INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Ephedra

A C to Java Migration Environment

by

Johannes Martin

M.Sc., Northern Illinois University, 1996

A Dissertation Submitted in Partial Fulfilment
of the Requirements for the Degree of

Doctor of Philosophy

in the Department of Computer Science.

We accept this dissertation as conforming
to the required standard.

Dr. H.A. Müller, Supervisor, Department of Computer Science, University of Victoria

---

Dr. R.N. Horspool, Department of Computer Science, University of Victoria

---

Dr. J.H. Jahnke, Department of Computer Science, University of Victoria

---

Dr. K.F. Li, Department of Electrical and Computer Engineering, University of Victoria

---

Dr. S.R. Tilley, Department of Computer Science, University of California, Riverside

Supervisor: Dr. H.A. Müller

Abstract

The Internet has grown in popularity in recent years, and thus it has gained importance for many current businesses. They need to offer their products and services through their Web sites. To present not only static content but also interactive services, the logic behind these services needs to be programmed.

Various approaches for programming Web services exist. The Java programming language can be used to implement Web services that run both on Internet clients and servers, either exclusively or in interaction with each other. The Java programming language is standardised across computing platforms and has matured over the past few years, and is therefore a popular choice for the implementation of Web services.

The amount of available and well-tested Java source code is still small compared to other programming languages. Rather than taking the risks and costs of redeveloping program libraries, it is often preferable to move the core logic of existing solutions to Java and then integrate it into Java programs that present the services in a Web interface.

In this Ph.D. dissertation, we survey and evaluate a selection of current approaches to the migration of source code to Java. To narrow the scope of the dissertation to a reasonable limit, we focus on the C and C++ programming languages as the source languages. Many mature programs and program libraries exist in these languages.

The survey of current migration approaches reveals a number of their restrictions and disadvantages in the context of moving program libraries to Java and integrating them with Java programs. Using the experiences from this survey, we established a number of goals for an improved migration approach and developed the *Ephedra* approach by closely following these goals. To show the practicality of this approach, we implemented an automated tool that performs the migration according to the *Ephedra* approach and evaluated

the migration process and its result with respect to the goals we established using selected case studies.

Ephedra provides a high degree of automation for the migration process while letting the software-engineer make decisions where multiple choices are possible. A central problem in the migration from C to Java is the transformation of C pointers to Java references. *Ephedra* provides two different strategies for this transformation and explains their applicability to subject systems. The code resulting from a migration with Ephedra is maintainable and functionally equivalent to the original code save some well documented exceptions. Performance trade-offs are analysed and evaluated in the light of the intended subject systems.

Examiners:

Dr. H.A. Müller, Supervisor, Department of Computer Science, University of Victoria

Dr. R.N. Horspool, Department of Computer Science, University of Victoria

Dr. J.H. Jahnke, Department of Computer Science, University of Victoria

Dr. K.F. Li, Department of Electrical and Computer Engineering, University of Victoria

Dr. S.R. Tilley, Department of Computer Science, University of California, Riverside

Contents

List of Tables

List of Figures

Part I

Overview and Survey

Chapter 1

Introduction

1.1 Motivation

In its relatively short history, the Internet has shown tremendous growth, both in the number of services offered and in the number of users [12]. A recent survey shows that almost 60 percent of North America's population uses the Internet, as does more than eight percent of the entire world population [56]. Some businesses realised the potential of Internet commerce early on, and thus have tried to attract new customers and keep current customers by offering their services also through the Internet. To stay competitive in the global marketplace, their competitors have to follow suit and migrate their services and products to also offer them online through Internet clients.

For many applications, for example browsing product catalogues or obtaining account balances from a financial institution, it is sufficient for the Internet clients to access the data stored on the company's information systems. As soon as value-added services are to be offered, it is desirable to have the Internet client not only access data from the company's information systems, but also perform computations on the data. Offloading these computations from the central servers helps to keep the servers available for other tasks. If the Internet client can perform the computations independently, delays through network congestion or heavy loads on the central servers can be avoided, thus improving customer satisfaction.

Internet clients commonly run as part of a Web browser. There are several approaches to running a client within a Web browser. *Java Script* is an interpreted language that is embedded into HTML documents. The language is not large, and Web browsers support various dialects, so a cross platform deployment of complex applications is difficult. Many Web browsers allow for so called *Plug-ins* to extend the functionality of the Web browser. These are usually platform specific and need to be installed on every client machine, so using them in a large heterogeneous network is difficult, in particular if the software needs to be updated frequently.

Most Web browsers include a *Java Virtual Machine* (JVM) that allows them to run programs written in the *Java programming language* (Java). Java is a modern object-oriented programming language and comes with large class libraries. Both the JVM and Java are standardised, and Java programs can be downloaded by the Web browser on demand, making it simple for system administrators to deploy and update these programs. Java programs are compiled to *Java Byte Code* before their deployment, and this byte code is interpreted by the JVM at run-time, so a Web browser can verify that programs downloaded from untrusted parties do not compromise system security. Java is therefore a popular programming language for writing clients of distributed Internet applications.

The importance of Java is also growing on the server-side of distributed Internet applications. Server-side Java applications either communicate with client-side Java applications or generate HTML pages to present data generated by these applications. Since Java Virtual Machines have recently been optimised for integration into Web servers, they can offer good performance. Java Byte Code is stored in a binary format and can therefore be interpreted much more efficiently than other scripting and programming languages that are popular on Web servers and need to be interpreted at execution time. If both the client- and the server-side of an application are coded in Java, the communication between client and server is simpler than if they are coded in different programming languages that possibly use slightly different communication protocols.

1.2 Problem

To avoid a difficult, risky, and thus expensive redevelopment of the business logic that is already present and well-tested in current legacy applications, it is desirable to integrate parts of these legacy applications — usually written in a legacy programming language such as COBOL, Fortran, or C/C++ — into the new clients and servers, written in Java.

In this dissertation, we focus on the integration of programs written in C and C++ into Java programs. For this integration to be successful, the differences between C/C++ and Java need to be known. Chapter 2 explores the most important differences. Many of them stem from the different goals the designers of these programming languages had: C was to be a language to support low-level and high-performance system programming while Java's developers put an emphasis on the security that was needed for applications to run within Web browsers. Chapter 4 surveys and evaluates a number of current approaches to reconciling the differences between C and Java. In particular, integration of native binary code, C to Byte Code compilation, re-implementation, and source code transliteration approaches are discussed.

In the remainder of this dissertation, we will use the term "C" to refer to both the C and C++ programming languages. If we want to emphasise that something applies only to the C++ programming language, we will use the term "C++".

1.3 Approach

We deem a conversion of the C source code to Java source code the most appropriate integration strategy for the purpose of turning monolithic legacy applications into distributed Internet applications. Researchers familiar with source code conversion have formulated a number of requirements that we analyse and assess in their importance for our needs (Chapter 5). We then formulate a coarse overall strategy for the source conversion and explain it in detail in Part II of this dissertation: *Ephedra*. To gain confidence in our strategy, we implemented it in software and tested it on a number of case

studies (Chapters 10 and 11).

Our strategy roughly follows Malton's *phases of source conversion* (Chapter 6.1). In the first phase (*normalisation*), K&R C [33] source code is converted to ANSI C [1] to facilitate type checking and to remove obvious errors in the code that had been undetected before because of the lack of prototypes. Still in this phase, type casts between data types are analysed to find hardware dependent code and relationships between these data types. Those relationships are then used to create class hierarchies out of these data types and to eliminate Java-incompatible type casts. In the second phase (*translation*), the normalised C source code is converted to Java source code. This code can optionally be improved in the third and last phase (*optimisation*).

At every stage of this process, the source code remains compilable and executable, so errors introduced during the conversion process can be detected and corrected early. We implemented tools to support and automate the conversion process. Part II describes the approach in detail.

1.4 Outline

This dissertation first explains the most important differences between C and Java and shows how they cause problems in the migration and integration of legacy C programs into Java applications (Chapter 2). Related commercial applications and research contributions are then presented in Chapter 3. Chapter 4 surveys those current strategies that are most relevant for migrating and integrating legacy C programs into Java applications, and shows whether and how they address the language differences and integration problems.

The experiences gained in this survey are then used to establish goals (Chapter 5) for a new conversion approach which is introduced in Part II. Part III evaluates this new approach by showing a software implementation of our strategy, presenting case studies on the conversion of C and C++ programs from various application domains done using this implementation, and discussing the quality of the code generated in these case studies with respect to the goals set in Chapter 5. Finally, open problems, future work and applications are discussed in Chapter 13.

1.5 Summary

In this chapter, we motivated the need for the migration of legacy applications to Internet platforms. We presented a rough overview of the main problems in a migration from C/C++ to Java in particular. We explained our approach to the solution of these problems and gave an overview of the remainder of this dissertation. In the next chapters, we investigate the main problems that arise in migrating C programs to Java and survey existing approaches and current research in migration and language conversion.

Chapter 2

Problem Definition

As explained in *The Java Language Specification* [28], Java is related to C and C++, but there are important differences that pose problems in a migration from C to Java. We investigated these differences by comparing the specifications of the Java programming language [28] and the Java Virtual Machine [39] with the specifications of C and C++ [33, 1, 61].

The most important differences are:

Preprocessor: C programmers can use *include files* and *macros* to avoid duplicating some code. They are expanded during the preprocessing of the source code. Java does not have a preprocessor.

Macros have been used mostly for defining constants and small functions, but in some cases they define arbitrary code fragments that might not be syntactically complete when seen independently. It is difficult to convert these latter ones to Java in a meaningful way.

Control Flow: The control flow statements of Java are mostly equivalent to those of C. One exception are `goto` statements, which are not supported in Java. However, Java allows labelled `break` and `continue` statements within loop statements that can be used in the conversion of most `goto` statements.

Expressions: C compilers perform many widening and narrowing type conversions between fundamental types implicitly where necessary. For ex-

ample, no explicit type conversions are required for assignments between floating point and integral variables. Java requires explicit type conversion wherever a type conversion could result in the loss of precision.

C compilers and run-time environments do not check whether incorrect type conversions are attempted. When a variable is converted to a type it is not actually compatible with, the error often goes unnoticed. Variables are not guaranteed to reference storage that actually contains an object of the variables' data types. In Java, type conversions that cannot be validated during compilation because the actual types of objects are unknown at that time are verified for their correctness at run-time. Variables are therefore guaranteed to reference objects of the variables' data types.

C allows the programmer to have a number of expressions evaluated in sequence with the result of only the last expression being kept using so called *comma expressions*. They are frequently used in macros to achieve side-effects. Some C compilers provide language extensions, that allow for arbitrary statements to be included in such a sequence. These language extensions and C comma expressions have no equivalent in Java. Source code using these language features is therefore difficult to convert to Java.

Data Types: In Java, the domains of the primitive data types are guaranteed to be the same for all compilers and platforms. In C, this is true for only some of the data types.

All but one of the primitive Java data types are signed. In C, the programmer can specify for every integral variable whether it should be signed or unsigned.

In C, data types can be named and aliased using `typedef` directives. No equivalent construct exists in Java.

Structured types can be allocated statically or dynamically in C, in Java their allocation is always dynamic. In C, *pointers* contain the physical memory addresses of statically or dynamically allocated storage and can

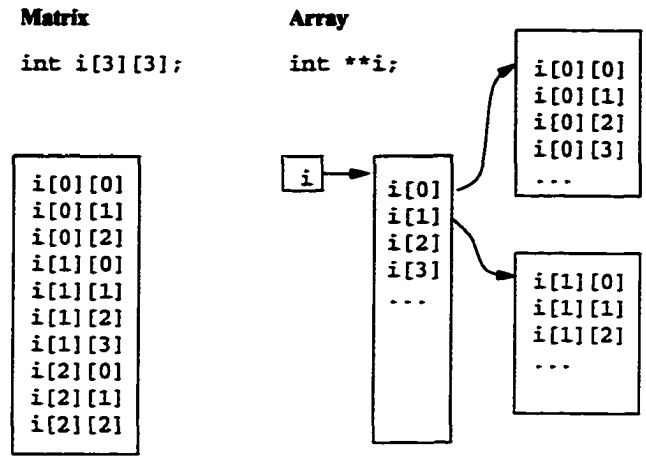

Figure 2.1: C matrices and arrays

be freely manipulated using arithmetic operations, while Java *references* are abstract and immutable.

C allows for type conversions between and among primitive and structured types, and type checks can be overridden. Java enforces strict type checking both at compile and run-time.

C supports both *matrices* and *arrays* (see Figure 2.1). Their differences are often overlooked even by experienced C programmers, and become apparent only in their multi-dimensional versions. Multi-dimensional matrices must have fixed bounds on all but the high-order dimension and are internally represented by a uni-dimensional array whose size is a multiple of the number of elements in the low-order dimensions. Multi-dimensional arrays may be unbounded in all dimensions and are represented internally through several layers of pointers. Bounds are checked for neither matrices nor arrays, even though some bounds are known already at compile time.

Only *arrays* are supported by Java. They are internally represented by a reference to the storage containing the array elements. Arrays are treated as structured types and as such inherit from the Java `Object` class. The Java Virtual Machine checks the bounds of all arrays at compile and

run-time.

The order of indexing multi-dimensional arrays is the same in C and Java, with the highest-order index followed by the lower-order indices. Some languages, such as FORTRAN, use different conventions.

Inheritance: C++ supports multiple implementation inheritance. Java allows only single implementation inheritance, but also provides multiple interface inheritance.

Memory Management: C supports static, automatic, and dynamic allocation of memory for both primitive and structured types. Static and automatic variables are implicitly allocated and deallocated, dynamic variables need to be explicitly allocated and deallocated. In C++, *constructors* and *destructors* are executed at known times during allocation and deallocation, respectively.

In Java, primitive variables can be static or automatic, and are allocated and deallocated implicitly. Structured variables and arrays are dynamic and need to be allocated explicitly. Their *constructor* is executed at that time. They are deallocated by the Java Virtual Machine some time after it has determined that they are no longer used. Their *finalizer* is executed at this usually unknown time.

Exceptions: Both C++ and Java support exceptions. In Java, methods have to declare which exceptions they may throw. In C++, these declarations are optional.

In Java, run-time exceptions and errors, such as illegal storage accesses and overflows, are detected and reported using the Java exception mechanisms. In C, they may be caught and reported using the operating system's exception mechanisms or go unnoticed.

Parameterised Types and Methods: C++ supports parameterised types and methods through so called *templates*. Java has no direct equivalent. Java arrays can be used to achieve similar functionality in some cases.

Source Organisation: In C and C++, declarations and definitions of data types, functions, and variables, may (and in some cases must) be separated. They are commonly spread across source files. C++ *namespaces* allow the developers to logically group parts of their sources.

In Java, definitions and declarations are one. The sources can be organised using *packages* and *classes*. The physical placement of the source files on the permanent storage media has to follow the logical package and class structure of the software.

Run-time Environment C did not originally provide a standard run-time library. A set of library functions evolved and was later standardised. The situation is similar for C++.

A minimal set of standard libraries is defined for Java [28]. Most implementations of the Java Virtual Machine come with an extended set of standard libraries.

Multi-Threading Multi-threading and the synchronisation of threads is not supported by C language concepts. It can be implemented through run-time libraries or language extensions.

In Java, support for threads and synchronisation is realised through Java language concepts (*monitors*) as part of the standard library.

