

Java	**C++**
`java.util.ArrayList`	`::java::util::ArrayList`

Table 6: Qualified class names

Fully qualifying namespaces leads to verbose type references, but at least for types in the current namespace, unqualified access may safely be used. For any other namespace, it is not possible to guarantee that the correct type will be chosen without global knowledge about the code, and thus a conservative approach is chosen.

In Java, fully qualifying type names can be avoided by using import statements, which brings one or more type into the current lookup scope. In C++, the using directive fills the same purpose, but unfortunately, precedence rules of lookup differ between Java's `import` and C++'s `using` leading us to taking the conservative approach of always fully qualifying names in other namespaces.

Peterson et al. suggest including the package name in the class name, so that `java.util.ArrayList` becomes `java_util_ArrayList`. This is worse even than our conservative approach as the package name always has to be spelled out, whereas using C++ namespaces allows us to avoid using the package name in some cases at least.

4.3.2 Names

Names are used to refer to the declared packages, types, methods, fields and variables in a program. In Java, names can either be qualified or simple. Simple names are looked up in the current name scope, and the context of the lookup is used to disambiguate between different entities with the same name. Thus it is allowed in Java to have types, variables and methods all with the same name, and lookup by simple name will continue to work.

In C++, there is no provision to disambiguate unqualified names according to the semantic context. Further, methods and fields are not allowed to have the same name.

Peterson et al. suggest that without global knowledge of all names, naming clashes can be solved either by prefixing each name type with a specific prefix, i e all methods are prefixed by ‘`m_`’, classes by ‘`c_`’ etc, or by changing the original Java code in the cases where local information is not enough [15].

However, by turning unqualified names into qualified names, it is possible to change the C++ name lookup scope and can thus disambiguate names with only local knowledge. The method declaration and recursive call in Table 7, where ‘`a`’ is both a type, method and argument name can be translated correctly by qualifying type names with namespaces and member access with ‘`this`’. We will still need to apply some sort of mangling to fields and methods with the same name within a single class, but that decision can be taken locally on a class-by-class basis.

Java method	**C++ method**
`a a(a a) { a(a); }`	`::a a(::a a) { this->a(a); }`

Table 7: Avoiding conflicts using qualified names

To solve the problem where a method hides a base class field or vice versa, casts need to be inserted when accessing the base class member. Suppose the base class of the above example had an ‘`a`’ field - by casting ‘`this`’ to the base class type, the field can still be accessed.

4.4 Type system

In Java, there are two kinds of types: primitive types and reference types. Primitive types are the numeric types such as `int` and `float` as well as `boolean`. Variables of primitive type follow value semantics - they hold their value directly and copy the value on assignment which is also how fundamental types work in C++.

Variables of reference type follow reference semantics. The variable holds a reference to an instance of the type, an object, somewhere else in memory. When a reference type variable is assigned, only the reference is copied - the object pointed to remains the same. In C++, the most convenient way to represent reference types is through pointers - they follow the same semantics as Java references and pointer type relations follow the relations of the type they point to just as as do Java references.

4.4.1 Primitive types

While the primitive types in Java are similar to the fundamental types of C++, the Java types are more strictly defined with respect to size and representation. Where Java requires integral types to be represented in 2's complement and have set sizes for each type, the corresponding C++ types have implementation-defined sizes and representation. Instead, C++ defines a special header, `cstdint`, that contains names of types that correspond to integral types with 2's complement representation and specific sizes, as seen in Table 8.

These names are optional - if a particular implementation does not support them, it will not be possible to convert a Java program in a meaningful way. Fortunately other representations than two's complement are rare, as are compilers not supporting the standard sizes for integers. Table 8 shows the C++ types names corresponding to the Java primitive types.

Java defines two floating point types, `float` and `double`, as 32 and 64-bit floating point numbers adhering to the IEEE 754 standard. C++ also has a `float` and a `double` type, but does not define their representation and size. Typically however, these two types however correspond to their Java counterparts and C++ offers compile time support to detect if that is the case through the `sizeof` operator and the `numeric_limits` class template.

Should a particular implementation lack 2's complement integral types or IEEE 754 floating point types, it may be possible to provide emulation using types specially crafted for the implementation, but we will assume that the compiler and the hardware platform does support them.

Java	**C++**
`boolean`	`bool`
`byte`	`int8_t`
`char`	`char16_t`
`double`	`double`
`float`	`float`
`int`	`int32_t`
`long`	`int64_t`
`short`	`int16_t`
`void`	`void`

Table 8: Primitive type mappings

4.4.2 Reference types

In Java, there are three kinds of reference types: classes, interfaces and arrays. Variables of reference type are pointers to an object that may be either of class or array type. Interfaces serve to define a contract - they contain no actual implementation code and may not be used to instantiate objects, thus the actual instance pointed to by a variable of interface type will never itself be of interface type. Classes may contain both declarations and definitions, but are limited to inherit from only one other class.

Variables of reference type have reference semantics - when the value of such a variable is copied to another variable, both share the same underlying instance.

In C++, we will represent classes with `class`:es and interfaces with `struct`:s. The distinction has no effect on actual machine code generation but serves as documentation - interfaces, whose members must all be public, align more closely with `struct`:s whose members are also public by default. Table 9 contains an overview of the concepts involved during type translation and how they affect the output.

Java	**C++**
class	class
interface	struct
enum	class
abstract	make constructors protected
final	make methods non-virtual or final
nested static class	class (no nesting)
inner class	class (no nesting), extra constructor parameter for instance
local class	class (non-local), extra constructor parameter for each closure
annotation declaration	struct
annotation use	ignore
generics	ignore (use erasure)
reference type variable	pointer variable

Table 9: Reference type translation overview

4.4.3 Boxing and unboxing

For each primitive type, Java defines a corresponding reference type that may be used to represent the value of the primitive types where reference types are expected, for example the collection classes.

The Java language allows implicit conversions between primitive types and their respective reference types - boxing and unboxing. A boxing conversion converts a primitive value to a reference type with the corresponding value, and vice versa for unboxing. Boxing conversions are guaranteed to always return a reference to the same instance for certain primitive values to maintain identity equality for the most commonly used values.

Had value semantics been used for reference types in the translated C++ code, implicit conversion operators and constructors could have provided a similar syntactic brevity for boxing and unboxing, but there is no way to specify such conversions for pointers. Instead, we translate boxing conversions to calls to the `valueOf` method of each reference type and `<type>Value` calls for unboxing conversions - these methods guarantee identity equality as required by the language.

4.4.4 Classes

Java allows classes to inherit from multiple interfaces but only one class. Interfaces in turn may inherit from other interfaces and there are no restrictions on inheriting multiply from the same interface in a class hierarchy. To avoid ambiguities and duplicates in the C++ class hierarchy, we will use virtual inheritance when translating interface inheritance. We note that it is not possible to avoid virtual inheritance for interfaces that are only inherited once in a particular hierarchy based on only local knowledge about the class being translated except for final classes - interface inheritance needs to be virtual in all classes that may be used as a base class.

In Java, all class and array types inherit implicitly from a common root class, `Object`. Interfaces may not inherit from a class, but throughout the Java language, when considering type, interfaces behave as if they did in fact have `Object` as base. Since we're simulating interfaces with an ordinary C++ `struct`, we will have it inherit from `Object` as well. Again, virtual inheritance is needed as `Object` may appear at several branches in a type hierarchy.

Classes in Java may be declared abstract or final. Abstract classes may not be directly instantiated and are thus allowed to contain unimplemented, or `abstract`, methods. In C++, there is no need to mark a class as abstract - the language allows classes to have unimplemented pure virtual methods as long as they are not instantiated. To mark that a class is not intended for instantiation, we make its non-private constructors protected which makes them inaccessible for direct instantiation.

Declaring a class to be final means that the language disallows further subclassing of that class. This constraint is possible to simulate using private constructors and special static factory methods in C++, but the syntactic burden of such a translation outweighs the benefit as it has no impact on runtime behaviour and requires an additional method for each constructor in the source class.

Methods in final classes are implicitly `final`, meaning that they can either be marked as `final` in C++ or simply not be declared as virtual, depending on whether they already override a base class or interface method or not.

4.4.5 Nested classes

Classes in Java may be nested in other classes or interface. There are two types of nested classes, static and non-static. Static nested classes are similar to ordinary top-level classes except that they gain access to private declarations in the enclosing type. Instances of static nested classes have access to static fields and methods of the enclosing type.

Non-static nested classes, or inner classes, implicitly gain a reference to an instance of the enclosing type when being instantiated, which allows them to also access instance methods and fields of the enclosing type.

The Java compiler handles inner classes by adding a hidden field of the enclosing type to the inner class and makes each constructor take an extra argument to initialize the hidden field.

When translating nested classes, we process them as we would an ordinary class, but do not nest them. In C++, the outer class remains an incomplete type in the declaration of the nested type disallowing return covariance and inheritance from the outer class, both permitted by Java.

For inner classes, we add a field that holds a pointer to the enclosing type and modify all constructors to take an extra parameter, just like a Java compiler. This parameter is then initialized with the value of the enclosing instance whenever an instance of the inner class is created with the new operator.

4.4.6 Local classes

Local classes are classes declared inside a method body. They are accessible only from the method in which they are created and as such, gain access to final local variables in that method. Local classes in non-static methods also gain access to the instance on which the method is being executed, just like inner classes. Local classes may also be created as part of an instance creation expression, in which case they are called anonymous classes. Such classes become subclasses of the type specified in the new expression and remain unnamed.

When translating local classes, for each variable from the enclosing method accessed in the local class an extra field and an extra constructor parameter is added. During instantiation, the variables and instance, if any, are passed as constructor arguments, copying the value of the variable at instantiation time.

4.4.7 Enum types

Enum types in Java are a special kind of class type that may only be instantiated during the declaration of an enum constant. Enum declarations are split into two parts - the constants and an optional body. In the body, fields, constructors and methods, possibly abstract, are defined as usual. Enum constants thus become instances of anonymous types that inherit from the enum type and must implement any abstract methods.

The Java Language Specification suggest looking at enum types as classes derived from the class `Enum`, with the constants being represented by static fields that are references to the enum type and a few extra methods providing support.

C++ `enum` types are not at all similar to the `enum` construct in Java. Instead, we will translate them as the Java language specification suggests - ordinary classes that may not be instantiated, and whose only instances are the ones available through the constant fields.

This emulation falls short in one area however - in Java, the constants of an enum may be used for the case labels of a switch statement. In our C++ emulation, the constants are represented by static fields which, due to not being `constexpr`, may not be used for `case` labels. Instead, `switch` statements need to be rewritten as a series of `if` statements.

4.4.8 Interfaces

Interfaces in Java serve to define a contract for a set of operations without providing an implementation. Interfaces members are implicitly public, and limited to types, constants and abstract methods. Multiple inheritance is allowed among interfaces, but they may never inherit from a class, including `Object`. However, since there are no instances of types that do not ultimately inherit from `Object`, the specification contains special provisions to make interface types behave as though they actually did inherit from `Object`. An interface that has no superinterface will implicitly have all members of `Object` declared, and when determining type relations for implicit conversion, assignment and other relevant areas, `Object` is considered a supertype of any interface without superinterfaces.

There is no direct equivalent of an interface in C++ but `class`:es and `struct`:s support a superset of the features of an interface. To carry the intent of implic public access to all members from Java to C++, we will use `struct` instead of `class` when translating interfaces. There is no way to express the supertype relation with `Object` other than through inheritance in C++, and such inheritance must then necessarily be virtual. As Peterson et al. note, this incurs a performance penalty on the translated code as dynamic casting becomes necessary for many cases where it could have been avoided. They further suggest that it is possible not to inherit from `Object` and use explicit casts whenever a variable of interface type needs to behave as an `Object` instance, but with return type covariance added to Java 1.5, such a solution no longer covers all cases.

4.4.9 Arrays

Arrays in Java are used to provied storage for multiple variables using indexed access. Array types inherit from `Object`, as well as `Cloneable` and `Serializable`, and are based on a component type, that itself may be an array. The length of an array is available dynamically after the array has been instantiated through the `length` field.

The type relations of arrays follow the type relations of their component type, for example an array of `String`:s will be assignable to a variable of `Serializable` array type, as a `String` is assignable to a `Serializable` variable.

To implement array type support in C++, a special class can be used that provides storage and the required members of all Java array types.

However, due to the relation between array types, it is not possible to provide a single generic class implementing array support for all array types. Instead, a separate class must be generated for each encountered array type. Arrays of derived types must inherit from the array type of the base of the derived component type to allow variable assignment, covariance and other constructs to carry over naturally to C++, in addition to inheriting from `Object`, `Comparable` and `Serializable`.

4.4.10 Annotations

Annotation types are special interfaces that are used to provide metadata about types and their members to compilers, source analysis tools and programs making use of reflection. We will translate annotation type declarations as we translate interfaces, but ignore them otherwise.

One potential use for annotations would be to provide additional information about types to the source-to-source translator itself, allowing the translator to generate more appropriate code in certain situations. For example, a `@NotNull` annotation on a field could make the translator assume that the field never carries a `null` value, and therefore allow it to skip the null check.

4.4.11 Generics and erasure

Generics in Java are used to provide additional information about types that the compiler uses to guarantee type safety, or the absence of runtime casting errors. It also allows the compiler to safely insert implicit casts where manual casting would have been needed, reducing the syntactic burden of the language.

To take advantage of generics, types and methods are decorated with type parameters. These type parameters are then reused in the type or method declaration providing type guarantees to the compiler. When a generic type or method is used, the user must supply actual types for each type parameter which allows the compiler to verify the type correctness of expressions that use the type parameters.

Once the compiler has verified type correctness, generic types and methods undergo a process called erasure. Type parameters used in type and method declarations are replaced by actual types according to rules set out in the specification, and implicit casts are inserted where needed to maintain correctness - generic type information is erased.

While generics syntactically look similar to C++ templates, and provide some of the same convenience to users, the differences are notable as the following examples show:

- In C++, classes are either templates or not, whereas Java allows generic classes to be used without the extra type parameters ("raw types")
- In C++, static members of template classes have access to the type parameters - in Java they do not
- In C++, static members are distinct in different instantiations of a template class - in Java only a single copy exists
- Instantiated class templates in C++ actually are distinct types as casting and the typeid operator shows - the class literal of two parameterized types gives a reference to the same underlying erased type

During translation, we will use the erased definition of all types which is enough to maintain runtime correctness. Even though we lose some of the syntactic advantage generics offer, there is no native generics support in C++, and simulation using templates would prove complicated, if at all possible. Unless otherwise noted, we will assume that types have been erased before further processing.

4.4.12 Class Initialization

Before a class in Java may be used, it needs to be initialized. During class initialization, static fields get their values and static initialization blocks are run.

C++ also has the concept of static initialization, but in C++, the order in which static initialization happens is not defined, except that it happens before the main program starts. There are two problems with this approach.

The first is that dependencies between multiple initializers are not resolved deterministically leading to undefined behaviour.

The second is that a large code base might consist of thousands of classes. If all their initializers were to be run at startup, load times would increase notably even for cases where the majority of the classes are not actually used.

Java solves the problem by defining exactly when a class is initialized and in what order initialization happens, delaying that initialization as far as is deemed possible ([2], §12.4.1):

Before a class is initialized, its direct superclass must be initialized, but interfaces implemented by the class need not be initialized. Similarly, the superinterfaces of an interface need not be initialized before the interface is initialized.

A class or interface type T will be initialized immediately before the first occurrence of any one of the following:

- *T is a class and an instance of T is created.*
- *T is a class and a static method declared by T is invoked.*
- *A static field declared by T is assigned.*
- *A static field declared by T is used and the field is not a constant variable (§4.12.4).*
- *T is a top-level class, and an assert statement (§14.10) lexically nested within T is executed.*

Class initialization consists of several steps. First, fields are assigned default values in accordance with their type. Then, fields initialized by constant expressions are initialized and finally field initializers and initialization blocks are executed in textual order.

The specification allows the possibility to construct a Java program that observes field default values by referencing not yet initialized fields from static initialization blocks.

When translating, we coalesce all static initialization code to a single static method. To our advantage, the default values given to static variables during static initialization in C++ coincide with those of Java, so we can proceed directly to initializing those constant values that could not be translated as C++ `constexpr` expressions. After those follow the field initializers and initialization blocks.