Chapter 3

Related Work

Migrating source code from C to Java is a hard problem with many facets: as C is a procedural language and Java is an object-oriented language, not only the syntax and semantics of the source code need to be translated, but also a paradigm shift is necessary to move from procedural to object-oriented code. The following sections review related software migration approaches.

3.1 Language Conversion

In their paper *The Realities of Language Conversions*, Terekhov and Verhoef give an account of their experiences with language conversions [63]. The examples they provide deal mostly with COBOL systems but apply to many instances of language conversions. They argue that the difficulties of such conversions are often underestimated and manifold: Too much emphasis is put on technology and tools that claim to aid in language conversion and too little attention is paid to training of the personnel that has a major impact on the success or failure of the migration project.

They articulate a number of important facts about source conversion, namely:

- Candidates for language conversion are usually the most critical systems of a business; thus, an emphasis must be put on the reliability of the conversion process.

- The software engineers performing the conversion must be experts in both the source and target languages to realise the intricate differences between the languages and the problems that can arise out of them.
- A converted system will not be as well designed as a system developed specifically for the target programming language.
- The more similar the source and target languages are, the more difficult will be the detection of their differences: syntactically close or identical source artifacts might have big semantic differences.
- It is very difficult or even impossible to go from a rich source language to a minimal target language.

Terekhov and Verhoef list several requirements that have to be met to achieve successful source code conversion. We will come back to these in Chapter 5. They also propose a coarse three-step process for source conversion.

In *The Migration Barbell*, Malton notes that there are many ad-hoc techniques for source conversion, but few systematic approaches [43]. He formalises the process proposed by Terekhov and Verhoef. We use his process description in the introduction of the Ephedra approach (Section 6.1). Malton also defines a set of goals for source conversion and identifies three distinct conversion styles, which are listed in ascending order of complexity:

Dialect conversion is the conversion of a program from one dialect of a programming language to another dialect of the same programming language. This usually has to be done, if a new version of a compiler is used to build the system, or if a different compiler product is selected.

API migration is the adaptation of a program to a new set of APIs. This occurs for example, if a different database or user interface is chosen for an information system.

Language migration is the conversion from one programming language to a different one. It may involve dialect conversion and API migration.

Malton's observations are based on dialect conversions in the COBOL, PL/I, and RPG domains, and pilot studies in source conversion from COBOL to Java, RPG to COBOL, and SQL to SQLj.

There are a number of papers describing experiences and lessons learned from source conversion projects. Kontogiannis *et al.* report on the conversion of the IBM compiler back-end from a PL/I derivative to C++ [34]. Yasumatsu describes a system for translating Smalltalk programs into a C environment [69]. Terekhov presents a case study on an automated language conversion project from a proprietary language to Visual Basic and COBOL [62]. Cordy *et al.* developed *The TXL Transformation System* as a programming language and rapid prototyping system specifically designed to support computer software analysis and transformation tasks [13].

3.2 Paradigm Shift

Another level of complexity is added to source conversion if it involves a paradigm shift. With the growing popularity of object-oriented languages, there have been many attempts to move from procedural to object-oriented systems. One of the major problems in this particular paradigm shift is the identification of candidates for classes and their members.

A common approach is to use data structures in the legacy system as basic building blocks for classes and to add functions that operate on these structures as methods to those classes [9, 40, 41, 70, 34].

A different approach is to use design documents of a subject legacy system, such as structure charts and data-flow diagrams, to recover a possible object-oriented architecture for the legacy system [25, 26].

Cimitile *et al.* centre the identification of classes around persistent data stores, such as files or tables in a database, with functions as candidate methods for these classes [11].

There are other paradigm shifts that may occur concurrently with the shift from procedural to object-oriented code. For example, C has a very flexible and lax memory access scheme using *pointers*, while the Java programming language imposes strict rules on memory management using *references*. So, in

a conversion from C to Java, two paradigm shifts have to be made.

Demaine developed a general method for converting C pointers to Java references, and also to Fortran [16]. He targets primarily scientific applications and describes two approaches for converting pointers to references and that can be combined to handle most cases of pointers in C. His theoretical considerations are well-founded and plausible, but unfortunately, he does not provide an implementation to show the feasibility of his approach. He also fails to motivate how scientific applications, that are usually CPU intensive and performance critical, can benefit from a conversion to Java. Demaine illustrates the code transformations on small isolated examples, but it is not clear whether and how they will work in complex expressions. Our techniques for mapping pointers to references are similar to his approach. We describe the commonalities and differences of these techniques in more detail in Chapter 8.

3.3 API Conversion

API conversion happens in the context of many maintenance and migration tasks. Many software engineers deal with API conversion on a regular basis when they have to adjust their products to new releases of third-party libraries these products depend on. Though new releases of libraries are usually compatible with earlier releases, they may depend on undocumented features or errors in the old versions of libraries. Netscape and GNU make on Linux systems are examples of programs that ceased to work correctly after a library upgrade [55].

If a library is to be replaced by a new library with a completely different interface, the task becomes more difficult. This may be necessary when a program is to be moved from one operating system to another, of if it is to use a different database system. One common approach is to use a set of wrapper functions that conform to the old library's interface and invoke the new library's functions. IBM used this approach to make it easier to migrate Windows applications to its OS/2 operating system [10].

3.4 Automated Conversion Tools

Many products on the market promise to help in the migration from one programming language to another. We reviewed selected tools that deal with the particular problem of moving source code from C to Java or the Java Virtual Machine.

Laffra and Tilevich present an automated C++ to Java transliteration tool [35, 65]. Their tool is rather simple and only succeeds in converting C++ constructs that have close Java equivalents. It is a significant help in the conversion of large volumes of legacy C++ code, but the software engineer has to do substantial manual work to finish the transliteration.

Novosoft has released a C to Java transliterator that has been proven to transliterate large volumes of code correctly (for example the PGP encryption software) [50]. However, their mapping of C data types to Java is non-intuitive and circumvents many Java security and run-time features such as garbage collection and memory protection. An integration of the transliterated code with mainstream Java code is quite difficult.

Waddington generates Java Byte Code from C source code [67]. This allows for more freedom in control flow where the Java language is too restrictive. His mapping of data types to Java is similar to the one used by Novosoft, and thus the generated code is difficult to integrate with mainstream Java code and poses some risks because of the circumvention of type checking and storage protection.

3.5 Dimensions of Migration

When performing a migration of a legacy system to a new technology or platform, one usually has to make certain trade-offs with respect to the anticipated qualities of the conversion process and the new code.

Architectural re-design of a legacy system requires certain amounts of knowledge of the application domain. This knowledge cannot usually be reconstructed from the source code but only from supporting documentation and experts in the domain. Today's software engineering tools are still mostly

incapable of dealing with anything but structural information, and thus provide little support for automated re-design. A software engineer is required to make choices based on domain knowledge where the tool cannot determine a proper choice based on structural information.

To deal with large legacy systems, on the other hand, a high level of automation is desirable. It is impractical to let highly-paid software engineers perform changes on millions of lines of code. Conversion techniques for the transformation of large volumes of code therefore need be designed with the possibility of automation in mind.

The higher the level of re-design, the better is the opportunity to produce source code that conforms to generally accepted style and coding guidelines of the target language. In the case of Java as target language, this conformance also supports Java's built-in security and safety features. Generally, standards conformance supports program understanding and facilitates future maintenance tasks.

The approaches mentioned in the previous sections put their emphasis on different dimensions of language conversion. While some focus on a high level of automation and the ability to migrate large volumes of code, others centre more on architectural re-design to achieve a high level of conformance with the target language, and in the case of Java, the security constraints of the underlying hardware platform. Table 3.1 shows how some of the aforementioned techniques fit into this migration space. It is important to note that all three columns of the migration space impact the cost of the migration in the long run. While, at first sight, a high level of automation is the biggest cost factor, investments made into re-design, security, and language conformance will help to reduce maintainability problems in the long term.

3.6 Summary

In this chapter, we presented some of the different kinds and aspects of source code migration, namely language conversion, paradigm shift, and API conversion. We discussed related research in these areas including tools for the automated conversion of C source code to Java. We identified several prop-

Approach	Re-design	Automation, Scalability	Security, Language Conformance
Identification of classes [9, 40, 41, 70, 25, 26, 11]	high	varying	high
Kontogiannis [34]	medium	high	medium
Demaine [16]	little	high	medium
Laffra, Tilevich [35, 65]	little	medium	high
Novosoft [50]	none	high	little
Waddington [67]	none	high	little

Table 3.1: Dimensions of migration

erties pertaining to source code migration approaches and characterised the available tools and research contributions according to these properties. The next chapter describes a selection of these tools in more detail and evaluates their suitability for the integration of C source code into Java applications to be deployed on Internet platforms.

Chapter 4

Survey of Current Migration Strategies

There are two main approaches for the integration of C code with Java. In the first approach, the Java Virtual Machine is extended using code compiled to the native machine language of the target system. Section 4.1 discusses this approach. In the second approach, the C code is compiled for the Java Virtual Machine, either directly (Section 4.2) or by first converting the C source to Java. This conversion can be done either by translating the C data types and functions to mostly equivalent Java data types and functions (*transliteration*, Section 4.4), or by recovering the design and algorithms used in the C code and re-implementing them in Java (Section 4.3).

4.1 Integration of Native Binary Code

Being a virtual machine, the JVM does not execute programs at the speed that could be obtained with programs compiled to native machine language and does not allow access to many features particular to a specific concrete hardware platform.

To allow developers to implement performance critical code in the native machine language of a computer (either by coding it in assembly or using a native compiler) and to take advantage of features of a hardware platform

that are not exploited by the JVM, the *Java Native Interface* (JNI) has been designed [38].

Migration of C code to be integrated into Java programs using JNI is easy in that no or little modifications are necessary to the C source code. However, it is necessary to provide interface classes that handle the communication between the C and Java parts of the program. While the generation of these interfaces can be largely automated, they constitute a major performance overhead at run-time, as the interfaces frequently have to perform data type conversions. One has to evaluate carefully whether the performance gain through the implementation of code in C justifies the performance loss incurred through the interfaces.

A major drawback in the use of JNI is the loss of platform independence. Since the C code has been compiled to the native machine language of a particular hardware platform, the combined C/Java program will only run on this particular platform. This may be acceptable for server applications that run on only one particular platform but is not suitable for client applications that usually need to run on various different platforms.

Since the C code is not executed within the safe environment of the JVM, it is susceptible to failures and security breaches that could be prevented if it were running under the control of the Java Virtual Machine.

4.2 C to Byte Code Compilation

Though the Java Virtual Machine was designed with the Java programming language in mind, it is general enough to support many other programming languages. A comprehensive list of programming languages available for the JVM can be found in [66]. The *Java Back-End for GCC* implements a C compiler that generates machine language for the Java Virtual Machine [67]. While this appears to be a good strategy, it has been implemented in a way that makes integration with Java programs difficult and circumvents Java's type safety.

To map the memory handling of C run-time environments onto the Java Virtual Machine accurately and to work around the strict type checking of

the JVM, a large array is allocated that is used to store all variables used in the C source code. As in a C program compiled to native machine code, faulty programs may cause corruption of this entire array, thereby causing unexpected errors that are difficult to debug. One can argue that this is not a problem when dealing with mature program libraries, but even these have the occasional bug and need to be enhanced and maintained, and a software engineering tool will be more useful if it helps locating sources of errors in the software.

Another problem with this approach is that special interfaces are needed for the Java code to communicate with the C code. Support routines are needed to build Java objects out of the raw data in the memory array. As with JNI, these conversions create a potentially significant run-time overhead.

4.3 Re-Implementation

One way to turn source code from one programming language into another is to use the original design documents of the code and follow a forward engineering approach to re-implement the code using these documents in the new target language. In many cases however, these original design documents do no longer exist, or do not reflect the real architecture and functionality of the system: As Lehman states in his *Laws of Software Evolution*, software "systems must be continually adapted else they become less satisfactory" (*Law of Continuing Change*) [36, 37]. Because of time pressures, design documents are rarely kept up to date with the development of the code. When this is the case, the current design needs to be recovered from the source code (*reverse engineering*). Once the design has been documented, it can be implemented in the target language, possibly after refinement and adaptation to features of the target language or anticipated changes. Depending on the degree of re-design desired, the abstraction can be taken to different levels (see Figure 4.1).

This approach is often taken if the current system has maintenance or performance problems, or large changes to the requirements are anticipated. A number of papers that describe this approach have been published [5, 8, 53]. Usually only parts of this process can be automated. Opdyke [52] and Fowler

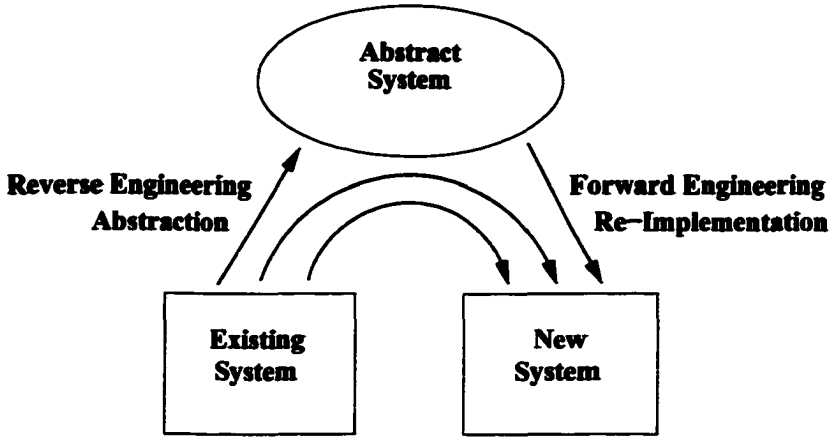

Figure 4.1: Levels of re-engineering [46]

et al. [23] present *refactoring*, behaviour-preserving reorganisation of source code, as a means for architecture refinement that can be well automated.

As we are looking for an automated approach that is suitable for converting large amounts of source code, and have no imminent desire to change the design of the source code, we did not pursue this option, but focused our attention on source code transliteration.

4.4 Source Code Transliteration

With the transliteration approach, the original source code is converted to the target language, while changing the data structures and program logic as little as possible. Various degrees of change are possible: by emulating the source language's data types in the target language (*data type emulation* [63, 69]), the amount of change can be kept low. If data types used in the original source are substituted by data types of the target language, the code may have to be changed more to deal with the differences of the data types.

To explain the differences between the approaches to transliterate C source code to Java better, we present a small sample C program that exhibits some of the difficulties in C to Java migration (Figure 4.2). In particular, it shows pointers to primitive data types, structure assignments, and function calls with

```
struct s1 {
  int i;
};

struct s1 foo(int *i, struct s1 s) {
  while (s.i++, *--i > 0) {
    int h;
    for (h = *i; h < s.i; h += *i)
      if (h % 3 == 0)
        goto continue_while;
      else
        if (h % 2 == 0)
          goto break_while;
  continue_while:
  }
break_while:
  s.i += *i;
  return s;
}

void bar() {
  struct s1 s = { 5 };
  struct s1 t;
  int* pi = malloc(100 * sizeof(int));
  int i;
  for (i = 0; i < 100; i++)
    pi[i] = i;
  t = foo(pi + 100, s);
}
```

Figure 4.2: Sample C program

non-primitive call-by-value parameters. It also employs goto statements and comma expressions.

4.4.1 C2J++

C2J++ is a tool for converting C++ classes to Java classes [35, 65]. It transliterates data types and control flow quite well as long as they are similar in C++ and Java, but struggles where there are differences.

Figure 4.3 shows the transliteration result for the example C source code as produced with C2J++. As C2J++ expects all methods and variables to be within a class, the sample code was slightly modified before the transliter-

```
class s1 {
  int i;
}

class foobar {
  public s1 foo(int  i, s1 s) {
    while (s.i++,  --i > 0) {
      int h;
      for (h =  i; h < s.i; h +=  i)
        if (h % 3 == 0)
          goto continue_while;
        else
          if (h % 2 == 0)
            goto break_while;
    continue_while:
    }
  break_while:
    s.i +=  i;
    return s;
  }

  public void bar() {
    s1 s = { 5 };
    s1 t;
    int  pi =
      malloc(100   sizeof(int));
    int i;
    for (i = 0; i < 100; i++)
      pi[i] = i;
    t = foo(pi + 100, s);
  }
}
```

Figure 4.3: C2J++ transliteration result

ation. C2J++ transliterates the C source into a syntactically mostly correct Java program. It fails to recognise and convert comma expressions and goto statements. Data type conversions are done in a trivial way, usually by removing address and dereference operators. C2J++ fails to distinguish between a star as used in a multiplication and a star used to dereference a pointer, and simply removes any star it encounters. The expression inside the `malloc` statement in our example exhibits this problem. The different assignment and parameter passing semantics of C and Java are ignored, with the result that the transliterated program has a very different behaviour than the original.

C2J++ flags the changes it has made with comments (they have been removed in Figure 4.3 to increase readability), so the programmer can review and correct the transliterated code. While C2J++ can assist a developer in the integration of C code into Java programs, extensive manual efforts are necessary to transliterate large volumes of source code.

4.4.2 C2J

Novosoft's C2J is similar to C2J++ in that it transliterates C source to Java, but it does not handle C++ [50]. It has been applied successfully to nontrivial programs and solves many of the problems that C2J++ struggles with. It also transliterates control flow features of C that are not supported by Java, such as comma expressions and goto statements. C2J comes with a large C run-time library providing most of the routines that typical C programs require.

Figure 4.4 shows a part of the transliteration generated by C2J. The transliterated source is much longer than the original. The logic of the program is difficult to understand, since many complex transformations have been applied to exactly emulate the behaviour of the original C source (the source code shown has already been simplified by removing dead-code and superfluous nesting).

C2J shares some of the disadvantages of the Java Back-End for GCC: Data structures are stored in a large array, thus circumventing Java's type checking and run-time security checks. The array access required for many operations and the sometimes necessary extra type conversions create a run-time over-

```
class sample {
  public int cfoo(int ci, int cs) {
    nextlevel();
    int label = 0; int retval = calloca(4);
    int ch_5 = 0; int y1 = 0; label = 0;
  break_while:
    switch(label) {
    case 0:
      label = -1;
    lab_sample0:
      while(true) {
        sincMEMINT((int)((cs + 0)),+1);
        if ((((getMEMINT((int)((ci-= 4))))>(0))?1:0)==0 )
          break lab_sample0;
        label=0;
        do {
        continue_while:
          switch (label) {
          case 0:
            label =- 1;
            ch_5  = (int) getMEMINT((int)(ci));
          lab_sample1:
            for ( ; (((ch_5)<(getMEMINT((int)((cs + 0)))))?1:0)!=0 ; ) {
              if ( (((((int)((ch_5)%(3))))==(0))?1:0)!=0) {
                label = 1;
                break continue_while;
              } else if ( (((((int)((ch_5)%(2))))==(0))?1:0)!=0) {
                  label = 2;
                  break break_while;
                }
              ch_5= (int)((int)((ch_5) + (getMEMINT((int)(ci)))));
            }
          case /*continue_while*/ 1:
            label = -1;
          }
        } while (label != -1);
      }
    case /*break_while*/ 2:
      y1 = (int)((cs + 0));
      setMEMINT((int)(y1),(int)(((int)((getMEMINT((int)(y1)))
                                    + (getMEMINT((int)(ci)))))));
      retval = ((int)cs);
      prevlevel();
      return retval;
    }
  }
  /* ... */
}
```

Figure 4.4: C2J transliteration result (excerpt only, hand optimised)

head. They also make the code difficult to read. In this sense, C2J provides only little advantage over the Java Back-End for GCC: debugging might be easier with the transliterated Java source code available, but on the other hand, some C control flow structures may be more efficiently implemented by compiling the C source code directly to machine language for the JVM.

4.5 Summary

In this chapter, we surveyed some of the C to Java source code migration approaches introduced in Chapter 3 in more detail. We evaluated Java Native Methods (JNI), C to byte code compilation, re-implementation strategies, and transliteration tools and assessed their suitability for the integration of C source code into Java. Our findings helped us to identify and prioritise goals for an improved migration approach, which we describe in the next chapter.

Chapter 5

Goals for an Improved Migration Approach

The survey presented in the previous chapter points out a number of deficiencies of current C to Java migration strategies that severely limit their usefulness for migration of mission critical business applications. Terekhov and Verhoef propose a number of requirements that have to be met to achieve a successful source conversion [63]:

- An inventory of all native and simulated constructs in the source language needs to be built.
- For every such construct, a conversion strategy to a native or simulated construct of the target language must be found. Some source constructs may be obsolete in the target language and thus have no target construct at all. The conversion strategies should be illustrated by source and target code fragments.
- It must be clarified, whether and to what extend the target system must be functionally equivalent to the source system. Should obvious errors in the system be corrected during the migration? Does the new system have to be compatible with the existing test cases?
- One of the goals of the migration process should be to achieve a maximum of automation.

- The new system should be maintainable. If developers familiar with the source system are to maintain the target system, then the new system should have a structure similar to that of the existing system, possibly by using emulated language constructs. If a new set of developers is to maintain the system, the new code should use as many native language constructs as possible.

- The efficiency and size of the new system must be acceptable.

- If the conversion tools are to be used more than a few times, their run-time efficiency is also of importance.

Some of these requirements pose questions that need to be answered before conversion strategies can be developed. For example, Malton postulates that source conversion should result in a system that uses native language constructs wherever possible; emulation of non-native language constructs has a negative impact on maintainability and should be avoided [43].

Though Malton's choices appear reasonable, these questions have to be answered for every individual conversion project, as requirements may vary. For a given project, some choices might not be compatible with others: it may not be possible to achieve total automation and maximum performance, and complete functional equivalence might not be possible with well readable and maintainable code.

As Ephedra is not aimed at a specific source conversion project but seeks to provide a generalised approach for migrating from C to Java, we had a little more freedom in making our choices. We prioritised them as follows:

Maintainability: The generated code has to be maintainable. As it is likely that developers with experience in Java will be maintaining the code, native source constructs should be used wherever possible. Emulation should only be used where a native source construct would decrease the maintainability of the code significantly.

Functional Equivalence: The generated code has to be mostly functionally equivalent to the original code. The conversion process has to document

all possible incompatibilities. Studies have shown that it is not generally cost effective to prove that a migrated software system is functionally equivalent to the original system, for example through the use of a wide spectrum language (WSL) [6]. A more cost effective and sensible approach is to define a set of acceptance tests that the migrated software system must pass, for example the current regression tests of the original system [57].

High Automation: We aim to achieve high automation, but rely on human developers to intervene where automated conversions would produce non-maintainable code. The automated tools will document these problems and guide developers in their solution.

Efficiency of Generated Code: The generated code should not be significantly slower than code written by a human developer.

Efficiency of Tools: The tools are intended to be used in a variety of projects, their performance should not be significantly worse than that of a compiler.

Terekhov and Verhoef mention in their requirements that a catalogue of native and simulated constructs in the source language and their mappings to the target language should be built. As the use of simulated constructs varies from project to project, it is impossible to provide a complete catalogue for Ephedra. We provide a partial catalogue of the most common native and simulated language constructs in Part II of this dissertation, along with a few references to related work on the conversion of other language constructs.

Part II

Ephedra

The previous chapters pointed out limitations and problems of the current approaches to integrating C source code into Java programs: C2J++ requires extensive manual work in the verification and correction of the transliterated code. JNI, C2J, and the Java back-end for GCC are susceptible to compromising Java's type safety and security, which can result in poor maintainability and possibly large performance overheads.

The goal of the Ephedra approach is to supply a better solution to the problem of integrating C source code into Java programs. It provides a structured approach to migrating C source code to the Java Virtual Machine, minimising manual intervention by the software engineer wherever possible and guiding him or her wherever full automation cannot be achieved. The resulting Java source code does not circumvent the safety features of the Java Virtual Machine and can be easily integrated with existing Java programs. While the emphasis of our approach is on the C language, Ephedra also supports the conversion of the most commonly used C++ language elements.

Chapter 6

Approach

6.1 Phases of Source Conversion

One of the fundamental concepts of problem solving in science also applies to software engineering: a big problem has to be broken up into many manageable small problems. This concept provided the rationale for many developments in software engineering. Procedures in procedural programming, units or modules in modular programming, and classes, namespaces, and packages in object-oriented programming all aim to reduce the complexity of the problems to be solved.

Source conversion, being a non-trivial problem, also has to be solved by providing a systematic, step-by-step approach. Terekhov and Verhoef present a three phase process for language conversion which has been refined and expressed more formally by Malton, whom we cite in the following paragraphs [43]:

> **Phases of Source Migration:**
>
> To keep the difficulties under control, there are three phases of source migration: [...]
>
> **Normalisation** is reducing the translation space. At the lexical level, some kind of source factoring is essential so that the code factor is available to design analysis. Normalisation is

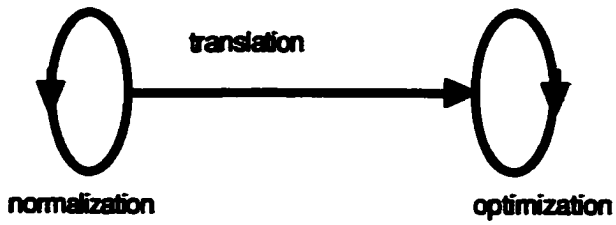

Figure 6.1: Malton's Migration Barbell

in fact directed dialect conversion, where the target dialect is chosen to make the translation step easier. It may involve API migration as well, if there are APIs which will not be available on the target environment.

Translation is actually the hardest step! The goal is to make a version which runs in the target environment and can be improved to a maintainable artifact by optimisation. The semantics should be preserved of all factors: this is difficult to define as a requirement. We have found that blind translation is the most effective approach, since there is very little support semantically for the intermediate stages.

Optimisation is similar to optimisation in compiler back-ends, except that the improvements intended are improvements to reflect the goals of migration, especially [to] make maintainable source. This will probably involve removing or simplifying APIs inherited from the source during normalisation, recognising and improving the idioms to conform to native style on the target side, and possibly improving the software architecture. This is also directed dialect conversion, with a different direction.

The above can be summarised by Figure 6.1 which because of its resemblance to a weight-lifting device we have taken to calling the Barbell Model. It reminds [us] that, to keep balanced the considerable weight of work required when migrating software source,

> re-engineering tasks should be kept on the ends (where they can be carried out in a single semantic framework or platform, with integration testing and software management processes in place) and the tricky task of keeping the ends together is kept narrow.

The Ephedra approach can also be expressed in three phases, with some of the phases divided into sub-phases. In our research, we focused our attention on the normalisation and translation phases while relying on existing work for the optimisation phase. For the distribution of the conversion tasks among these phases, we evaluated where they would be placed best.

The elimination of `goto` statements, for example, could be done in the normalisation phase. Since C/C++ lacks multi-level `break` and `continue` statements, the conversion of the `goto` statements would result in poor quality code. As `goto` statements do not exist in Java, they cannot be dealt with during the optimisation phase. So the translation phase was the obvious choice for this conversion.

We tried to keep the translation phase free from the need for user involvement. Thus, the elimination of type casts that are illegal in Java is done during the normalisation phase, because it needs the developer to make choices. As the conversion does not depend on language concepts only available in Java but not in C, there is no disadvantage in doing it during the normalisation phase.

There is good tool support for refactoring of Java source code. We therefore decided to delay most of the major structural changes to the optimisation phase, to take advantage of these existing tools.

6.2 Shift to Object-Orientation

Object-oriented programming, while invented in the 60's with Simula 67 [14], became popular in the late 80's and early 90's with the availability of programming languages such as Smalltalk [27], C++ [61], and Java [28]. Some programmers adopted object-oriented ideas while still working with non object-oriented languages. In fact, research continues on how to express object-

oriented features in procedural languages such as C [17]. While implementations of these features in C may not look as elegant as in C++, they are possible: the original C++ front-end *cfront*, written by Stroustrup, transliterates C++ code into C code that emulates many C++ features using C data structures and pointers and is subsequently compiled into object code [61].

While comparing C source code generated from C++ by *cfront* to C source code written by developers of procedural systems, we found similarities in the data and control structures used. Moreover, we could manually convert the procedural code to object-oriented code quite easily. From these experiments the following question arose: Is it possible to discover intentional or even unintentional object-oriented structures in C code using the inverses of some of the mappings used in the *cfront* conversion from C++ to C? If it were possible, this would help immensely in the migration of procedural legacy systems to object-oriented platforms.

In performing the paradigm shift from procedural to object-oriented code, we experimented with this question. In Section 7.2 we present a strategy for discovering implicit polymorphism in procedural C code and converting it to C++ using the explicit polymorphic language concepts of C++ during the normalisation phase of the conversion process.

Other problem areas in the conversion of procedural to object-oriented code have been well researched, and we presented an overview in Chapter 3. The majority of these problems can be dealt with either during normalisation or optimisation. As tool support is better for Java, we propose to handle them during the optimisation phase. This also supports the use of native Java language concepts rather than converted C++ language concepts.

6.3 Overview: A Three-Step Approach

Ephedra suggests a three-step approach for the conversion of C and C++ programs to Java, roughly oriented on Malton's Migration Barbell [43]. The following sections give an overview of the code transformations applied during a migration using Ephedra. Later chapters describe these steps in more detail.

6.3.1 Normalisation

The first normalisation step (Section 7.1) is necessary only during the conversion of K&R style C code. This kind of code does not contain function prototypes, thus limiting the ability of the compiler to perform type checking. To find and remedy major type inconsistencies that may even be the cause of problems in the current code, all necessary function prototypes are inserted in the code. An ANSI C or C++ compiler will be able to compile the code after this step.

Being a procedural language, C allows for data types to be explicitly related through composition only. Object-oriented languages also support inheritance relationships. In the second normalisation step (Section 7.2), data types and type conversions are analysed to find relationships that are implicitly present in the code and can be better and explicitly expressed using inheritance relationships. The code is then changed accordingly, and as a side effect, most Java incompatible type casts disappear from the code. This step will have to be performed on most C and many early C++ programs.

An obstacle in the conversion from C++ to Java is the use of C++ multiple inheritance which is only partially supported by specific language constructs in Java. Wen examined the use of multiple inheritance in C++ programs and identified different categories of multiple inheritance [68]. She also developed approaches to convert these different multiple inheritance types to Java using Java language constructs such as interfaces. In the normalisation phase, we also convert multiple inheritance to single inheritance. We do not present any new strategies for the conversion of multiple inheritance but refer to the thesis by Wen.