We can then insert calls to the initialization code for each class in each of the following places:

- The beginning of every static initializer of any subclass
- The beginning of every constructor
- The beginning of every static method
- When reading or writing non-constant fields by replacing field access with a special method that calls the initializer before returning a reference to the field

Peterson et al. suggest calling the initialization of all dependent classes on startup [15], arguing that it would be prohibitive from a performance perspective to initialize classes lazily, but a downside of such initialization is that startup time grows with the number of classes present and that classes will be initialized regardless if they will actually be used.

4.4.13 Instance Initialization

Like C++, class instances in Java are initialized using constructors - special methods that are called whenever a new instance is created. However, before code contained in constructors is executed, fields are assigned default values, then field initializers and initialization blocks are run ([2], §12.5):

Whenever a new class instance is created, memory space is allocated for it [...] all the instance variables in the new object, including those declared in superclasses, are initialized to their default values (§4.12.5).

...

1. *[...]*
2. *If this constructor begins with an explicit constructor invocation of another constructor in the same class (using this), then evaluate the arguments and process that constructor invocation recursively using these same five steps [and then] continue with step 5.*
3. *This constructor does not begin with an explicit constructor invocation of another constructor in the same class (using this). If this constructor is for a class other than Object, then this constructor will begin with an explicit or implicit invocation of a superclass constructor (using super). Evaluate the arguments and process that superclass constructor invocation recursively using these same five steps [and then] continue with step 4.*
4. *Execute the instance initializers and instance variable initializers for this class, assigning the values of instance variable initializers to the corresponding instance variables, in the left-to-right order in which they appear textually in the source code for the class [and then] continue with step 5.*
5. *Execute the rest of the body of this constructor. [...]*

When translating, we thus need to handle default values, make explicit or implicit calls to super constructors or chained calls to other constructors and finally run the code of the constructor body. Additionally, we need to take static initialization into account, which must be run before any instance initializers.

One final difference needs to be addressed. In Java, if a constructor makes a call to a virtual method implemented in a subtype of the class currently being constructed, the implementation in the subtype will be called with the subtype partially initialized. In C++, virtual calls in constructors are resolved as if an instance of the type currently being constructed was being used - the final virtual table is not available until the object has been fully constructed.

We translate constructors by dividing the constructor into two parts. The first part, the C++ constructor, initializes fields to their default values then calls the static initializer to make sure the class has been initialized. The second part, a separate `init` method, is implemented as follows:

1. If the Java constructor begins with an explicit `this` constructor call, evaluate the arguments can call the corresponding `init` method in this class then continue with step 4.
2. Except for `Object`, generate a call to the `init` method from the super class, either from an explicit super call in the Java constructor or from the implicit call that Java mandates.
3. Run instance initializer, if any
4. Run constructor body

When instantiating an object with the new operator, we thus need to call the `init` method on the newly created object explicitly to complete the two-phase construction. This contrasts with idiomatic use of C++ and should therefore be considered a significant burden, but is necessary to implement correct behavior in the presence of virtual calls in the constructor. It is also not possible to require two-phase construction only for those classes that actually make use of virtual calls, as there is no way to tell from a Java constructor signature if virtual calls will be made.

However, while it is possible to construct a program that makes use of virtual calls during constructors, such programs are rare. If a virtual method is implemented in a subclass of the class currently being constructed, that subtype will not yet be fully initialized meaning its fields will still have their default values and instance initializers will not yet have been run.

Thus, it may be beneficial to sacrifice correctness for convenience and run the second phase of construction directly from the C++ constructor. Since it is possible to detect virtual calls during translation, a translator can issue a warning or deliberately generate code that will fail C++ compilation to notify the library user of the semantic deviation.

Another option, often used when two-phase construction is necessary in C++, is to make constructors private and supply a static factory method for each constructor that performs the `new` and `init` calls. Such a solution has the benefit that a reader of the converted class will have no opportunity to only partially initialize the object, but enforces an unusual syntax for constructing instances.

4.5 Exceptions

Exceptions in both Java and C++ are used to abruptly break program execution flow and transfer control to an exception handling routine. Exceptions in Java come in two flavors - checked and unchecked. Checked exceptions thrown in a method body or any of the methods called by the body are required to be declared after the method signature while unchecked exceptions require no specification. Further, all exceptions must inherit from a common base, `Throwable`.

C++ allows any type to be used in a `throw` expression, allowing a straightforward conversion of throwing and catching exceptions. There is no concept of checked exceptions in C++, but being a compile-time only feature of Java, it can safely be ignored.

Additionally to the explicit throw expression some expressions in Java may cause an exception being thrown implicitly. Examples include trying to dereference a reference variable holding a null reference and accessing array elements outside the range of the array.

For `null` reference checking, a naive approach would be to insert a conditional statement whenever a variable is dereferenced, but such an approach would be syntactically cumbersome and difficult to implement for compound expressions.

Instead, a helper function shown in Table 10 that implements the appropriate check and throw exceptions if needed can be used, minimizing the syntactic overhead at the check site. A similar function can be used to implement divide-by-zero checking for the division and remainder operators.

```
template<typename T>
static T* npc(T* t)
{
	if(!t) throw new ::java::lang::NullPointerException();
	return t;
}
```

Table 10: Null pointer check function

4.6 Methods

In both Java and C++, methods are used to declare code related to a particular class. Typically, methods hold the majority of the code that will be executed for any given Java program and thus form the center point for most Java applications.

A method declaration consists of a signature that declares its name, parameters and return type, an optional exception specifier and an optional body containing the executable code.

Translating a method signature to C++ is mostly straightforward - return types, names and parameter lists follow the same pattern. Several modifiers may be applied to the method signature. In both C++ and Java, methods may be `static`, `abstract` or `final` though the syntax varies slightly. One important syntactic difference is that in Java, methods are implicitly virtual - this needs to be made explicit in C++ through the `virtual` keyword. The remaining modifiers, `synchronized`, `native` and `strictfp` affect the implementation, or body, of the method, but not the translation of the signature.

Exception specifiers in Java document the checked exceptions a method may throw. The exception specifiers are verified at compile time to be consistent across the type hierarchy and with the exceptions potentially being thrown by the method body. They are however not part of the method signature and thus do not participate in overload and override resolution, and have no impact on the runtime behavior.

There is no equivalent feature in C++, and while older versions of C++ allowed runtime checked throw clauses on the method signature, flaws in its specification led to the feature being deprecated. It is however safe and reasonable to simply ignore exception specifiers during translation, as they checked at compile-time only in Java.

4.6.1 Overriding

The concept of overriding the method of a base class in a subclass is one of the cornerstones of the object oriented approach of Java. In Java, unless specifically disabled through a keyword, methods are virtual and can have overrides.

Java classes and interfaces implicitly inherit all the methods of their direct superclass and direct superinterfaces that are accessible. Whether or not a method in a class overrides an inherited method is based on the concept of subsignatures ([2], §8.4.2):

Two methods have the same signature if they have the same name and argument types.

Two method or constructor declarations M and N have the same argument types if all of the following conditions hold:

- *They have the same number of formal parameters (possibly zero)*
- *They have the same number of type parameters (possibly zero)*

- *Let <A1,...,An> be the formal type parameters of M and let <B1,...,Bn> be the formal type parameters of N. After renaming each occurrence of a Bi in N's type to Ai the bounds of corresponding type variables and the argument types of M and N are the same.*

The signature of a method m1 is a subsignature of the signature of a method m2 if either:

- *m2 has the same signature as m1, or*
- *the signature of m1 is the same as the erasure of the signature of m2.*

[...]

A method declaration d1 with return type R1 is return-type-substitutable for another method d2 with return type R2, if and only if the following conditions hold:

- *If R1 is a primitive type, then R2 is identical to R1.*
- *If R1 is a reference type then:*
  - *R1 is either a subtype of R2 or R1 can be converted to a subtype of R2 by unchecked conversion (§5.1.9), or*
  - *R1 = | R2 |.*
- *If R1 is void then R2 is void.*

[...]

An instance method m1 declared in a class C overrides another instance method, m2, declared in class A iff all of the following are true:

1. *C is a subclass of A.*
2. *The signature of m1 is a subsignature (§8.4.2) of the signature of m2.*
3. *[...]*

[...]

If a method declaration d1 with return type R1 overrides or hides the declaration of another method d2 with return type R2, then d1 must be return-type substitutable for d2, or a compile-time error occurs. [...].

Subsignatures are thus defined in terms of signature and erasure meaning that in the presence of generics, a method may actually override base class methods with two different signatures. Restrictions are placed on return type, but in general, it is allowed to refine the return type of a method with a more specific type.