C++ adds to the C programming language not only by adding object-oriented language features such as classes and inheritance but also by providing a number of additional language constructs, in particular constants and references. By replacing some C macros by C++ constants and selected C pointers by C++ references, a functionally equivalent version of the source code can be produced whose Java translation is more maintainable and better in run-time performance (Section 7.3).

After any of these transformations have been performed, the code can be compiled with a C++ compiler and finally verified using existing test suites.

The Ephedra normalisation step does not strictly conform to Malton's Migration Barbell [43]: it takes advantage of the fact that C and C++ are very closely related programming languages and involves a *translation* of C source code to C++. As this translation simplifies the transformation tasks of the normalisation step and does not involve major syntactic changes, we view it as part of the normalisation step.

6.3.2 Translation

In the translation phase, the C/C++ source code is transliterated to Java source code. Data structures are converted to classes and functions are converted to methods and collected in classes. During the transliteration, the code is also analysed for invocations of C storage allocation library functions, which are converted to near equivalents in the Java language.

The Java source code that is the result of the translation phase can be compiled with any Java compiler and verified with the existing test suites.

6.3.3 Optimisation

The refactoring community has developed well-founded strategies for improving the architecture of object-oriented programs. Many of these strategies have been formalised in algorithms and implemented in software engineering tools. We did not develop any new techniques for the optimisation step but rely on existing research [23, 52].

6.4 Summary

In this chapter, we explained the overall approach Ephedra takes in the conversion of C source code to Java. We described Malton's *Phases of Source Migration* model and illustrated how Ephedra fits into this model by presenting an overview of the code transformations applied during each of these phases.

The following chapters describe the normalisation and translation phases in detail.

Chapter 7

Normalisation

7.1 Insertion of C Function Prototypes

There are two major styles of C source code: Kernighan and Ritchie first designed the C programming language in 1978, and source code written in that style is often called *K&R C* [33]. In 1989, many developments and extensions to K&R C were standardised by the American National Standards Institute [1]. Code following this standard is usually called *ANSI C.*

The most obvious difference, and for Ephedra the most significant difference between these styles, is the use of *function prototypes.* In K&R C, only variables have to be declared before their use, any unknown identifier that is followed by the function call operators is assumed to be the identifier of a function, and that function is assumed to take a variable number of arguments and return an integer value. It is left to the programmer to make sure that these assumptions are actually valid.

In ANSI C however, developers are encouraged to declare functions using function prototypes that specify the return type and the number and types of the arguments, if any. These function prototypes should be added to each source file before the first invocation of that function in the source file. Figures 7.1 and 7.2 show the differences between the two styles by means of a small example program.

As the consistency checking between function definitions and function in-

```
main(argc, argv)
  int argc;
  char *argv[]; {
  int argument;

  if (argc != 2) {
    printf("usage: %s value\n", argv[0]);
    exit(1);
  }

  argument = atoi(argv[1]);
  printf("the square of %d is %d\n", argument, square(argument));
  exit(0);
}

square(i)
  int i; {
  return i * i;
}
```

Figure 7.1: K & R style C program

```
#include <stdio.h>              /* declares printf library function         */
#include <stdlib.h>             /* declares atoi and exit library functions */

int square(int i);                      /* declares the square function     */

int main(int argc, char **argv)         /* declares the main function       */
{
  int argument;

  if (argc != 2) {
    printf("usage: %s value\n", argv[0]);
    exit(1);
  }

  argument = atoi(argv[1]);
  printf("the square of %d is %d\n", argument, square(argument));
  exit(0);
}

int square(int i)
{
  return i * i;
}
```

Figure 7.2: ANSI style C program

```
foo() {             // return type defaults to 'int'
}                   // but no value is actually returned

bar() {
  int i = foo();  // statement is valid, but result is undefined
  foo();
}
```

Figure 7.3: C program with possibly faulty return types

```
void foo() {        // states explicitly that there is no return value
}                   // and accordingly, no value is returned

void bar() {
  int i = foo();  // C compiler correctly flags this statement as error
  foo();          // valid statement
}
```

Figure 7.4: C program with corrected return types

vocations is left to the programmer in K&R C, there is a risk for introducing errors. To increase the chances for a successful migration from C to Java, it is important to eliminate sources of errors first. Therefore, as the first step in the Ephedra migration approach, a C program written in K&R style is converted to ANSI C style. The Free Software Foundation's GNU Compiler Collection (GCC [24]) comes with the tool *protoize*, which automates most of the conversion.

To improve the readability and correctness of the source code further, the return types of functions should be checked. If no return type is specified in a prototype of a function, a C compiler assumes it to be an integer. In many programs, this default formal return type is used even for functions that do not return a value. Modern C and C++ compilers can detect and flag these discrepancies and aid the software engineer in locating the code that needs change. Figures 7.3 and 7.4 show an annotated example.

7.2 Data Type and Type Cast Analysis

Object-oriented programming languages greatly facilitate reuse by providing generalisation and specialisation features, such as inheritance. While the publications mentioned in Chapter 3 show how to identify distinct classes within a system, they do not take advantage of generalisation or specialisation features provided by the target programming language. It seems worthwhile to examine how commonalities and relationships of structures in a legacy system can be exploited to design inheritance hierarchies in the target system.

Type casts between otherwise unrelated structures in a C program are often used to model generalisation or specialisation features in procedural code [44]. These type casts pose problems when trivially transliterated to Java since type casts between data types that are not related through inheritance are prohibited in Java. Even stricter type checking is performed at run-time to prevent objects from being cast to a class type they do not actually instantiate.

To migrate a C program to Java successfully, either such type casts have to be removed, or the data structures involved in the type casts have to be changed to make their implicit relationship explicit. Section 7.2.1 shows how problematic type casts and the data structures involved can be identified. Section 7.2.2 presents algorithms that can be used to automate this process. In Section 10.2, a tool is shown that implements these algorithms and also suggests changes to the data structures that will make the type casts admissible in a Java program.

7.2.1 Example Transformation

7.2.1.1 Sample Code

Figure 7.5 shows a simple example of a C program using distinct data structures and type casts to express specialisations of a data structure. The specialised data structures (`Manager` and `Worker`) contain all the fields of the base data structure (`Employee`) as well as additional fields. Whenever pointers to variables of type `Manager` or `Worker` are allocated, their *state variable* `employeeKind` is initialised to identify the variables' run-time type. The pro-

```
enum EmployeeType { WORKER, MANAGER };
struct Employee {
  int employeeKind;
  char name[20];
  char extension[4];
};
struct Manager {
  int employeeKind;
  char name[20];
  char extension[4];
  int numUnderlings;
  struct Employee* underlings;
};
struct Worker {
  int employeeKind;
  char name[20];
  char extension[4];
  struct Manager* manager;
};
void showEmployee(struct Employee *e) {
  printf("%s", e->name);
  switch (e->employeeKind) {
    case WORKER:
      {
        struct Worker* w = (struct Worker*) e;
        printf(" is managed by %s.\n",
               w->manager->name);
      } break;
    case MANAGER:
      {
        struct Manager* m = (struct Manager*) e;
        printf(" manages %d employees.\n",
               m->numUnderlings);
      } break;
  }
}
```

Figure 7.5: Example of related data structures and their use in a C program

gram accesses the variables using pointers of type `Employee` for general processing. For computations particular to the specialised data structures, the program determines the run-time type of the variable using the state variable, and casts it to a pointer of the specialised type (`Manager` or `Worker`).

A large number of similar techniques have been used to express specialisations and generalisations like this. In some cases, the specialised data structures contain the base data structure (rather than containing the same fields). Sometimes, data structures are padded with extra variables to ensure that all have the same size. In other cases, *unions* are used to group related data structures. Stroupstrup's *cfront* C++ front-end transliterates a class hierarchy into data structures similar to the ones in this example.

7.2.1.2 Identification of Inheritance

A typical approach to converting C to Java is to transform C structures into Java classes and C functions into static member methods of some Java class. If this approach is followed in the example, the resulting Java program will not compile: the type casts within the `showEmployee()` function are illegal in Java, since the structures involved in these type casts are not related.

A better solution for this migration problem needs to be found. The similarity of the data structures in the examples and the type casts between them suggest that they are closely related. Domain knowledge supports this observation: managers and workers *are* employees. This suggests that the target classes should be modelled in a way that explicitly states this relationship, namely by using inheritance.

Figure 7.6 shows a transformation of the example from Figure 7.5 using classes related to each other by inheritance. The `Employee` structure continues to contain the fields for the general data structure, but the `Manager` and `Worker` structures *inherit* these fields from `Employee` rather than redefining them. The code can now be transformed to Java by converting the C++ data structures to Java and wrapping the `showEmployee()` function with a class. The remaining type casts are legal in Java.

```
enum employeeType { WORKER, MANAGER };
struct Employee {
  int employeeKind;
  char name[20];
  char extension[4];
};
struct Manager : public Employee {
  int numUnderlings;
  struct Employee* underlings;
};
struct Worker : public Employee {
  struct Manager* manager;
};
void showEmployee(struct Employee *e) {
  printf("%s", e->name);
  switch (e->employeeKind) {
    case WORKER:
      {
        struct Worker* w = (struct Worker*) e;
        printf(" is managed by %s.\n",
               w->manager->name);
      } break;
    case MANAGER:
      {
        struct Manager* m = (struct Manager*) e;
        printf(" manages %d employees.\n",
               m->numUnderlings);
      } break;
  }
}
```

Figure 7.6: Transformation of code from Figure 7.5

7.2.2 Formalisation and Evaluation

7.2.2.1 Algorithm

The algorithm in Figure 7.7 formalises the transformation from Figure 7.5 to Figure 7.6. Abbreviations and primitive operations for artifacts in the subject system's source code used in this algorithm are explained in Table 7.1.

First, all type casts between non-primitive data structures in the program have to be analysed (Figure 7.7, Step 1). The algorithm iterates over all relevant type cast expressions and records the relations between source and target data structures.

As mentioned in Section 7.2.1.1, in some cases extra variables are used to pad all data structures to have the same size. Therefore, the algorithm checks which fields in the data structures are actually referenced by the program (Figure 7.7, Step 2). Besides such padding fields, the algorithm also identifies fields that are no longer used in the program, and can thereby help in the detection of errors in the code (if those fields really should be referenced) or the removal of dead code (if those fields are no longer needed).

Finally, the groups of related structures as identified by the analysis of type casts are transformed to class hierarchies, with the base classes containing the common fields, and the sub classes containing the specialised fields. Only those fields that are actually used in the program are included in the classes built (Figure 7.7, Step 3). For example, two fields can be considered common to two data structures, if they have the same type, name, and position in the data structures. Other definitions of common fields are possible: the fields' names could be phonetically or semantically compared, or compatible rather than identical field types could be considered as criteria for commonality.

7.2.2.2 Efficiency Considerations

When dealing with algorithms that are to be used on large software systems, it is important to consider the computational complexity of these algorithms to make sure they will perform adequately on the average software engineer's workstation. The three algorithms introduced are now examined from this

castSet	the set of all type cast expressions that involve only non-primitive types.
typeSet	the set of all non-primitive types defined.
fieldAccessSet	the set of all field access expressions (a field access expression in the algorithm refers to an expression that denotes a field of a structure, such as `e.name` or `e→name` in C).
ce.destType	the destination type of the cast expression *ce*.
ce.sourceType	the source type of the cast expression *ce*.
fe.field	the field involved in the field access expression *fe*.
t.fieldSet	the set of all fields of type *t*.
t.fieldReferenced[f]	true, if field *f* of *t* is ever referenced.
t.destTypeSet	a set of types that is to contain, after completion of the algorithm, all non-primitive types the type *t* is cast to.
t.sourceTypeSet	a set of types that is to contain, after completion of the algorithm, all non-primitive types the type *t* is cast from.
commonFields(S)	a function that returns the set of common fields of all types in set *S*.
relatedTypes(S)	a function that returns a set of types in set *S* related through type casts (as determined in steps 1 & 2 of the algorithm).
similarTypes(S, F)	a function that returns subsets of set *S* whose member types share fields other than the fields in set *F*. Every member of *S* is member of exactly one of the returned (possibly one-element) subsets of *S*.
createClass(F, b)	a function that creates a class containing the fields in set *F*, with base class *b*.
createClass(t, b)	a function that creates a class from type *t* (using the fields and name of the type), with base class *b*.

Table 7.1: Glossary of notations used in algorithm of Figure 7.7

```
/* Step 1: Analyse type casts */
for t in typeSet do
  t.destTypeSet := ∅
  t.sourceTypeSet := ∅
end for
for ce in castSet do
  add ce.destType to ce.sourceType.destTypeSet
  add ce.sourceType to ce.destType.sourceTypeSet
end for

/* Step 2: Check for use of fields within structures */
for t in typeSet do
  for f in t.fieldSet do
    t.fieldReferenced[f] := false
  end for
end for
for fe in fieldAccessSet do
  t.fieldReferenced[fe.field] := true
end for

/* Step 3: Build class hierarchy */
for s in relatedTypes(typeSet) do
  buildLevelOfClassHierarchy(s, NULL)
end for

proc: buildLevelOfClassHierarchy(S, base)
  F := commonFields(S)
  if ( ∃ t: t.fields = F ) then
    newBase := createClass(t, base)
  else
    newBase := createClass(F, base)
  end if
  for s in similarTypes(S, F) do
    buildLevelOfClassHierarchy(s, newBase)
  end for
```

Figure 7.7: Inheritance detection algorithm

perspective using the following definitions:

C_n complexity for algorithm n

n_{casts} number of type cast expressions between non-primitive types

n_{access} number of field access expressions

n_{types} number of structured data types

$n_{related}$ maximum number of types in a set of related structured types

Step 1 This algorithm examines every type cast expression in a program exactly once. For every type cast between non-primitive data types, a data type is added to two sets of data types. Assuming a complexity of $log\ n$ for the insertion of an element into a set leads to the following statement:

$$C_1 = O(n_{casts} \log n_{related})$$

Experience with legacy systems and examination of object-oriented class libraries suggest that the number of data types in a set of related data types is relatively small and does not rise significantly as the subject system grows. The logarithmic expression can therefore be replaced by a constant:

$$C_1 = O(kn_{casts}) = O(n_{casts})$$

Step 2 This algorithm sets a flag for every field access expression.

$$C_2 = O(n_{access})$$

Step 3 The algorithm iterates over all sets of related types, and subsets of these. Since every data structure is member of exactly one of those sets and subsets, the algorithm processes every data structure exactly once.

$$C_3 = O(n_{types})$$

By adding the sub-complexities, we can conclude that the overall complexity of the algorithms depends *linearly* on the size of the subject system:

$$\begin{aligned} C &= C_1 + C_2 + C_3 \\ &= O(n_{casts}) + O(n_{access}) + O(n_{types}) \\ &= O(n_{casts} + n_{access} + n_{types}) \\ &= O(\text{number of lines of code}) \end{aligned}$$

7.3 Use of C++ Language Features

7.3.1 Macros and Constants, and Inline Functions

Lacking a more appropriate and efficient way to define constants in a C program, C programmers have used preprocessor macros for this purpose. The use of macros in C source code has created difficulties for many software engineering tool developers: since macros are expanded in a separate step before the lexical and semantical analysis of the source code, they are not represented in abstract syntax trees. Many software engineering tools operate on abstract syntax trees or similar constructs and can therefore not handle C macros properly but provide views of their subject systems with macro substitutions already applied.

The Ephedra source code translation tool shares this disadvantage. When a constant is defined as a macro, the translation tool does not see the constant's name in the abstract syntax tree it processes. It only sees its value. We therefore recommend conversion of constant definitions that use macros to constant definitions that use C++ constant variables, as shown in Figure 7.8. For string constants, this conversion has the side effect that, in the converted code, only one copy of the string exists, and thus, memory is saved. Macro definitions that represent small functions can be replaced by C++ *inline functions*, possibly using C++ templates if the type of the parameters is unknown.

Original C Code	Improved C++ Code
`#define MAXIMUM 5`	`const int MAXIMUM = 5;`
`#define MESSAGE "Hello"`	`const char* MESSAGE = "Hello";`

Figure 7.8: Conversion of macros to constants

7.3.2 Pointers and References

C++ introduces the new language feature of *references.* References can be regarded as immutable pointers. Lacking a more appropriate way to pass variables by reference, pointers have been used for this purpose in C programs. In C++, references can be used to distinguish uses of pointers for call-by-reference parameters and other purposes and thus improve the maintainability of the code.

One of Ephedra's pointer mapping strategies is more efficient for C++ references than for C++ pointers. Besides obtaining more maintainable code, using references instead of pointers can also improve the performance of the migrated code. Figure 7.9 illustrates two uses of pointers in C, one of which can be transformed to use a C++ reference. To find such pointers in a function, the function body needs to be analysed to determine whether the value of the pointer is changed, passed as a parameter to another function, or assigned to some other pointer.

7.4 Summary

In this chapter, we introduced the normalisation step of the Ephedra approach. We discussed source code conversions performed on the original C code before its translation to Java. In particular, we explained the conversion of source code written in K&R style C to ANSI C, the analysis and utilisation of type dependencies in the code, and the improvement of legacy C code through the use of C++ language features. In the following two chapters, we describe the translation step of Ephedra. Before providing a complete catalogue of source conversions in Chapter 9, we present the transformation techniques related to the conversion of pointers in Chapter 8.

Original C Code

```
// call-by-reference parameter
// conversion possible
void incrementByFive(int *i) {
  *i += 5;
}

// not a call-by-reference
// conversion not possible
int product(int *factors, int n) {
  int result = *factors++;
  while (--n)
    result *= *factors++;
  return result;
}
```

Improved C++ Code

```
// call-by-reference parameter
// conversion possible
void incrementByFive(int& i) {
  i += 5;
}

// not a call-by-reference
// conversion not possible
int product(int *factors, int n) {
  int result = *factors++;
  while (--n)
    result *= *factors++;
  return result;
}
```

Figure 7.9: Conversion of pointers to references

Chapter 8

Translation — Mapping Strategies for Pointers

C2J [50] and the Java Back-End for GCC [67] embed all structures and variables in a large integer array. This makes pointer operations and arithmetic easy and intuitive, but accessing the data through mainstream Java programs becomes difficult, and the data can be easily corrupted if there is an error in the code. Ephedra takes a different approach and maps C structured data types to corresponding Java classes. Where necessary, pointer arithmetic is *emulated* through member functions of these Java classes. This is accomplished without circumventing Java's storage allocation and memory protection schemes. Errors in the code will likely result in Java run-time exceptions and can thus easily be identified.

We developed two independent strategies for mapping C pointers to Java references. Both methods emulate the semantics of C pointers in Java, including pointer arithmetic, untyped pointers, and non-fatal array bound overflows (see Section 8.2 for a definition of these terms). The first method is well suited for programs that create a small amount of large data structures, while the other one is best suited for applications that create large arrays of small data structures or fundamental types, such as scientific algorithms.

8.1 Problem Overview

Java distinguishes between primitive and reference types. Primitive types are the arithmetic and boolean types provided by Java. Reference types are all system-provided and user-defined classes. Variables of primitive types always contain these primitive values while variables of reference types contain a reference to an object of their class or a subclass of that class. Variables of primitive types can only be passed by value, variables of reference types can only be passed by reference.

To pass variables of primitive types by reference, the variables need to be wrapped into classes. Java provides a number of such wrapper classes, but the values of objects of these classes are immutable. Ephedra therefore provides its own wrapper classes that are mutable. All variables of primitive types in the program that are passed by reference or that have their address taken need to be converted to variables of the corresponding wrapper types. Accesses and initialisations of these variables need to be transformed so that the wrappers' fields are accessed or initialised, respectively.

8.2 Definition of Terms

To discuss our approaches to mapping C pointer to Java references, we need to define a few terms related to pointers.

untyped pointers: Pointers are usually typed, i.e., their declaration includes an indication about which kind of variable they can point to. *Untyped pointers* are used where pointers can point to various different kinds of variables. They are frequently used in the declarations of standard library functions, such as functions for accessing permanent storage devices.

pointer arithmetic: As mentioned in Section 8.3, additive operators can be used to manipulate the address stored in a pointer, i.e., to change the storage destination the pointer is referring to. The use of these additive operators on pointers is called *pointer arithmetic*.

```
int *pi = (int*) malloc(5 * sizeof(int));
for (h = 0; h < 5; h++, pi++)
  ... ;
```

Figure 8.1: Example of non-fatal array bound overflow

non-fatal array bound overflows: When performing array traversals using pointers as described in Section 8.3, it is often necessary to compute storage addresses that do not actually refer to storage that has been allocated to a particular variable, or even the application. Theoretically, an array bound overflow condition is given. However, as long as the storage is not actually accessed, the overflow condition must not raise an exception, because it is *non-fatal.* Figure 8.1 illustrates this scenario. Before the loop is terminated, the pointer `pi` will point to a storage location outside the allocated storage area. But since the storage is not actually accessed, this is not an error.

For a C programmer, this scenario may appear trivial. As we will see, it requires special consideration when mapped to Java.

8.3 Kinds of Pointers

Since pointers, references, and arrays describe similar artifacts in a C program, we discuss them together in one chapter. We first describe their characteristics and differences:

Pointer variables hold the address of a storage area. Additive operators can be used to modify this address. When an additive operation is performed, the address in the pointer is incremented or decremented by a multiple of the size of the pointer's base type. The difference between two pointers of the same type is defined as the difference between the addresses they point to divided by the size of their base type.

Reference variables also hold the address of a storage area. However, these addresses have to be specified with the definition of the reference and are immutable after this definition. C++ uses slightly different syntax for referring to pointers and references.

Arrays allocate an area of storage to hold the specified number of values of their base type. When array variables are used without the subscript operators, they yield the address of that storage area and can be used like a pointer. However, like references, they are immutable. C supports multi-dimensional arrays. They are treated like one-dimensional arrays with a size equal to the product of the dimensions of the sub-arrays.

The subscript operators are defined for pointers and arrays as follows:

$$E1[E2] = *((E1) + (E2))$$

where $E1$ refers to a pointer or an array variable and $E2$ is an expression yielding an integral value. The addition operator functions the way described for pointer variables above.

Because of the described similarities of these slightly different concepts, we can regard references and arrays as special cases of pointers for our conversion problems: references are immutable pointers, arrays are immutable pointers whose definition also allocates storage for a certain number of objects. C compilers assure that references and arrays are not modified in the program code, we therefore do not have to take measures to guarantee the immutability of the converted references and arrays.

8.4 Classification of Pointer Uses

By investigating C programs, one can find several common uses of pointers:

access by reference: In C programs, the address of any variable may be taken to provide some other part of the program with a handle to some storage area. This is frequently done for function calls in C programs. For a subroutine to be able to modify storage that is under the control of

the calling routine, the calling routine passes the address of that storage (*call by reference*) rather than a copy of the storage area (*call by value*).

array traversals: Because of the aforementioned similarity of pointers and arrays, C programmers often use pointers to elements of an array to traverse these arrays, namely by incrementing the pointers. This is usually more efficient than a traditional array traversal using array indices since the address of an element does not have to be computed for every access to the element but only when the pointer is incremented.

dynamic memory: Pointers in C are used to access dynamically allocated memory.

polymorphism: As discussed in Section 7.2, C has limited built-in support for polymorphism, and some programmers have used pointers to achieve a similar effect in their programs.

communication with operating system libraries: many operating system routines, for example those for disk storage access, expect to be provided with the address of application storage to read or write data. C programs frequently use untyped pointers to pass these addresses to the operating system routines.

violation or circumvention of type conversion rules: some programs take advantage of the internal representation of certain types to achieve their purpose. For example, a program could determine or modify the exponent part of a floating point number by taking the address of that number and evaluating the storage at that address. The way floating point numbers are stored in application storage usually differs among different hardware platforms, so programs that use these techniques are not very portable. The use of pointers for these purposes is not recommended. Modern optimising C++ compilers may generate code with unexpected behaviour if these techniques are used.

8.5 Approach 1: Mapping using Augmented References

8.5.1 Basic Idea

In this approach, the Java classes that were derived from C data structures assume a number of additional properties to perform not only the functions of the original C data structures but also of pointers to these data structures. Pointers in the C source code are replaced by references to the new Java classes. We talk of augmented references since the additional properties of the generated Java classes allow us to use Java references to emulate the behaviour of C pointers.

Ephedra defines an abstract `Pointer` class, that provides functions emulating C pointer operations for classes that are not used within arrays. This pointer class also serves as the replacement of C's untyped pointer type `void*`. Another abstract class, `Array`, extends the `Pointer` class to emulate C pointer operations for classes that occur as array elements within the original C code.

As primitive types in Java are not classes, Ephedra defines wrapper classes for these types to provide a consistent approach. These classes contain a field called `value` of the primitive type they represent. Accesses to the value of variables converted from primitive types to these wrapper types have to be changed to access the `value` field of the converted variables. Initialisations of these variables (see also Section 9.7.5) do not only initialise their value but also allocate storage. The wrapper classes extend Ephedra's `Array` class to allow for primitive types to be members of arrays.

The special wrapper class `MultiPointer` wraps a variable of type `Pointer`. All pointers to pointer types in the original C source code are converted to this class.

This mapping approach is similar to Demaine's *block model* [16]. In contrast to Demaine's block model, our approach supports pointer arithmetic, which is a hard requirement for most real-world C programs.

8.5.2 Classes not Used within Arrays

Figure 8.2 shows an example of the use of pointers in C and a conversion to Java. The Java code contains excerpts of Ephedra's predefined `Pointer` class and the integer wrapper class `EInt`. Comparisons between pointers in C are converted to comparisons between references in Java, as shown in the `opPointerEquals()` method of `Pointer`. Classes that never occur as members of arrays in the original C source code, cannot possibly be subjected to pointer arithmetic. A call to the `opPointerAdd()` method of `Pointer` therefore raises an exception.

8.5.3 Classes Used within Arrays

The Ephedra `Array` class defines two fields: a reference to a Java array of `Arrays` and an index into that array. Every class that is used as element type of an array (either explicitly, or by dynamic storage allocation `new[]` or `malloc()` in the C source) is converted to be a subclass of this `Array` class. The two fields are initialised depending on how much storage is allocated for the variable in the original C source:

allocation of storage for one object: The converted code allocates storage for one Java object. The fields this object inherits from `Array` are initialised to zero and null, respectively.

allocation of storage for multiple objects: The converted code allocates storage for an array large enough to hold references to all objects as well as storage for the objects. The array elements are initialised to reference these objects. The fields these objects inherit from `Array` are initialised to contain a reference to the array and the index of the object within the array.

Using the fields defined by the `Array` class, pointer arithmetic can be emulated in Java code. Figure 8.3 illustrates the conversion in a slightly simplified way. Note that the Java code raises a run-time exception if the program tries to determine the difference between pointer variables that reference distinct

Original C++ Code

```
class A { };
void foo(int* i, A* a) { if ((void*) i == (void*) a) a++; }
...
  int i;
  A a;
  i = 5;
  foo(&i, &a);
```

Transformed Code

```
abstract class Pointer {
  public Pointer opPointerAdd(int relIndex) {
    if (relIndex != 0)
      throw new java.lang.ArrayIndexOutOfBoundsException
        ("non-zero array index for single object array");
    return this;
  }
  public int opPointerDiff(Pointer other) {
    if (this != other)
      throw new java.lang.RuntimeException
        ("attempt to get distance between unrelated pointer objects");
    return 0;
  }
  public boolean opPointerEquals(Pointer other) { return this == other; }
  public final boolean opPointerGreater(Pointer other) {
    return opPointerDiff(other) > 0;
  }
  public final boolean opPointerLess(Pointer other) {
    return opPointerDiff(other) < 0;
  }
}
final class EInt extends Pointer {
  public int value;
  public EInt() { }
  public EInt(int value) { this.value = value; }
  protected Array operatorASSIGN(Pointer src) {
    this.value = ((EInt) src).value;
    return this;
  }
}
class A extends Pointer { };
void foo(EInt i, A a) { if (i.opPointerEquals(a)) a.opPointerAdd(1); }
...
  EInt i = new EInt();
  A a = new A();
  i.value = 5;
  foo(i, a);
```

Figure 8.2: Conversion of pointers

Original C++ Code

```
class A { } ;
...
A *x;
A *y;

// increment pointer
x++;

// compare pointers
int i = x - y;
```

Transformed Code

```
class Array {
  Array[] containingArray;
  int indexWithinArray;
  ...
}

class A extends Array { }
...
A x;
A y;

// increment pointer
if (x.indexWithinArray >= x.containingArray.length)
  // handle boundary condition
else
  x = x.containingArray[x.indexWithinArray+1];

// compare pointers
if (x.containingArray != y.containingArray)
  // throw run-time exception
int i = x.indexWithinArray - y.indexWithinArray;
```

Figure 8.3: Conversion of pointers — pointer arithmetic

arrays. This is not the behaviour of the original C code where the result is undefined. Code that computes such a difference is likely to contain an error and, thus, Ephedra aborts the computation.

Since the Java transformations of the code frequently consist of several statements, and thus cannot be used within expressions, the code that appears in the example is put into methods of the **Array** class and replaced by calls to these methods. This also improves the readability of the converted code and keeps its size closer to the size of the original code. The member methods also handle boundary conditions: for example, a pointer in C may point to a memory location just outside of the storage area allocated to an array (non-fatal array bound overflow). As long as the storage is not accessed, this does not constitute an error. The methods of the **Array** class create auxiliary objects to handle these boundary conditions when necessary.

Since classes and pointers form a unit in this mapping approach, pointers share the type hierarchy with their associated classes. Type casts between pointer types whose base types are related through inheritance in the original C++ code are mapped to type casts between Java classes that are related through inheritance. Therefore they are legal and correctly mapped to Java. Type casts between typed and untyped pointers also translate correctly, since Ephedra's `Pointer` class is both superclass of all other classes involved in such type casts and the translation of C's untyped pointer type `void*`.

8.5.4 Efficiency Considerations

While this first mapping scheme produces well readable code for pointers and references to data structures, its performance did not satisfy our expectations for CPU intensive algorithms involving large arrays of fundamental data types. It requires one object to be created for every array element, even if the array is an array of fundamental types. There is a significant run-time overhead in the creation of objects, and accessing members of objects is significantly slower than accessing primitive Java variables. This problem prompted us to develop a different approach to translating C pointers that is more suited for applications that use these kinds of arrays.

8.6 Approach 2: Mapping using Inner Classes

8.6.1 Basic Idea

In this alternative mapping scheme, a unique Java class is created for every pointer type used in the original C program. The class is implemented as an inner class of the class representing the C type the pointer refers to. This approach is similar to Demaine's *improved conversion* [16]. In contrast to that conversion strategy, our approach supports untyped pointers that are needed in almost every program that uses the C standard I/O library.