In C++, as in Java, methods are overridden based on name and parameter types, with compatible restrictions on the return type.

However, since there is no concept of erasure and subsignatures in C++, method parameters must match exactly. Therefore, for every method whose signature matches the subsignature of another method, but not its signature, a bridge method must be added to the translated source code that implements the erased signature of the base method and forwards the call to the actual implementation with appropriate casts, as shown by Table 11.

Java	**C++**

`public interface I<T>`
`{`
`   // Erasure: void m(Object t)!`
`   void m(T t);`
`}`
 	
`public interface J`
`{`
`   void m(A t);`
`}`
 	
`class A implements I<A>, J`
`{`
`   // Overrides m(Object t)`
`   // in both I and J!`
`   void m(A a) { /* … */ }`
`}`
	`// Forward declaration needed`
	`class A;`
	
	`struct I`
	`  : virtual Object`
	`{`
	`   // Erased signature`
	`   void m(Object *t);`
	`};`
	
	`struct J`
	`  : virtual Object`
	`{`
	`   void m(A *t);`
	`};`
	
	`class A`
	`  : public virtual Object`
	`  , public virtual I`
	`  , public virtual J`
	`{`
	`    // Original method`
	`    void m(A *a);`
	
	`    // Erasure bridge method`
	`    void m(Object *a)`
	`    {`
	`        m(java_cast<A*>(a));`
	`    }`
	`};`

Table 11: Bridging methods for erased types

When looking up an implementation of an interface method, all the classes of a hierarchy are searched. When a class implements an interface, the implementation may therefore be taken from a base class, even if there is no override equivalent method in the subclass itself. This is different from C++ where an implementation must exist in the subclass for each pure virtual method. For each method missing from the subclass, a bridge method must be generated that calls the base class implementation, as demonstrated by Table 12.

Java	**C++**
`interface I`	`struct I`
`{`	`  : virtual Object`
`    void m();`	`{`
`}`	`    virtual void m();`
	`};`
`class A`	
`{`	`class A`
`    void m() { }`	`  : public virtual Object`
`}`	`{`
	`    void m() { }`
`class B extends A implements I`	`};`
`{`	
`   // no need for m()`	`class B`
`}`	`  : public A`
	`  , public virtual I`
	`{`
	`   void m() { A::m(); }`
	`}`

Table 12: Bridging interface methods implemented in base classes

4.6.2 Hiding

In C++, if there is a method with the same name but different signature as in a base class, the base class method will be hidden. Another way to look at this is that once a method with a particular name has been found, methods with the same name from base classes are no longer considered for resolving overloading. To use such base class methods, they must be brought into the namespace of the current class with a `using` directive, or casts must be introduced at the call site to ensure that the correct type is searched.

When translating such cases, using statements can be added to the generated code as shown in Table 13. A limitation of this approach is that should there be a private method with the same name in the base class, the generated code will no longer be valid and the second approach, a cast at the call site, must be used.

Java	**C++**
`class A`	`class A`
`{`	`  : public virtual Object`
`    void m() { }`	`{`
`}`	`    void m();`
	`}`
`class B extends A`	
`{`	`class B`
`    void m(int x) { }`	`  : public virtual A`
`}`	`{`
	`   void m(int x);`
`class C`	`   using A::m;`
`{`	`}`
`   void m(B b) {`	
`       // Both calls work!`	`class C`
`        b.m();`	`{`
`        b.m(5);`	`    void m(B *b)`
`   }`	`    {`
`}`	`        // Fails without using`
	`        b->m();`
	`        // Alternative syntax`
	`        // without using`
	`        static_cast<A*>(b)->m();`
	`        b->m(5);`
	`    }`
	`}`

Table 13: Overcoming C++ method hiding

4.7 Blocks and statements

Many of the core statements of both Java and C++ have equivalent definitions. In both languages, blocks are used to define scope and the basic looping and branching statements are the same. As Peterson et al. note, it is the similarity between these statements that makes translation between the languages an attractive option [15]. There are however important differences that will be covered in detail.

4.7.1 Labels

Labels in Java are used to identify the target statement for a `break` or `continue` statement.

When used as a target for `break` statements, execution continues after the labeled statement. We replace such `break` statements with `goto` and place the label after the statement instead of before.

Labels together with `continue` statements are used to begin a new iteration of the labeled loop, regardless of its nesting in other loops. Only labels on loop statements may be used with the `continue` statement.

A naive translation would be to place a label at the end of the loop block and jump there using a `goto` statement, in order not to miss the loop invariant check. Such a solution however would potentially jump over variable initializations for variables that are still live at the end of the loop block which is not allowed in C++.

Instead, we must choose a different strategy:

- For every label declared on a looping construct, define a uniquely named `bool` variable and set it to `false`
- After every nested loop, if the variable is `true`, continue execution as normal
- If the loop is directly nested in the loop that the variable belongs to, set the variable to `false` and issue a `continue` statement which will start a new iteration of the outermost loop
- If the block is nested more deeply, issue a `break` statement which will break the execution of the current loop and jump out to the next nesting level where a new check on the same variable will happen eventually leading to the point above

This strategy resembles the normal way of breaking or continuing nested loops in C++, thus this transformation, although more verbose than its Java counterpart, leads to idiomatic C++.

4.7.2 Assertions

The `assert` statement in Java allows developers to verify internal invariants, with the option to turn the checking off at runtime. C++, being statically compiled, does not have an equivalent statement which can be controlled at runtime.

Assertions are enabled on a class-by-class basis in Java, which means that in C++, a bool would have to be kept for every class and checked at runtime before executing the assertion code. This is not idiomatic to C++ - instead we translate the runtime enabled per-class assertions to compile-time globally enabled assertions that are prevalent in C++.

Assertions become a useful tool for verifying that the conversion has introduced no incompatibilities with the original code. However, since assertions are meant not to impact runtime behaviour in the absence of bugs, they can also safely be removed completely.

4.7.3 The switch statement

`switch` statements in Java and C++ are mostly equivalent. In both languages, the `switch` expression is evaluated once and case labels determine what code gets executed. Without a `break` statement, case labels fall through, and a `default` label gathers up cases not covered by case labels.

One subtle difference is how variable declarations are handled. Although the scoping of variables is the same in both languages, in Java, it is valid for a case label to jump over the initialization of a variable in a previous case, if the variable is no longer used or if it is initialized anew in the new case. Just as with `goto` in C++, variable initializations in scope may not be jumped over, so we need to move variable declarations outside the switch statement.

Another difference is enumeration support. In Java, `enum` constants may be used as case expressions, but due to our translation of `enum`:s into classes, we also have to rewrite `switch` statements to the equivalent `if` statements. Care must be taken to only evaluate the switch expression once and store the result in a temporary variable that is then used for the successive `if` statements.

4.7.4 The for statement

`for` statements in Java come in two styles, basic and extended. The basic `for` statement is equivalent to that of C++, while the extended `for` provides syntactic sugar for iterating over collections and arrays.

C++ has a range based `for` statement similar to the extended for of Java, but it is not possible to trivially translate between the two. C++ range based `for` assumes that a `begin` and `end` function will return iterators that represent the beginning and end of a collection, while in Java, there is no concept of an end iterator that can be used for comparison with the current position - instead there is only one iterator that itself knows if it has reached the end.

Therefore, instead of emulating C++ iterators using the interface of Java iterators, we will simply transform the extended for loop into an ordinary for loop and use that as base for our translation. The Java Language Specification shows how such a transformation is done (§14.14.2):

If the type of Expression is a subtype of Iterable, then let I be the type of the expression Expression.iterator(). The enhanced for statement is equivalent to a basic for statement of the form:

```
for (I #i = Expression.iterator(); #i.hasNext(); ) {
      VariableModifiersopt Type Identifier = #i.next();
  Statement
}
```

[...]

Otherwise, the Expression necessarily has an array type, T[]. Let L1 ... Lm be the (possibly empty) sequence of labels immediately preceding the enhanced for statement. Then the meaning of the enhanced for statement is given by the following basic for statement:

```
T[] a = Expression;
L1: L2: ... Lm:
for (int i = 0; i < a.length; i++) {
      VariableModifiersopt Type Identifier = a[i];
      Statement
}
```

4.7.5 The synchronized statement

The synchronized statement in Java is used to implement language level support of mutual exclusion locking. Synchronization statements consist of an expression yielding an `Object` reference representing the resource needing protection and a block of statements to be executed while the lock is held. Regardless of how the synchronized block is terminated, the lock is automatically released.