8.6.2 Arrays and References

Java arrays and references are powerful enough to emulate C arrays and references using this approach. Therefore, no special functionality has to be implemented in the transformed C data structures for C code that only uses arrays and references but not pointers. The wrapper classes for primitive data types need some extra methods to allow for array elements to be passed by reference, as illustrated in Figure 8.4. As in the first approach, where the transformation of simple C expressions require several lines of Java code, the code is wrapped into member functions of appropriate classes to improve the readability of the transformed source code. The conversion benefits from earlier transformations of the source code that replaced certain pointers by references (see Section 7.3.2).

8.6.3 Pointers

When a pointer to a data structure is declared somewhere in the C source code, Ephedra defines an inner class in the corresponding Java class that is responsible for emulating the pointer's semantics. The class defines two fields containing a reference to an array of class elements and an index into that array. It also defines a number of member functions for manipulating these fields. Figure 8.5 shows the `ppInt` class, an inner class of the previously shown `pInt` class that represents the C `int**` type. All other classes representing

Original C++ Code

```
class A { };

void foo(int& i, A& a) {
  i = 10;
}

...

  int i[5];
  A a[3];
  i[3] = 5;
  foo(i[2], a[1]);
```

Transformed Code

```
final class pInt {
  public int[] value;
  public int index;

  public final pInt set(int[] src) {
    value = src;
    index = 0;
    return this;
  }
  public final pInt preIncr(int offset) {
    index += offset;
    return this;
  }
}

class A { };

void foo(pInt i, A a) {
  i.value[i.index] = 10;
}

...

  int[] i = new int[5];
  A a = new A[3];
  for (int temp = 0; temp < 3; temp++)
    a[temp] = new A();
  i[3] = 5;
  foo(new pInt().set(i).preIncr(2), a[1]);
```

Figure 8.4: Conversion of pointers — references

pointers are defined with an equivalent interface.

If casts between different pointer types are present in the original C code, the participating pointer classes need to extend Ephedra's abstract `Pointer` class, which is shown in Figure 8.6 along with the appropriate extension of the `ppInt` class. The `set()` method is invoked whenever a type cast between pointer types need be performed.

8.6.4 Efficiency Considerations

This second approach to map pointers to references performs much better for large arrays of primitive types than the first approach, as we could verify in one of our case studies (Section 11.4). There are still some possible performance bottlenecks though. A close look at the `postIncr()` method of the `ppInt` class shows that a new `ppInt` object is created for every post-increment operation. The creation of objects is a costly operation in most Java Virtual Machines and can therefore lead to performance degradation. In many programs, post-increment expressions occur isolated, i.e., they are used only because of their side effect (incrementing a counter) and their result (the previous value of the counter) is discarded. Such post-increment expressions can be converted to pre-increment expressions to improve the performance of the transliterated Java program.

8.7 Untyped Pointers and Serialisation

C's untyped pointer type `void*` is used by many C standard library routines to pass blocks of storage in parameters. Reading or writing to permanent storage devices is a good example. Ephedra implements serialisation methods for all data types that are directly (as parameters) or indirectly (as fields of directly or indirectly involved classes) involved in calls to C functions requiring `void*` parameters. For this purpose, Ephedra defines the `Pointable` interface (Figure 8.7). It provides methods for determining the size of data types in bytes and for reading and writing the contents of a data type to byte arrays that can be passed to Java library functions dealing with disk access and similar operating

```
public final class pInt {
  ...
  public static final class ppInt {
    public pInt[] value;
    public int index;

    public ppInt() {
      value = null;
      index = 0;
    }
    public final pInt get(int offset) {
      return value[index + offset];
    }
    public final ppInt set(pInt src) {
      value = new pInt[1];
      value[index = 0] = src;
      return this;
    }
    public final ppInt set(pInt[] src) {
      value = src;
      index = 0;
      return this;
    }
    public final ppInt set(ppInt src) {
      value = src.value;
      index = src.index;
      return this;
    }
    public final ppInt reset() {
      value = null;
      index = 0;
      return this;
    }
    public final ppInt preIncr(int offset) {
      index += offset;
      return this;
    }
    public final ppInt postIncr(int offset) {
      ppInt temp = new ppInt().set(this);
      index += offset;
      return temp;
    }
  }
}
```

Figure 8.5: The `int**` equivalent — Ephedra's `ppInt` class

```
public abstract class Pointer {
  public abstract int getIndex();
  public abstract Object getValues();
  public abstract Pointer set(Pointer src);
}

public final class pInt {
  ...
  public static final class ppInt {
    ...
    public final int getIndex() {
      return index;
    }
    public final Object getValues() {
      return value;
    }
    public final Pointer set(Pointer src) {
      value = (pInt[]) src.getValues();
      index = src.getIndex();
      return this;
    }
  }
}
```

Figure 8.6: Ephedra's `Pointer` class

```
public interface Pointable {
  public abstract int getSizeInBytes();
  public abstract int toBytes(byte[] bytes, int startIndex);
  public abstract int fromBytes(byte[] bytes, int startIndex);
}
```

Figure 8.7: Ephedra's `Pointable` interface

system functionality.

Figures 8.8 and 8.9 illustrate how classes implement the `Pointable` interface. For primitive data types, the appropriate wrapper classes assume the responsibility of implementing the serialisation methods. To perform a serialisation using these methods, the library functions have to determine whether a variable is of primitive or reference type and then invoke the appropriate serialisation routine of the wrapper class (for primitive types) or the Java class corresponding to the original C data structure.

8.8 C-Style Storage Allocation and De-Allocation

C, as opposed to C++, does not provide `new` and `delete` operators. Instead, standard library functions, such as `malloc()` and `free()` have to be used to allocate and de-allocate storage. Rather than converting calls to these library functions, Ephedra transforms the calls into calls to appropriate C++ operators and converts the C++ operators as previously shown. Given a fundamental or structured data type `X`, the expression `(X*) malloc(n)` is converted to `new X[n / X.sizeInBytes]` followed by a loop to allocate storage for every array element. The `new` expression and the loop statement are wrapped into a method to improve the readability of the code. If `n` is not a multiple of `X.sizeInBytes`, the original code is possibly incorrect. The Ephedra conversion shown above rounds the size of the memory block to be allocated down. The Java Virtual Machine will raise an exception if memory outside this block is accessed and in this way assist the developer in discovering the error.

Original C++ Code

```
class X {
public:
  int x;
};

class Y : public X {
  int y;
};
```

Transformed Code

```
class X implements Pointable {
  public int x;

  public static final int sizeInBytes = pInt.sizeInPrimitiveBytes;
  public static int readBytes(X into, byte[] bytes, int startIndex) {
    into.x = pInt.readPrimitiveBytes(bytes, startIndex);
    startIndex += pInt.sizeInPrimitiveBytes;
    return startIndex;
  }
  public static int writeBytes(X from, byte[] bytes, int startIndex) {
    startIndex = pInt.writePrimitiveBytes(from.x, bytes, startIndex);
    return startIndex;
  }

  public int getSizeInBytes() {
    return sizeInBytes;
  }
  public int fromBytes(byte[] bytes, int startIndex) {
    return readBytes(this, bytes, startIndex);
  }
  public int toBytes(byte[] bytes, int startIndex) {
    return writeBytes(this, bytes, startIndex);
  }
}
```

Figure 8.8: Serialisation of variables — Part 1 of 2

Original C++ Code

```
class X {
public:
  int x;
};

class Y : public X {
  int y;
};
```

Transformed Code

```
class X implements Pointable {
  ...
}

class Y extends X implements Pointable {
  private int y;
  public static final int sizeInBytes =
    X.sizeInBytes + pInt.sizeInPrimitiveBytes;
  public static int readBytes(Y into, byte[] bytes, int startIndex) {
    startIndex = X.readBytes(into, bytes, startIndex);
    into.y = pInt.readPrimitiveBytes(bytes, startIndex);
    startIndex += pInt.sizeInPrimitiveBytes;
    return startIndex;
  }
  public static int writeBytes(Y from, byte[] bytes, int startIndex) {
    startIndex = X.writeBytes(from, bytes, startIndex);
    startIndex = pInt.writePrimitiveBytes(from.y, bytes, startIndex);
    return startIndex;
  }

  public int getSizeInBytes() {
    return sizeInBytes;
  }
  public int fromBytes(byte[] bytes, int startIndex) {
    return readBytes(this, bytes, startIndex);
  }
  public int toBytes(byte[] bytes, int startIndex) {
    return writeBytes(this, bytes, startIndex);
  }
}
```

Figure 8.9: Serialisation of variables — Part 2 of 2

8.9 Summary

In this chapter, we discussed *pointers* as one particularly difficult problem in the conversion of C source code to Java. After giving a general overview of the problem and defining some terms important for the understanding of pointers in C, we explained the different kinds of pointers supported by the C and C++ programming languages. We presented a classification of pointer uses in C programs and described two methods for transforming code involving pointers to Java. The first method is most suitable for programs using pointers to data structures, while the second is more efficient for programs involving large arrays of fundamental data types. We concluded the chapter by discussing the transformation of serialisation and storage allocation. They are realised through pointers in C, and their translation therefore depends on the transformation of pointers. The next chapter illustrates the transformations Ephedra uses to convert the remaining C language constructs to Java.

Chapter 9

Translation — Detailed Catalogue

In the translation step of the Ephedra approach, the C source code that was improved and corrected in the normalisation steps is transliterated to Java source code. As one of our primary goals is the integration of the generated code into mainstream Java programs, readability of the generated code and compliance with the Java object model are critical issues.

Since C and Java are quite similar, some parts of the original program can be transformed trivially, i.e., by changing the syntax only. More complicated transformations are needed for the conversion of structured type declarations, initialisers, and expressions involving pointers.

The following sections present a catalogue of the source code transformations performed by Ephedra to convert the C source code to Java source code. In its organisation, the catalogue follows the *Reference Manual* of the C++ language [61]. For each paragraph in the reference manual, we show the transformation of the C++ source artifact defined in that paragraph to corresponding Java code. The catalogue explains for which parts of the C++ language Ephedra provides transformations and lists those parts of the C++ language that are not currently handled by Ephedra. The heading of each section provides a reference to the paragraph of the C++ Reference Manual that describes the particular C++ source artifact. Some paragraphs are skipped

since they are discussed in the context of a related paragraph. Some transformations are illustrated using code examples. The code examples do not show complete code but only code excerpts that illustrate the code transformations. Unless they are an essential part of the transformations, access specifiers and enclosing classes or methods are usually omitted. Where pointers, references, or arrays occur in the code examples, the second approach to mapping C pointers to Java is used. Section 10.3 describes the Ephedra tool that automates these transformation to generate a valid Java program.

Where appropriate, we mention how other transliteration approaches, in particular C2J [50] and C2J++ [35, 65], handle specific conversion tasks.

9.1 Lexical Conventions (§r.2)

C programs consist of one or more *files* that are translated in several phases. Declarations and definitions can be spread over several of these files. In the first phase of translation, the preprocessor performs file inclusion and macro substitution to generate a sequence of tokens, a so called *translation unit.* A translation unit contains the complete declarations and possibly also definitions of one or more artifacts.

Java programs also consist of one or more files, but they also constitute the translation units. Since no preprocessing is performed, each source file must contain only complete declarations with their definitions. Classes and interfaces are the only top-level declarations allowed, all other declarations need to be members of a class or an interface.

To convert a C program to Java, one or more C translation units need to be gathered. All top-level declarations that are not classes need to be wrapped into classes. Finally, the classes need to be written out to files. A widely accepted convention among Java programmers is to put only one public class and any non-public classes associated with it into each source file, and to name each source file after the public class. Ephedra follows this convention. Since all classes in C++ are public, the Java code resulting from the transformations will only contain public classes. Hence, every Java source file contains exactly one class which is public (definitions of inner classes, regardless whether public

or not, are part of the definitions of their enclosing classes in both C++ and Java).

9.1.1 Tokens, Comments, Identifiers (§§r.2.1 – r.2.3)

The lexical structure of C and Java source files is very similar. Both consist of a stream of tokens: identifiers, keywords, literals, operators, and separators. Comments may appear between tokens and have the same format in both C and Java. The set of possible C identifiers is a subset of the set of possible Java identifiers, so no transformation is necessary (unless a C identifier is also a Java keyword, see below).

9.1.2 Keywords and Operators (§r.2.4)

C and Java keywords are similar, but Java defines a few keywords that are not present in C. If a C program uses an identifier that conflicts with a keyword in Java, the identifier has to be changed. Ephedra adds an underline character to every such identifier. If this creates a conflict with some other identifier, further modifications are made. The Java keywords that are valid for identifiers in C are:

`abstract`	`boolean`	`byte`	`extends`	`final`
`finally`	`implements`	`import`	`instanceof`	`interface`
`native`	`null`	`package`	`strictfp`	`super`
`synchronized`	`throws`	`transient`		

There are only minor differences in the operators supported by C and Java. They are explained in the sections on expression transformations.

9.1.3 Literals (§§r.2.5)

There are only minor differences in the representation of literals in C and Java source files. For integer constants, C allows specification of the precision and signedness. Java allows the specification of precision only. In the conversion, this difference causes problems only for unsigned integer constants that are

larger than the largest Java integer constant. Ephedra does not propose a general solution for this problem, but suggests that the developers inspect the code in question.

Character and string literals are specified similarly. Java does not allow hexadecimal escapes and restricts octal escapes, so the literals need to be checked for Java compliance and possibly rewritten. C++ wide-characters can be transformed to Java unicode characters.

9.2 Basic Concepts (§r.3)

9.2.1 Declarations and Definitions (§r.3.1)

C differentiates between *declarations* and *definitions*. Declarations introduce names into a program. Definitions provide implementation details of the named artifact. Artifacts can be declared several times in a program but only defined once. A C compiler gathers and matches all declarations and definitions for a source artifact to generate code. In Java, the declaration has to be in the same place as the definition of a source artifact, except in an *import* or *package* declaration. During the conversion, the declarations and definition have to be gathered, matched, and written out as one Java declaration as shown in Figure 9.1.

9.2.2 Scopes (§r.3.2)

Scoping rules are mostly identical in C and Java. In C, enclosed blocks of code can declare a name that has already been declared in an enclosing block of a function, thus hiding the declaration in the enclosing block. This is not allowed in Java. The declarations in the enclosing block will have to be renamed. C2J++ [35, 65] does not resolve naming conflicts that arise out of the different scoping rules of C and Java. C2J [50] does resolve these conflicts, in fact it renames all variables during the transliteration, and thus makes it difficult for developers who know the original code to recognise the code.

C defines a *file scope*. A name declared outside all blocks and classes has

Original C++ Code

```
// point.h
class Point {
  int x, y;
public:
  int getX();
  int getY();
};

// point.cc
#include "point.h"

int Point::getX() { ... }
int Point::getY() { ... }
```

Transformed Code

```
// Point.java
class Point {
  private int x;
  private int y;

  public int getX() { ... }
  public int getY() { ... }
}
```

Figure 9.1: Transformation of declarations and definitions

file scope. The closest equivalent in Java is the *package scope*. However, only class and interface declarations are allowed in Java's package scope. Therefore, Ephedra wraps a non-class declaration in file scope into a class. The class is placed in a package whose name is derived from the name of the namespace in which the declaration was defined. If the declaration was defined in the global namespace, Ephedra creates an auxiliary name for the package. C2J++ [35, 65] does not handle namespaces, and wraps all top-level non-class declarations in the first class declaration it encounters in the source code. C2J [50] creates a wrapper class for every compilation unit.

9.2.3 Start and Termination (§r.3.4)

A C program must contain a function called `main()`. The function is the designated start of the program. It is usually passed two parameters specifying the number and contents of the command line arguments passed to the program.

A Java class may contain a function called `main()`. It is passed an array of Strings as the only parameter. A Java Virtual Machine can be instructed to start a Java program with the `main()` function of a particular class and to pass its command line arguments to it.

Ephedra creates a Java `main()` function that transforms the array of Strings

Original C++ Code

```
// foo.c


// method defined in
// top-level scope
int foo() {
  int i;
  ...

  // new definition of i
  // hides previous one
  for (int i1 = 0; ...; ...)
    ...
}
```

Transformed Code

```
// foo.java
package foo_package;

// method wrapped into class
class foo {
  public int foo() {
    int i;
    ...

    // variable renamed to
    // avoid naming conflict
    for (int i = 0; ...; ...)
      ...
  }
}
```

Figure 9.2: Transformation of scoping and naming conflicts

to a data structure acceptable to the transformed C `main()` function (according to the transformation rules for functions) and subsequently transfers control to that function. This is similar to the technique used by C2J [50]. C2J++ [35, 65] does not handle the conversion of the `main()` function.

In C and Java, programs are terminated by calling the standard library `exit()` function. In C, there is an `abort()` function that terminates a program abnormally. There is no equivalent in Java, so the `System.exit()` function has to be used instead. Another possibility would be to throw a special exception.

9.2.4 Storage Classes (§r.3.5)

Both C and Java support local variables of the *automatic* storage class. Automatic variables are initialised each time the control flow reaches their definition and destroyed on exit from their block.

In C, local variables of a function may be declared with the *static* storage class. As opposed to automatic variables, they are initialised only once during the run-time of the program and retain their value between invocations of the function. Local variables in a Java method can only be automatic. Ephedra transforms all static variables of a function to static variables of its enclosing

class. They are named in a unique way to avoid conflicts with other variables. Section 9.7.1.1 describes the transformation in more detail.

9.3 Types (§r.3.6)

9.3.1 Fundamental Types (§r.3.6.1)

Both C and Java provide the `void` type that specifies an empty set of values. No transformation is necessary.

C defines a number of integral types: `char`, `short`, `int`, and `long`. Depending on the C compiler's implementation, variables of type `char` can be signed or unsigned. Variables of the other integral types are signed. The signedness of an integral variable can be changed in its declaration by prefixing the type name with the keyword `signed` or `unsigned`. The `char` type is large enough to store any member of the compiler implementation's basic character set; its size is one byte (§r.5.3.2). The sizes of the other integral types are implementation dependent; the `short` type is guaranteed to be no larger than the `int` type, and the `long` type is guaranteed to be no smaller than the `int` type. C provides the `float` and the larger or identical `double` data types for floating point arithmetic. Some C++ implementations define the boolean type `bool`, a wide character type `wchar_t` and another integral type `long long` which is larger than `long`, and a third floating point type `long double` which is no smaller than `double`.

Java defines the following integral types: `byte`, `char`, `short`, `int`, `long`. All of them except for `char` are signed, and there are no equivalents with opposite signedness. The `char` integral type is not intended for arithmetic operations but for storage of unicode characters. The sizes of the types are exactly defined: variables of type `byte` can hold values from -128 to 127, `char` from 0 to 65535, `short` from -32768 to 32767, `int` from -2147483648 to 2147483647, and `long` from -9223372036854775808 to 9223372036854775807 (all inclusive). Java provides single precision 32 bit and double precision 64 bit floating point types whose values and operations conform to the IEEE 754 standard.

It is difficult to choose a good transformation for the integral data types, in particular since the characteristics of many of them are implementation dependent for C. By transforming all unsigned C data types to Java's `char` type and all signed C data types to Java's `short` type one could achieve a transformation that conforms to the definition of the C language. However, many current C programs depend on the C `int` data type to be larger than Java's `short` data type and would therefore fail after the conversion. Also, Java discourages the use of the `char` data type for arithmetic and favours the use of the signed integral data types.

C Data Type	Java Data Type
void	void
wchar_t	char
bool	bool
signed char	byte
unsigned char	byte
signed short	short
unsigned short	short
signed int	int
unsigned int	int
signed long	int
unsigned long	int
signed long long	long
unsigned long long	long
float	float
double	double
long double	double

Table 9.1: Transformation of fundamental data types in Ephedra

We therefore chose to transform the integral data types as displayed in Table 9.1. All integral C data types (except for `wchar_t`) are transformed to signed Java data types to achieve maximum conformance with Java coding standards. In automated transformations, the developers are informed in

which places transformations from unsigned to signed data types were performed so they can check whether the code depends on boundary conditions that are no longer met. The developers will usually need thorough knowledge of the code to make a proper assessment.

We decided not to transform C character arrays (strings) to the Java `String` or `StringBuffer` data types. C programs often use pointer arithmetic to manipulate character arrays. The Java `String` and `StringBuffer` data types manage strings in a different way than C character arrays, and thus, a conversion would be rather complicated and would probably require manual intervention. We are considering the conversion of some character arrays that do not involve pointer arithmetic to Java `String` data types in a future version of Ephedra.

As a means for reducing the amount of manual labour required in a source conversion, Terekhov and Verhoef suggest the use of data type emulation, in which the original unsigned data types would be emulated through classes in Java or by instrumenting the code operating on these data types [63]. It has the advantage that the conversion can be fully automated with no need for a developer to understand the original code to validate the conversion. We rejected this option for the conversion of fundamental data types to improve the performance and readability of the generated code: a language conversion is a long-term investment and should emphasise maintainability.

C2J and C2J++ perform similar transformations [35, 65, 50]. Both convert unsigned C data types to the closest signed Java data type without informing the developer about the change. C2J uses larger data types for performing comparisons of unsigned data types to ensure that the comparisons yield the right result. This does not work for `unsigned long` data types though, since there is no integral Java data type larger than `long`. Similar to Ephedra, C2J uses byte arrays to store character strings. In contrast, C2J++ converts them to Java's `char` and `String` data types.

9.3.2 Derived Types and Type Names (§§r.3.6.2 – r.3.6.3)

Both C and Java support the definition of derived types. Their transformations are discussed in detail in the following sections. Both fundamental and derived types can be given additional names in C using the **typedef** mechanism. There is no equivalent in Java. In the transformation, all references to the additional names of a type need to be changed to refer to the original names, unless the original data type is unnamed. In that case, exactly one of the additional names can be used to name the unnamed data type. Figure 9.3 shows an example transformation.

Original C Code

```
typedef int INTEGER;
typedef struct A {
} B;
typedef struct {
} C;

...

  INTEGER i;
  B b;
  C c;
```

Transformed Code

```
class A {
}
class C {
}

...

  int i;
  A b;
  C c;
```

Figure 9.3: Transformation of **typedef**

C2J++ [35, 65] discards **typedef** declarations during the conversion from C to Java. After a conversion with C2J [50], none of the C **typedef** and structured type declarations are visible in the Java code. Their semantics are now hidden in the Java source code statements that access C2J's designated storage area.

9.4 Standard Conversions (§r.4)

9.4.1 Float and Double (§r.4.3)

In both C and Java, conversion from a less precise to a more or equally precise floating point type does not change the value being converted. Conversion from a more precise to a less precise floating point type in C is implementation dependent. The value may be rounded either up or down. If the value is out of the range of the less precise type, the result is undefined. Java rounds towards zero. If the value is out of the range of the less precise type, the result is positive or negative infinity. Since the conversions as implemented in Java do not conflict with the ones of C, no transformation is necessary.

9.4.2 Floating and Integral (§r.4.4)

As with the conversion between floating point types, in C the results of conversions involving boundary conditions are undefined or implementation specific. In Java, they are defined. When moving from C to Java, no transformations are necessary, as the Java implementations of the conversions do not contradict the C language specification.

9.4.3 Arithmetic Conversions (§r.4.5)

In expressions involving binary operators, both C and Java implicitly convert the operands to have matching data types. In these conversions, the smaller of the two data types is transformed to the larger one. In some cases involving operations on signed and unsigned data types in C, both operands are transformed to a data type that is larger than either one of them. Since we decided to transform all C data types to signed Java data types, this latter conversion does not apply in the converted code.

C does not require arithmetic types to be converted explicitly to boolean types where these are required or returned by operators or statements and vice versa. An arithmetic zero value is considered a boolean *false*, all non-zero arithmetic values are considered a boolean *true*. A boolean *false* is implicitly

converted to zero, a boolean *true* is implicitly converted to one. Java does not allow for implicit or explicit conversion between these types, so special expressions have to be used where a conversion is required. Figure 9.4 shows an example.

Original C++ Code	**Transformed Code**
`int i;`	`int i;`
`bool b;`	`boolean b;`
`...`	`...`
`b = i;`	`b = i != 0 ? true : false;`
`i = b;`	`i = b ? 1 : 0;`

Figure 9.4: Conversion between boolean and arithmetic types

9.4.4 Pointer and Reference Conversions (§§r.4.6 – r.4.7)

In C, a constant expression that evaluates to zero is converted to a pointer (commonly called *null pointer*) implicitly when necessary. This conversion is illegal in Java, even if done explicitly. Java provides the `null` keyword to represent a *null reference*.

In C, a variable of type `void*` can hold a pointer to any variable or function. In Java, the `Object` class is the super class of *all* Java classes, so a variable of type `Object` can be a reference to an object of any Java class. Ephedra provides wrapper classes for fundamental data types to emulate C pointers to fundamental data types. Later sections discuss transformations of C pointers to Java references in more detail.

Both C++ and Java support implicit conversions from a pointer to a derived class to a pointer of one of its base classes. In C++, there are some restrictions if *protected* or *private* inheritance is used. Since all inheritance in Java is *public*, these restrictions do not apply in Java. In particular, all implicit type casts between classes done by a C++ compiler are also valid in Java and performed implicitly by the Java compiler.

9.4.5 Pointers to Members (§§r.4.8, r.5.5, r.8.2.3)

C++ pointers to members are a little known feature of the language, and thus only infrequently used. There is no equivalent in Java, though Java's reflection API could be used to emulate similar constructs in Java. Ephedra does not support the conversion of pointers to members. The software engineer has to decide for a suitable transformation on a case-by-case basis. We do not discuss pointers to members further in this dissertation.

9.5 Expressions (§r.5)

An expression is a sequence of operators and operands that specifies a computation. Their syntax, order of evaluation, and meaning is defined for fundamental data types in both C and Java. In some cases, C defines the behaviour of an expression by specifying that the result is undefined in some cases. Java usually has stricter definitions, defining a result even for cases that are undefined in C. We mention differences between C and Java in the following sections only if the Java definition is laxer than the C equivalent, that is, if special attention needs to be paid during the conversion process.

In C++, operators can be overloaded for use with class types. A C++ compiler transforms uses of overloaded operators into function calls to specific member functions of the class for which the operator is defined. Since Java does not support operator overloading, these same transformations have to be performed in a source conversion from C++ to Java. The member functions have to be renamed since their names contain characters that are not permitted in Java identifiers. Ephedra chooses new names reflecting the origin and purpose of these functions, as shown in Table 9.2.

C++ allows the user to define conversions of class objects to and from fundamental types and pointers. The compiler applies them implicitly if they are unambiguous by invoking special conversion functions defined by the user. Java does not support such conversions. During a source transformation, these implicit conversions need to be turned into explicit function calls. The conversion functions will have to be renamed since their names contain characters

that are not permitted in Java identifiers.

Original Operator Function Name	Transformed Function Name
`operator =()`	`operatorASSIGN()`
`operator +()`	`operatorADD()`
...	...
`operator int()`	`toInt()`
`operator char*()`	`toString()`

Table 9.2: Renaming of operator functions

9.5.1 Postfix Expressions (§r.5.2)

9.5.1.1 Subscripting (§r.5.2.1)

Both C and Java define a subscript operator. The C subscript operator can be applied to any array or pointer. In Java, the subscript operator can only be applied to arrays. Chapter 8 explains in detail how these data types are converted to Java and how the behaviour of the C subscript operator is emulated in Java.

C2J++ [35, 65] does not modify subscript expressions, so the resulting Java code may not be correct, possibly not even syntactically. C2J [50] converts subscript expressions to expressions accessing its dedicated storage area, and thus hides the subscripting semantics.

9.5.1.2 Function Call (§r.5.2.2)

Both C and Java define a function call operator. An expression specifying the function to be called as first operand is followed by a possibly empty list of expressions containing the parameters enclosed by parentheses.

C allows various kinds of expressions for the first operand while Java restricts this first operand to be the name of a method. This causes problems in the conversion of function pointers. Other features of C not supported in Java are variable length parameters lists and default arguments in C++. Section 9.7.4 explains how Ephedra resolves these issues.

When a function is called in C, each formal parameter of the function is initialised with its actual argument. In the process of this initialisation, temporary variables may have to be created if a formal parameter is of reference type. In the source conversion, these temporary variables have to be created explicitly where necessary.

In C, the order of evaluation for the parameters is undefined. In Java, parameters are evaluated from left to right. Since there is no contradiction, no transformation is necessary.

9.5.1.3 Explicit Type Conversion (§r.5.2.3)

In C++, a type name followed by a parenthesised expression list constructs a value of the specified type given the expression list while possibly creating a temporary object. If the type refers to a class and a constructor matching the expression list has been defined, that constructor is used.

There is no syntactic equivalent for this explicit type conversion. If the type name refers to a fundamental type or a type derived from it, the expression can be converted in the same way as a *type cast expression* (Section 9.5.3).

If the type name refers to a class type, Ephedra transforms the expression to a Java *new expression* that calls an appropriate constructor of that class. This is roughly equivalent to what a C++ compiler would do implicitly, except that the constructed temporary object is destructed implicitly by the compiler in the case of C++ at a known time but recycled at an unknown time by the garbage collector in Java. If the destructor of a class has side effects, this conversion may change the behaviour of the program. We still favour this solution over a more C++ like solution since it increases readability. Figure 9.5 shows an example of the conversions applied to explicit type conversion involving class types.

9.5.1.4 Class Member Access (§r.5.2.4)

In C, there are two different ways to access members of a class. If the value of a variable is a class instance, then members of that instance can be accessed using the dot postfix expression. If the object is referred to by a pointer

Original C++ Code

```
class A {
public:
  A(int i) { }
  static void f(A a) { }
};

...

  A::f(A(1));
```

Transformed Code

```
class A {

public A(int i) { }
public static void f(A a) { }
}

...

  A.f(new A(1));
```

Figure 9.5: Explicit type conversion (as postfix expression)

variable, the arrow postfix expression is normally used. In Java, objects are always referred to by references. The dot postfix expression is used to access its members. In a conversion, all arrow postfix expressions have to be replaced by dot postfix expressions. Figure 9.6 shows an example.

Original C++ Code

```
class A {
public:
  int i;
};

...
  A a;
  A *pa;
...
  a.i = ... ;
  pa->i = ... ;
```

Transformed Code

```
class A {

  public int i;
};

...
  A a = /* initialisation */;
  A pa;
...
  a.i = ... ;
  pa.i = ... ;
```

Figure 9.6: Class member access

C2J++ [35, 65] transforms class member access the same way Ephedra does. As C2J [50] does not maintain definitions of structured data types, accesses to class members are hidden in statements accessing C2J's dedicated storage area.