To translate synchronized statements, and indeed synchronized methods, a helper class that performs a lock in its constructor and unlock in the destructor will be used. A local variable scoped with the synchronized block is added, passing a reference to `Object` retrieved from the expression in the synchronized statement. C++ scoping rules ensure that the lock will be taken when the variable is initialized and released at block exit. An alternative would have been to make explicit lock and release call, but in the presence of exceptions, implicit and explicit, such a solution quickly becomes unwieldy.

We leave the lock and release functions unimplemented, noting that their implementation also must be compatible with the wait and notify functions present on each `Object`. As noted previously, the standard library classes `std::condition_variable` and `std::recursive_mutex` can be used to provide an implementation.

4.7.6 The try statement

`try` statements in Java are used to enable exception handling for a particular block of code. `try` statements begin with a `try` block, followed by `catch` and `finally` blocks. `try` and `catch` are equivalent to those of C++ - exceptions originating from the `try` block are caught by the leftmost `catch` block with a compatible catch type and execution continues normally as long as all exceptions have been caught.

C++ however does not support `finally` blocks. In Java, `finally` blocks are used to clean up resources used in the `try` statement as there is no other deterministic way to ensure that a resource is released once it is no longer needed in the presence of garbage collection. In idiomatic C++, resources are cleaned up as part of object destruction which is deterministic and therefore, there is generally no need for a finally block.

It would be tempting to replace the finally block with a class that runs the code in the finally block upon destruction and place a variable of that class in a block that encompasses the rest of the `try` statement. Such a solution would guarantee that the code in the finally block is after the try statement has completed, even in the presence of exceptions and return statements.

However, if the code of a finally block ends abruptly because of an exception or return statement, that termination takes precedence over the original reason for terminating the try statement. An exception in a destructor in C++ terminates the program and a return statement would exit only the destructor, not the method encompassing the try statement.

Instead, we have to choose a more verbose solution where the `finally` code is repeated at least twice. The original `try` statement is surrounded with a `try` statement as shown in Table 14.

Then, in the catch-all handler inside the if block, and at every point where the original `try` block is left, the `finally` code is repeated. As Peterson et al. note, there exists a surprising number of ways to leave a block, but the example supplied is invalid in that it risks running the `finally` code several times in case of exceptions.

The `finally_done` variable is needed in case the code in the `finally` block itself throws to ensure that it is not executed again in the catch-all handler - it must be set to true before running the finally code inside the added try block.

```
try {
    bool finally_done = false;

    try {
        if(condition) { // Early return
            // finally_here
            return;
        }
    } catch(/* original exceptions */) {
        // code
    }

    // finally_here
} catch(...) {
    if(!finally_done) {
         // finally_here
    }
    throw;
}
```

Table 14: Finally emulation

One final detail is that a try statement with a finally block is not required to have any catch handlers. In such cases, the original try block is simply removed as the finally emulation replaces it.

4.8 Expressions

Just as with statements, most expressions in the Java language are shared with C++.

4.8.1 Evaluation order

C++ leaves the order of evaluation of compound expressions implementation defined in many cases. Thus, when evaluating a simple expression such as '`f() + g()`', it is not guaranteed that `f` will be evaluated before `g`. This is useful for example when '`+`' is a function that takes its arguments on a stack and thus needs the value of `g()` to be pushed first - instead of calculating expressions in order, the C++ compiler is free to reorder the calls and put the results directly on the stack. Assuming that `g()` was evaluated first and results in an exception, `f()` will not be evaluated at all, which intuitively might not be expected.

In contrast, Java has an intuitive and deterministic definition of execution order. Expressions are evaluated as expected, with left-to-right evaluation in general, while respecting operator precedence and parentheses. Function arguments are evaluated in the intuitive left-to-right order and if any expression terminates abruptly, later expressions are guaranteed not to be evaluated.

Thus, a careful translator must split compound expressions into their constituent parts, assigning intermediate results to temporary variables and then recombine the results.

If the translator is able to determine that a particular expression has no side effects, including being exception free, it becomes safe by extension to allow the C++ compiler to reorder it with respect to other expressions.

4.8.2 Lexical literals

Java inherited most of its literal syntax from C++, thus most valid Java literals are also valid C++ literals. There are however a few smaller differences, for example:

Negative integers in Java are parsed as one entity, whereas in C++ they are treated as a positive integer with a negative sign which in C++ leads to the smallest negative integer for a 32-bit 2's complement `int` being interpreted incorrectly as the positive part does not fit in the `int`:s allowable range (and the same for 64-bits)

Character literals are allowed to take the value of a single part of a surrogate pair using the '`\u`' escape, which is disallowed in C++ and needs to be rewriting using '`\x`' escapes instead - the same for string literals.

String literals return a `String` object in Java, not an array of characters, which in C+ + can be implemented using user-defined string literals

Floating point literals using binary exponents are not supported in C++ and must have their exponents adjusted.

4.8.3 Class literals

Class literals return a reference to the `Class` object associated with a certain class. In C++, we simply put a static function in each class that calls a runtime-defined function with the name of the class - implementing the function is up to the runtime.

The `Class` object contains metadata about the requested class and is part of the reflection support in Java. Interestingly, one of the most common reflection operations on the `Class` object is to get the name of the class, for example for logging purposes. Implementing this minimal support in C++ is trivial and allows many applications to work as expected that would otherwise need manual intervention.

This function can also be used to support an implementation of the `getClass` method inherited from `Object` by each Java class - by adding a special virtual method to `Object` and overriding that function in each class to return the translated class literal, `getClass` is guaranteed to return the class instance associated with the actual type of an instance.

4.8.4 Class instance creation

In both C++ and Java, new instances of classes can be created using the ‘`new`’ operator. Java allows anonymous classes to be defined as part of the class instance creation - such classes are actually subclasses or implementers of the type given to the new operator.

This is similar to how lambda expressions in C++ create a local anonymous type and return an instance of that type, with access to the enclosing instance and variables. However, lambda expressions in C++ are limited to a single method and can therefore not be used generally to replace anonymous classes - instead we treat anonymous classes as we would treat normal classes and generate the appropriate class definition separately and use the normal new expression to create an instance.

4.8.5 Array creation expressions

Since we are emulating Java arrays using classes, we need to replace array creation expressions with expressions that create an instance of the array class. The emulated array class has a special constructor taking a `std::initializer_list` parameter that allows the same array member initialization syntax in both languages using member lists delimited by brace symbols.

4.8.6 Field access

Field access follows the same pattern in Java and C++ - at the left hand side there is an expression returning an instance of a class while the right hand side contains the name of the field. Obviously, since we’re using pointers to represent references, we must use the arrow operator instead of the dot.

In Java, it is also possible to access the field of a base class using the super keyword - `x.super.y` becomes the field `y` in the superclass of the type of `x`. By adding a typedef named `super` to each class, we can use similar syntax to access `super` class fields: `x->super::y`.

Field access is subject to `null` reference checking - if a variable holds a `null` reference, an exception should be thrown. We pass the result of the left hand side expression through a function that implements the null check, thus "`x.a`" becomes "`npc(x)->a`" where `npc` is the null pointer checking function.

4.8.7 Method invocation

Intuitively, method invocation in Java and C++ is very similar. In both cases, a search list of methods is identified by examining method qualification, name and parameters, and then the best fitting method is chosen, if possible.

In both languages, the static types of the parameters is used to determine which method signature to use when making the call - the dynamic type of the parameters has no influence.

In Java, informally, if multiple methods could match the arguments, the most specific method is chosen, or in other words, the method requiring the fewest conversions. This intuitive rule is similar in C++ when classes and subclasses are involved, but different when integer conversions come in play.

In the example shown in Table 15, Java considers the conversion from `int` to `long` to be more specific than the same conversion from `int` to `double` - the `long` data type is closer to `int` than `double`. In C++, both implicit conversions are considered equal when choosing among matching methods, thus we need to apply an explicit conversion to the method arguments to avoid ambiguities:

```
class C
{
     void m(long x) { }
     void m(double x) { }
     void f() { m(42); }
}
```

Table 15: Ambiguity in C++, but not in Java

In Java, the method taking a long parameter is chosen over the double overload, while in C++, a cast is needed to ensure the correct function being called.

In both languages, if the chosen method is virtual, it is the dynamic type of the associated instance that determines the actual implementation being called.

4.8.8 Array access

In our emulation of Arrays in Java, for reference types, we use an array of Objects to store the references. Our `operator[]` then returns a copy of that reference, cast in accordance with the type of the array. This approach means that there is no way to write to arrays using the bracket syntax, as we cannot return a reference to the actual storage. Thus we rewrite all array writes to use a member method of the class implementing arrays, but continue to use the operator for reads.

4.8.9 Cast expressions

Java has a single cast operator whose syntax is inherited from C. It is used both for narrowing and widening conversions, and both with reference and primitive types. The same operator can be used for cross casting across a hierarchy, and failed casts result in a `ClassCastException` being thrown.

Two of C++'s cast operators are relevant for our translator. `static_cast` may be used for all widening casts (from subclass to superclass) and all numeric casts. For narrowing class conversions, and for casting across a hierarchy, we will instead implement a special function, `java_cast`, using `dynamic_cast` to do the actual casting. If a pointer cast fails using `dynamic_cast`, a null pointer is returned which our helper function translates into a `ClassCastException` being thrown.

4.8.10 Remainder operator

Unlike C++, the `%` operator in Java may be used with floating point operands as well as integral. When translating, the operator needs to be turned into a call to the `fmod` standard C++ function. Care must be taken to throw an `ArithmeticException` should the divisor be zero.

4.8.11 String concatenation operator

Java does not allow operator overloading to expand the meaning of the standard operators but does allow strings to be concatenated using the `+` operator. Strings may also be concatenated with other object and primitive types in which case the conversion method `toString` is used during concatenation.

In C++, we are representing string references as pointers, and thus cannot overload the `+` operator. Instead, we create an instance of the standard Java class `StringBuilder`, and use its append method to concatenate the strings, and finally call `toString`() to retrieve the result of the concatenation. This is similar to how the operator is implemented in popular Java compilers and guarantees that we will get the same string formatting when conversion is involved.

4.8.12 Shift operators

There are three shift operators in Java, left, signed right and unsigned right. Since there are no unsigned primitive types in Java, there is a special shift operator allowing unsigned semantics to be used with signed types. We implement the right unsigned shift by casting the left operand to its corresponding unsigned type.

4.8.13 Type comparison operator

Dynamic type comparison in Java is done using the `instanceof` operator. This operator returns true when it is guaranteed that a cast to the given type will succeed and the value given is not null. Using the `dynamic_cast` operator in C++ with the given reference, we can achieve the same effect by checking that the result of the cast is not null.

4.9 Limitations

There are a few features of Java that have no clear home in a native C++ application.

In a statically compiled language, it is generally assumed that the code that will be executing is available at compile time. Dynamic class loading in Java allows the virtual machine to load and unload classes dynamically, compiling them into machine code on the spot. The bytecode of the classes may be generated on the fly, downloaded from an external source or changed during the course of execution. Supporting such a feature in C++ would essentially mean implementing a full Java Virtual Machine that is able to compile byte code into machine code - if such functionality is required, it seems more beneficial to simply load an existing JVM in the first place.

Closely related to dynamic class loading is reflection. Reflection allows Java programs to discover type information at runtime, and to access fields and methods dynamically at runtime. Again, implementing full support for reflection, including dynamic method calling and field access, would require functionality close to that of a JVM. Closely related are annotations which are used at compile time by code analyzers and at runtime to provide additional reflection. Neither use is applicable in the converted source code.

Generics in Java have no direct correspondence in C++, and the translator takes the same approach as do Java compilers, by simply replacing them with explicit casts and generated bridge methods. In some cases, this decreases the clarity of the generated code by adding syntactic burden to otherwise simple expressions.

Several features of Java that are central to modern applications are implemented in the virtual machine and runtime environment or standard library. Threads, synchronization and memory handling are examples of such features, many of which are highly platform dependent, and their handling, or lack thereof, has been described in earlier sections.

As has been mentioned, Java Native Interface calls are not automatically translated. Supporting the Java Native Interface API in the converted code would burden the translated code with API details leading to an inefficient solution, and translating JNI use would require parsing C or C++ which is beyond the scope of this thesis.

Compared to the commercial Java to C++ converter by Tangible, the translation techniques described in this thesis offers a more complete language support. Nested and anonymous classes are supported as are final variables, static initializers and many of the other features described as limitations in the manual that accompanies the translator. Instead of attempting API conversion for the Java runtime classes as does the Tangible converter, the high degree of language support allows the use of an existing implementation by converting the referenced classes as well.

Chapter 5. Implementation and experimentation

During the course of this thesis, a translator was implemented to verify the soundness of the proposed translations. The implementation is written in Java as an Eclipse [26] plugin, taking advantage of the Java parser and dependency management bundled with Eclipse. It is available as an open-source project and can easily be installed and run from within Eclipse.

Figure 1 provides an overview of the main components of the implementation and its interaction with the services offered by Eclipse.

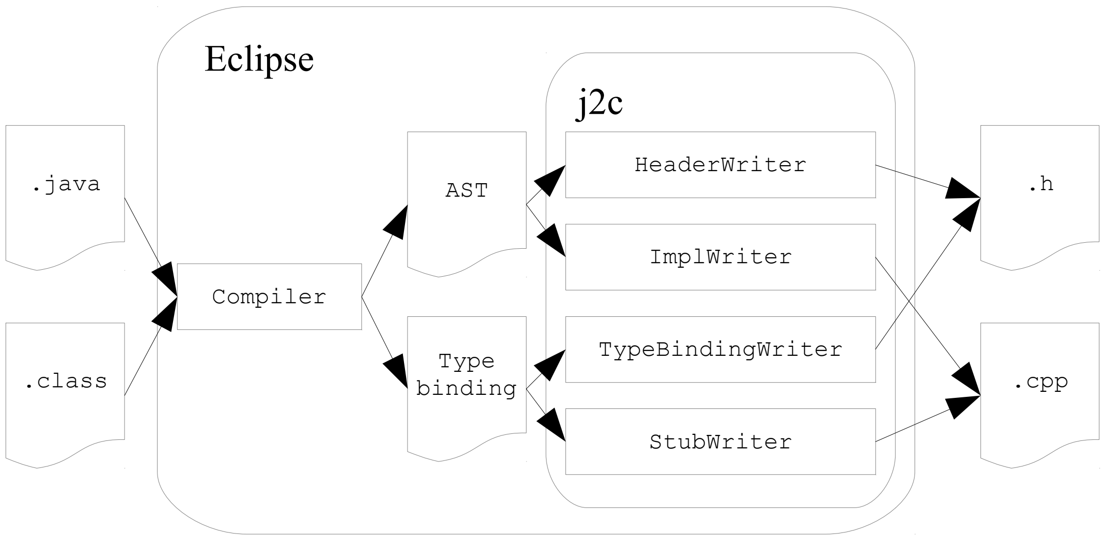

Figure 1: High-level overview of j2c

5.1 Implementation overview

The Java parser in Eclipse provides a type-resolved Abstract Syntax Tree (AST) that can be used to programmatically inspect a Java compilation unit. From the text of the compilation unit, a tree is built where each node roughly corresponds to a production in the Java grammar.

In addition to a tree representation of the source code, type information about each node is provided in the form of type bindings. Type bindings contain information about a type such as its declared methods, fields and subtypes, and can be generated either from source code or class files. For dependencies that have no source code available, the type bindings can be used to generate header and stub files that can be used as a starting point when providing a manually written implementation.

For the plugin to work correctly, a Java project must be created in Eclipse that specifies the details of the code being translated, such as dependencies and Java version. The Eclipse parser then processes each compilation unit resulting in an abstract syntax tree representation.

The core of the translator itself consists of several tree visitors and code generators, as well as a few utility classes that handle common logic such as naming rules.

For each file type written by the translator, i e header and implementation, a separate visitor class is used to traverse the tree. The visitor classes contain methods for each node type in the tree. Each method analyzes its tree node in its context and writes the corresponding C++ code to a string buffer, recording additional information that will be needed when writing the file preamble. The preamble consists of include directives for all dependencies of the class and some helper definitions for synchronization and casting.

For dependencies that lack source code, stub files are written based on the class type binding that includes fields, methods and nested types. For each method in the class, an empty implementation is written - this allows the application to be compiled and linked without missing references, but to run the application, implementations need to be written for the methods that will actually be called.

Stubs are also generated for methods marked as native in the Java source code. As there is no automatic translation for the Java Native Interface, such methods must be implemented manually. Table 16 provides an overview of the classes involved for processing different inputs and outputs.

File type	Source (.java)	No source (.class)	Common (.java and .class)
Header (.hpp)	`HeaderWriter`	`TypeBindingWriter`	`Header`
Implementation (.cpp)	`ImplWriter`	`StubWriter`	`Impl`
Native stubs (.cpp)	`StubWriter`	`StubWriter`	-
Common (.hpp and .cpp)	`TransformWriter`	-	-
Forward declaration (.hpp)	-	-	`ForwardWriter`
Makefile	-	-	`MakefileWriter`

Table 16: Implementation class names based on input and output type

The class `MainWriter` translates any main method, or entry point, encountered in the Java source code into a separate C++ file containing code that converts the traditional arguments of a C++ `main` function into the `String` array expected by Java.

After writing classes, the class `ForwardWriter` writes forward declarations on a per-package basis - this to reduce the number of generated files. Finally, the class `MakefileWriter` generates a `Makefile` that can be used together with the make tool [27] to build the converted code.

The generated code is built as a static library, and for each main method, a separate executable is generated. Appendix B contains a larger example of a Java class and the corresponding files generated by the translator.

5.2 Extending the translator

An additional feature of the plugin is to provide extension points for the translation. The plugin is written with the general case in mind, but by making assumptions about the source code being translated, it may be possible to generate better C++ code. Such assumptions may be realized use by implementing the `Snippet` interface whose methods are called at certain points of translation.

For example, OpenJDK contains a method called `ensureClassInitialized` that ensures that a particular class has been initialized, as it normally would be when first being used. This is a native method that calls the virtual machine to run initialization for a particular type, but we know that in the translated code, such initialization is handled by a static method named `clinit`.

Thus a snipped called `ReplaceInvocation` replaces all calls to `ensureClassInitialized` with a call to our generated `clinit` method, avoiding the need for a native implementation of the `ensureClassInitialized` method that would have been hard to write with no reflection support.

5.3 Experimentation

Tests carried out show that the translator is able to successfully convert the Java `System` class, and all of the ca 1000 top-level classes it depends on in the OpenJDK implementation.

In those dependencies, reflection is used at one point to initialize the encoder used to translate Java `String`:s into bytes of the native platform encoding, for example UTF-8. Also, 41 native methods, most of them trivial, need implementing to handle various tasks related to initialization and platform interaction.

With these methods implemented, it is possible to use the `System.out` member to write text to the console, as shown by the trivial example in Table 17.

Though the example may look trivial, its dependencies are not. Before a Java application can start, many of the features of the Java platform need to be initialized and checked. Character conversion, operating system feature discovery and security checks are examples of code that will run before control is given to user code.

Lacking formal verification, the test lends confidence in that our translator produces correct code, as the converted dependencies contain examples of most language constructs discussed in this thesis, including inheritance, virtual methods, arrays, class and instance initializers, exception handling and other advanced features.

Notable is the lack of use of reflection (with one exception), dynamic classloading and other Java-specific features that are mentioned in the section on limitations.

```
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("So long and thanks for the fish"),
    }
}
```

Table 17: Deceptively simple test application

On a Linux test platform, when compiled with GCC 4.7, the application binary weighs ca 14 MB excepting debug symbols, and uses ca 200 kB of heap allocated memory during execution with no garbage collector attached. The use of pruning techniques, such as the ones described by Varma [13] could help reduce the binary size of the application, but that kind of optimizations have not been applied in this case.

In other tests, the full source code of several libraries such as SWT, the full OpenJDK and H2 have been translated and compiled, but given time constraints and the amount of native methods present, it was not possible to test the resulting translation by actually executing the resulting code.

Performance-wise, the speed of translation is similar to that of compiling Java to bytecode, as can be expected. The translation of OpenJDK that consists of 17192 non-local classes and 2824 interfaces takes approximately 195 seconds on the test platform, averaging ca 10 ms/type, with no special attention to performance having been paid during the development of the translator.

Chapter 6. Conclusion

In this thesis, the problem of source-to-source translation from Java to C++ has been examined, from the perspective of being able to reuse existing Java software components in a C++ setting.

Through the individual translations for the statements and expressions of Java presented in Chapter 4 and the general principles outlined in Chapter 3, we are able to translate a substantial subset of the features offered by Java to readable and maintainable C++ code.

The generated code offers a high degree of similarity with the original code as most Java constructs carry over naturally to C++. Further, no special preparation of the original Java code, or manual modification of the generated code, needs to be done to achieve good conversion results, within the limits outlined in the section on limitations, making the process repeatable.

Experimentation shows that even for large projects such as OpenJDK, it is possible to generate working code with minimal effort compared to rewriting the code by hand, making the proposed approach a viable alternative when facing a requirement to reuse an existing code base. The fact that the converted code produces equivalent output after having performed a significant amount of tasks lends confidence in the correctness of the proposed translation.

Thus, through extensive language support and several options for runtime support, source-to-source translation becomes an attractive way of reusing Java code in from C++. Seen from a C++ perspective, Java code that was previously not available for reuse, because the existing alternatives such as JNI are not suitable, now can be accessed and used with little manual intervention.

In short, the conclusion is that source-to-source translation is a viable alternative offering distinct advantages over other methods of reusing Java code.

As always, the solution could be improved. Some of the manual intervention needed could possibly be avoided, and conversion rules that align better with idiomatic C++ could be researched, as outlined in the following section.

6.1 Areas of further research

In this thesis no attempts at performing API conversion have been made, not even of the core Java classes. To fit better with native code, and to avoid unnecessary external dependencies, it would be interesting to examine at least partial API conversion for the `java.lang` and `java.util` packages.

The performance of the generated code has not been examined during the course of this work. In general, one might assume that Java code is written with the presence a Just-In-Time compiler in mind, meaning that no attention is given to virtual method calls, casting and other aspects that a JIT may provide optimization for. Earlier studies show however, that the performance of the translated code largely depends on the programming techniques and compiler used, and that it should be at least comparable to that of native Java execution [15].

The Java Virtual Machine specification includes an API for accessing Java from native code and to implement Java methods in native code. If a translator could make fruitful use of such native code, there would be obvious benefits as less code would have to be manually replaced.

Although not required by the standard, it is assumed that a garbage collector be present to reclaim memory allocated dynamically during the course of program execution. The interaction of a garbage collector with the generated code, and other solutions such as reference counting could and should be further studied.

Formal verification of at least a subset of the features supported by the translation would lend confidence in that the converted code is indeed equivalent to the source code, but considering that the complexity of the source and target languages, such verification would require considerable effort. Complicating the fact is that there exists no complete formal semantics for C++ itself [28], even older versions than the one targeted here. Such an analysis would necessarily have to begin with establishing the semantics of the constructs we use in C++, for example the reliance on static initialization being thread safe that was only recently added.

Appendix A. Bibliography

1: TIOBE Programming Community Index, `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

2: Gosling, James and Joy, Bill and Steele, Guy and Bracha, Gilad, *Java(TM) Language Specification, The (3rd Edition)*, 2005

3: Terekhov, Andrey A. and Verhoef, Chris, *The Realities of Language Conversions*, 2000

4: J2C Java to C++ Converter, `http://code.google.com/a/eclipselabs.org/p/j2c/`

5: ISO, *International Standard ISO/IEC 14882:2011*, 2011

6: Parnas, D. L., *On the criteria to be used in decomposing systems into modules*, 1972

7: JNI: Java Native Interface, `http://docs.oracle.com/javase/6/docs/technotes/guides/jni/`

8: SWIG: Simplified Wrapper and Interface Generator, `http://www.swig.org/`

9: GCJ: The GNU Compiler for the Java(TM) Programming Language, `http://gcc.gnu.org/java/`

10: log4cplus: Logging library, `http://log4cplus.sourceforge.net/`

11: CppUnit: C++ unit testing framework, `http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page`

12: Boyle, James M. and Muralidharan, Monagur N., *Program Reusability through Program Transformation*, 1984

13: Ankush Varma , *A Retargetable Optimizing Java-to-C Compiler for Embedded Systems*, 2003

14: Proebsting, Todd A. and Townsend, Gregg and Bridges, Patrick and Hartman, John H. and Newsham, Tim and Watterson, Scott A., *Toba: java for applications a way ahead of time (WAT) compiler*, 1997

15: Translating Java Programs into C++ , `http://jklp.org/public/profession/papers/java2c++/paper.htm`

16: Trudel, Marco and Oriol, Manuel and Furia, Carlo A. and Nordio, Martin, *Automated translation of Java source code to Eiffel*, 2011

17: Java to C++ Converter, `www.tangiblesoftwaresolutions.com/Product_Details/Java_to_CPlusPlus_Converter_Details.html`

18: Jython: Python for the Java Platform, `http://www.jython.org/`

19: The Scala Language Specification, `http://www.scala-lang.org/`

20: OpenJDK: open-source implementation of the Java Platform, Standard Edition, `http://openjdk.java.net/`

21: Ralf Lämmel, Ekaterina Pek, Jürgen Starek, *Large-scale, AST-based API-usage analysis of open-source Java projects*, 2011

22: Hoare, C. A. R., *Monitors: an operating system structuring concept*, 1974

23: The Boehm-Demers-Weiser conservative garbage collector, `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

24: Choi, Jong-Deok and Gupta, Manish and Serrano, Mauricio and Sreedhar, Vugranam C. and Midkiff, Sam, *Escape analysis for Java*, 1999

25: Stroustrup, Bjarne, *The Design and Evolution of C++*, 1994

26: Eclipse Platform, `http://www.eclipse.org/`

27: GNU Make, `http://www.gnu.org/software/make/`

28: Michael Norrish, *A Formal Semantics for C++*, 2008

Appendix B. Example of converted code

In this appendix, a larger example, a simple merge sort implementation, is shown along with its translation as generated by the `j2c` translator.

B.1 Sort.java

```
public class Sort
{
    public static void mergeSort(Comparable[] a) {
        Comparable[] tmpArray = new Comparable[a.length];
        mergeSort(a, tmpArray, 0, a.length - 1);
    }

    private static void mergeSort(Comparable[] a,
        Comparable[] tmpArray, int left, int right)
    {
        if (left < right) {
            int center = (left + right) / 2;
            mergeSort(a, tmpArray, left, center);
            mergeSort(a, tmpArray, center + 1, right);
            merge(a, tmpArray, left, center + 1, right);
        }
    }

    private static void merge(Comparable[] a,
        Comparable[] tmpArray, int leftPos, int rightPos,
        int rightEnd)
    {
        int leftEnd = rightPos - 1;
        int tmpPos = leftPos;
        int numElements = rightEnd - leftPos + 1;

        // Main loop
        while (leftPos <= leftEnd && rightPos <= rightEnd)
            if (a[leftPos].compareTo(a[rightPos]) <= 0)
                tmpArray[tmpPos++] = a[leftPos++];
            else
                tmpArray[tmpPos++] = a[rightPos++];

        while (leftPos <= leftEnd)
            tmpArray[tmpPos++] = a[leftPos++];

        while (rightPos <= rightEnd)
            tmpArray[tmpPos++] = a[rightPos++];

        // Copy tmpArray back
        for (int i = 0; i < numElements; i++, rightEnd--)
            a[rightEnd] = tmpArray[rightEnd];
    }

    public static void main(String[] args)
    {
        mergeSort(args);

        String separator = "";
        for (String s : args) {
            System.out.println(separator + s);
            separator = ", ";
        }
    }
}
```

B.2 fwd.hpp

```
// Forward declarations for
#pragma once

#include <stdint.h>
#include <limits>

class Sort;
template<typename T> class Array;
typedef Array<char16_t> char16_tArray;
typedef Array<int32_t> int32_tArray;
typedef Array<int8_t> int8_tArray;
```

B.3 Sort.hpp

```
// Generated from /Sort.java

#pragma once

#include <fwd.hpp>
#include <java/lang/Object.hpp>

struct default_init_tag;

class ::Sort
    : public virtual ::java::lang::Object
{

public:
    typedef ::java::lang::Object super;
    static void mergeSort(::java::lang::ComparableArray *a);

private:
    static void mergeSort(::java::lang::ComparableArray *a,
        ::java::lang::ComparableArray *tmpArray_, int32_t left,
        int32_t right);
    static void merge(::java::lang::ComparableArray *a,
        ::java::lang::ComparableArray *tmpArray_, int32_t leftPos,
        int32_t rightPos, int32_t rightEnd);

public:
    static void main(::java::lang::StringArray *args);

    // Generated
    Sort();
protected:
    void ctor();
    Sort(const ::default_init_tag&);



public:
    static ::java::lang::Class *class_();
    static void clinit();

private:
    virtual ::java::lang::Class* getClass0();
};
```

B.4 Sort.cpp

```
// Generated from /se.arnetheduck.j2c.test/src/Sort.java
#include </Sort.hpp>

#include <java/io/PrintStream.hpp>
#include <java/lang/Comparable.hpp>
#include <java/lang/NullPointerException.hpp>
#include <java/lang/String.hpp>
#include <java/lang/StringBuilder.hpp>
#include <java/lang/System.hpp>
#include <java/lang/ComparableArray.hpp>
#include <java/lang/StringArray.hpp>

template<typename T>
static T* npc(T* t)
{
    if(!t) throw new ::java::lang::NullPointerException();
    return t;
}

::Sort::Sort(const ::default_init_tag&)
{
    clinit();
}

::Sort::Sort()
    : Sort(*static_cast< ::default_init_tag* >(0))
{
    ctor();
}

void ::Sort::ctor()
{
    super::ctor();
}

void ::Sort::mergeSort(::java::lang::ComparableArray *a)
{
    clinit();
    ::java::lang::ComparableArray *tmpArray_ =
        (new ::java::lang::ComparableArray(npc(a)->length));
    mergeSort(a, tmpArray_, int32_t(0),
    npc(a)->length - int32_t(1));
}

void ::Sort::mergeSort(::java::lang::ComparableArray *a,
   ::java::lang::ComparableArray *tmpArray_, int32_t left,
   int32_t right)
```

```
{
    clinit();
    if(left < right) {
        int32_t center = (left + right) / int32_t(2);
        mergeSort(a, tmpArray_, left, center);
        mergeSort(a, tmpArray_, center + int32_t(1), right);
        merge(a, tmpArray_, left, center + int32_t(1), right);
    }
}

void ::Sort::merge(::java::lang::ComparableArray *a,
    ::java::lang::ComparableArray *tmpArray_, int32_t leftPos,
    int32_t rightPos, int32_t rightEnd)
{
    clinit();
    int32_t leftEnd = rightPos - int32_t(1);
    int32_t tmpPos = leftPos;
    int32_t numElements = rightEnd - leftPos + int32_t(1);
    while (leftPos <= leftEnd && rightPos <= rightEnd)
        if(npc((*a)[leftPos])->compareTo((*a)[rightPos])
            <= int32_t(0))
            tmpArray_->set(tmpPos++, (*a)[leftPos++]);
        else
            tmpArray_->set(tmpPos++, (*a)[rightPos++]);

    while (leftPos <= leftEnd)
        tmpArray_->set(tmpPos++, (*a)[leftPos++]);

    while (rightPos <= rightEnd)
        tmpArray_->set(tmpPos++, (*a)[rightPos++]);

    for (int32_t  i = int32_t(0); i < numElements; i++, rightEnd--)
        a->set(rightEnd, (*tmpArray_)[rightEnd]);

}

void ::Sort::main(::java::lang::StringArray *args)
{
    clinit();
    mergeSort(args);
    ::java::lang::String *separator = u""_j;
    {
        auto _a = npc(args);
        for(int _i = 0; _i < _a->length; ++_i) {
            ::java::lang::String *s = (*_a)[_i];
            {
                npc(::java::lang::System::out())
                   ->println(::java::lang::StringBuilder()
                       .append(separator)
                       ->append(s)
                       ->toString()
                   );
               separator = u", "_j;
           }
       }
   }
}

extern ::java::lang::Class *class_(const char16_t *c, int n);

::java::lang::Class *::Sort::class_()
{
    static ::java::lang::Class *c = ::class_(u"Sort", 4);
    return c;
}

void ::Sort::clinit()
{
    super::clinit();
}

::java::lang::Class *::Sort::getClass0()
{
    return class_();
}
```

B.5 Sort-main.cpp

```
#include </Sort.hpp>

extern void init_jvm();

int main(int, char**)
{
    init_jvm();

    ::Sort::main(/* TODO convert args to string array */nullptr);

    return 0;
}
```