9.5.1.5 Increment and Decrement (§r.5.2.5)

Both C and Java provide post-increment and post-decrement operators. They increment resp. decrement their operand *after* returning its original value as result. The operators are defined for arithmetic types in both languages. No conversion is necessary in these cases.

C also defines these operators for pointer types. The C compiler assumes the pointer variable to contain the address of an element of an array of the specified type and increments or decrements the address in the pointer variable to point to the next or previous element in the array. Java does not have pointer types and does not allow for arithmetic operations to be performed on references. Chapter 8 shows how pointers and pointer arithmetic are transformed to Java in the Ephedra approach.

9.5.2 Unary Operators (§r.5.3)

C defines the indirection and address-taking operators to allow for conversions between pointer and non-pointer types. As these types do not coexist in Java, conversions are not necessary. In the transformation from C to Java, the indirection and address-taking operators are stripped. Some other transformation may have to be performed at the same time. Chapter 8 shows these transformations in detail.

C's unary arithmetic operators + (unary plus), - (unary minus), ! (logical negation), and ~ (one's complement) also exist in Java. However, Java puts some restrictions on the operands of these operators. The operands need to be of arithmetic types for the unary plus and minus operators, boolean for the logical negation operator, and of integral types for the one's complement operator. In the source transformation, type conversions may need to be inserted to comply with these restrictions. Section 9.4.3 explains how this is done.

9.5.2.1 Increment and Decrement (§r.5.3.1)

The pre-increment and pre-decrement operators are handled analogously to the post-increment and post-decrement operators (Section 9.5.1.5, Chapter 8

for pointers).

9.5.2.2 Sizeof (§r.5.3.2)

C defines a `sizeof` operator to allow a programmer to determine the size in bytes of an expression or type. Java does not have an equivalent operator, in fact, the actual size of variables is hidden from the programmer. The `sizeof` operator is frequently used in two different scenarios in C programs: to allocate storage for one or more variables and to move storage areas between variables or between variables and disk storage. For primitive data types, Ephedra replaces `sizeof` expressions by constants (defined in Ephedra's primitive wrapper classes) that reflect the size in bytes of a variable of that type on disk storage. For class data types, Ephedra stores the sum of the sizes of all members in a static class variable and replaces `sizeof` expressions for that type by a reference to that variable. This transformation closely emulates the behaviour of the C `sizeof` expression and allows for the correct conversion of memory allocation and disk storage access. Figure 9.7 shows an example conversion.

Original C++ Code

```
class A {
public:
  int i;
};

class B : A {
public:
  float f;
};


...
  ... = sizeof(int);
  ... = sizeof(B);
```

Transformed Code

```
class A {
  public int i;
  public int sizeInBytes =
    EInt.sizeInBytes;
}
class B extends A {
  public float f;
  public int sizeInBytes =
    A.sizeInBytes +
    EFloat.sizeInBytes;
}
...
  ... = EInt.sizeInBytes;
  ... = B.sizeInBytes;
```

Figure 9.7: Sizeof expressions

9.5.2.3 New (§r.5.3.3)

The C++ **new** operator allocates and optionally initialises memory from the program's heap. It allocates sufficient memory to store a value of the specified primitive or class type. In the case of arrays, it allocates memory for all array elements. The operator initialises the storage allocated using the specified initialiser expression or a constructor in case of classes.

Java's **new** operator is defined only for class and array types. It allocates memory for one object of the specified class or an array of references to objects of the specified class. In the latter case, it does not allocate storage for these objects. This has to be done in separate steps.

Several transformations are necessary. Primitive types have to be replaced by their corresponding Ephedra wrapper types. The allocation of arrays has to be performed in several steps to allocate the storage for both the array and the objects within the array. Ephedra provides a general purpose Java method **opNewARRAY** for every class that is invoked for every occurrence of a **new[]** operator call in the C++ code, so these steps do not have to be coded repeatedly. Figure 9.8 illustrates the conversions.

Original C++ Code

```
class A {
public:
  A(int, int, int) { }
};


... = new int;
... = new int(4);
... = new int[10];


... = new A;
... = new A(1, 2, 3);
... = new A[5];
```

Transformed Code

```
class A {
public A(int, int, int) { }
public A[] opNewARRAY
  (int size, A[] initializer) { ... }
}

... = new pInt(0);
... = new pInt(4);
... = new pInt
  (pInt.opNewPrimitiveARRAY(10, null));

... = new pA(new A());
... = new pA(new A(1, 2, 3));
... = new pA(X.opNewARRAY(10, null));
```

Figure 9.8: New expressions

9.5.2.4 Delete (§r.5.3.4)

The C++ `delete` operator de-allocates storage that was allocated using the `new` operator. For class types, it calls the class' destructor before deallocating the storage. In case of arrays, the destructors for all the array's elements are called and the storage for all elements is de-allocated. If the operand to the `delete` operator is mutable, its value is undefined after the application of the operator. The `delete` operator can be applied to a null pointer. In that case, it has no effect.

Java does not provide a `delete` operator. The programmer cannot force the Java Virtual Machine to de-allocate memory. Rather, the Java Virtual Machine de-allocates unused memory in usually unknown intervals if the storage is needed for other objects using garbage collection algorithms. At this time, the `finalize()` method of the object to be deleted is invoked by the Java Virtual Machine.

For primitive types, Ephedra replaces the `delete` operator by an assignment of the null reference to the variable referencing the storage to be de-allocated. This helps the Java Virtual Machine to determine that the storage is no longer needed, and also helps the programmer in understanding the code.

The destructors of C++ classes are transformed to regular member methods in Java classes, and calls to the `delete operator` are replaced by invocations of these methods. Ephedra provides a general purpose method `opDestructARRAY` that invokes the destructors for all elements of an array. Figure 9.9 shows these transformations on an example.

9.5.3 Explicit Type Conversion (§r.5.4)

A C cast-expression allows for explicit type conversion of variables or values. There are different kinds of type conversions:

- conversions between values of arithmetic types
- conversions between different pointer or reference types
- conversions between pointer and integral types

Original C++ Code

```
class A {
public:
  ~A() { }
};

int *i;
A *a;

// de-allocate one object
delete i;
delete a;

// de-allocate array
delete[] i;
delete[] a;
```

Transformed Code

```
class A {
  public static A DESTRUCT(A _this) {
    // C++ original destructor code
    return null;
  }
  public static A opDestructARRAY(A[] array) {
    // invoke destructor for every element
    return null;
  }
}

pInt i;
pA a;

// de-allocate one object
i.set(null);
a.set(A.DESTRUCT(a));

// de-allocate array
i.set(null);
a.set(A.opDestructARRAY(a));
```

Figure 9.9: Delete expressions

- conversions using user-defined conversion operators

Conversions between values of arithmetic types can be performed similarly in Java. See Section 9.4.3 for conversion to and from boolean types.

Conversions between different pointer and reference types in C are converted to conversions between reference types in Java. These conversions are only legal if the types involved are related by inheritance. All other such conversions, as well as conversions between pointers and integral types, have to be removed by the developer before the source code is transformed. Section 7.2 discussed the removal of these type casts in detail.

Java does not support user-defined conversion operators. In a conversion to Java, they have to be converted to regular member functions of a class, and a type cast using these conversion operators has to be converted to an invocation of the converted regular member function.

9.5.4 Arithmetic and Logical Operators (§§r.5.6 – r.5.17)

The logical, arithmetic, and assignment operators provided by C are all present and defined in a compatible way in Java. As with unary operators, Java puts some restrictions on the operands of many operators. Arithmetic operators need to have arithmetic operands, logical operators need to have boolean operands. In a conversion from C to Java, these operands have to be transformed as discussed in Section 9.4.3.

Care must be taken when handling compound assignment operators that involve incompatible types. They may have to be transformed into an equivalent expression using a simple assignment operator as shown in Figure 9.10.

Original C++ Code	**Transformed Code**
`bool b;`	`boolean b;`
`...`	`...`
`b += 1;`	`b = (b ? 1 : 0) + 1 != 0;`

Figure 9.10: Compound assignment operators

As C2J++ [35, 65] performs a lexical analysis of the code only, it has no means to determine the type of a variable, and can therefore not perform any conversions necessary due to type conflicts. C2J [50] transforms the source code similarly to Ephedra.

9.5.5 Comma Operator (§r.5.18)

C allows for multiple expressions to be chained together into one expression using commas. Expressions of this kind are frequently used in C macros. All subexpressions are evaluated, but only the result of the final subexpression is used. Java does not support this kind of expression (except in `for` statements), but a transformation is not difficult: all but the last subexpression are augmented with extra code to make them into boolean expressions that are always true, and all subexpressions are concatenated with logical AND operators.

If one of the subexpressions has a `void` value, it cannot be augmented to become a true boolean expression. In this case, the subexpression has to be wrapped into a method that always returns `true`, and a call to this method is inserted into the comma expression. Variables used within the subexpression have to be passed to the wrapper method. Figure 9.11 shows some example transformations. After the transformation, the code may not be legible any more. If identical sequences of expressions occur in the code several times, the developer should consider wrapping all or some of the expressions in a method to enhance the readability of the code.

An alternative solution is to convert the left-hand side of a comma expression to an independent statement. In certain cases this requires the insertion of temporary variables and conditional statements, which may be difficult to implement in transformation tools. We therefore decided not to pursue this option.

C2J++ [35, 65] does not handle the conversion of comma expressions. C2J [50] converts their semantics exactly, even if this conversion has a negative effect on the readability of the source code.

Original C++ Code

```
void foo(int k) {}

...
int i, h, j;

if (foo(i), i++,
    h--, j *= 5) ;
```

Transformed Code

```
void foo(int k) {}
boolean wrapfoo(int m) {
  foo(m);
  return true;
}
...
int i, h, j;

if (((wrapfoo(i) &&
      i++ != 0 || true) &&
     h-- != 0 || true) &&
    (j *= 5) != 0) ;
```

Figure 9.11: Conversion of comma operator

9.5.6 Constant Expressions (§r.5.19)

C requires constant expressions in several places, such as the case labels of switch statements. This is exactly where Java requires them, and hence, no conversion is necessary.

9.6 Statements (§r.6)

Java supports all statements defined for C with the exception of the `goto` statement. Also, expression statements are more restrictive in Java than in C. As a result, hardly any transformations are necessary for the conversion of statements. The following sections show how to handle `goto` and expression statements.

9.6.1 Labelled Statement (§r.6.1) and Jump Statements (§r.6.6)

Java does not support the `goto` statement. Even though C supports it, it has been used rarely, as its use often leads to poorly structured code. In some cases, `goto` statements have been used to get around limitations of the C language, and Ephedra concentrates on detecting and transforming these

cases, while asking the developer to improve code that uses `goto` statements where they should not be used.

We identify the following classes of `goto` uses and transformations:

nested loops `break` and `continue` statements in C always break or continue the innermost loop only. Breaking out of or continuing an outer loop can be achieved through `goto` statements. In Java, loops can be labelled and `break` and `continue` statements can refer to these labels to signify which loop in a set of nested loops should be broken or continued.

`goto` to the end of a function This scenario is often used to avoid coding cleanup code multiple times in a function. A label is put before the cleanup code, and instead of a `return` statement a branch to that label is used to return from the function. In Java, the same effect can be achieved by putting a label on a lexical block extending from the beginning of the function to just before the labelled statement in the C source and then using a labelled break statement.

Examples of these transformations are shown in Figures 9.12 and 9.13.

Original C++ Code

```
for ( ... ) {
  for ( ... ) {
    if (...)
      goto endOfOuterLoop;
    else
      goto contOuterLoop;
  }
  contOuterLoop: ;
  }
endOfOuterLoop: ;
```

Transformed Code

```
outerLoop:
  for ( ... ) {
    for ( ... ) {
      if (...)
        break outerLoop;
      else
        continue outerLoop;
  }

}
```

Figure 9.12: Goto statements (in nested loops)

Ephedra does not transform other `goto` statements that break the flow of control. As C2J [50] proves, such a transformation is possible, but it usually results in poorly structured code. Our opinion is that a developer should rewrite such code to improve readability, as in C2J++'s approach [35, 65].

Original C++ Code

```
if (failure) {
  // print error message
  goto cleanup;
}
...
cleanup:
  // close files etc.
```

Transformed Code

```
mainblock:
  {
    if (failure) {
      // print error message
      break mainblock;
    }
    ...
  }
  // close files
```

Figure 9.13: Goto statements (at the end of a function)

9.6.2 Expression Statement (§r.6.2)

Most statements in a program are expression statements. Expression statements are statements that consist of zero or one expressions. These expressions are usually assignments or function calls, but C allows most expressions to be used in expression statements. Java allows only assignments, function calls, increment/decrement expressions, and new expressions in expression statements.

Expression statements involving other kinds of expressions may have been used by a programmer to take advantage of their side-effects. In a source conversion, we need to check whether an expression statement that is not a valid expression statement in Java has side effects. If it has side effects, it should be wrapped into an if statement with empty body. Expression statements without side effects can be removed from the program. Figure 9.14 shows some example transformations.

9.7 Declarations (§§r.7 – r.8)

Declarations specify the interpretation given to each identifier in a program. Both in C and Java, some declarations are allowed only in the context of an enclosing class or interface declaration. Java is more restrictive in this respect: only class and interface declarations are allowed in the top-level scope of a program. All other declarations have to occur within a class or interface declaration.

Original C++ Code	Transformed Code
`bool f1() {}`	`boolean f1() {}`
`bool f2() {}`	`boolean f2() {}`
`int i1, i2;`	`int i1, i2;`
`bool b;`	`boolean b;`
`2;`	`// no side-effect, removed`
`i1 == 5;`	`// no side-effect, removed`
`b ? i2-- : i3++;`	`if (b) i2-- else i3++;`
`z1() && z2();`	`if (z1() && z2()) ;`

Figure 9.14: Expression statements

In a conversion from C to Java, the top-level declarations that are not allowed in Java need to be wrapped into classes. Ephedra gathers all top-level declarations of a single file that are not class declarations and transforms them into static members of a class whose name is derived from the file name. Figure 9.15 shows an example transformation.

Original C++ Code	Transformed Code
`// file euroconversion.c`	`class euroconversion {`
`double DMperEURO;`	`  static double DMperEURO;`
`double`	`  static double`
`  DMtoEURO(double DM) { }`	`    DMtoEURO(double DM) { }`
`double`	`  static double`
`  EUROtoDM(double EURO) { }`	`    EUROtoDM(double EURO) { }`
	`}`

Figure 9.15: Wrapping of top-level declarations

9.7.1 Specifiers (§r.7.1)

Specifiers in a declaration identify various properties of the identifiers to be declared. Java supports fewer specifiers than C. Where there are no Java equivalents for C specifiers that have an impact on the functionality of the program, transformations may have to be applied to the code.

9.7.1.1 Storage Class Specifiers (§r.7.1.1)

C defines the four storage class specifiers `auto`, `register`, `static`, `extern`.

`auto` specifies that a variable should be allocated on the stack. It is the default for both C and Java if no storage class specifier is present in a declaration. In a conversion from C to Java, the `auto` specifier can be removed from the source code.

`register` specifies that the value of a variable should be stored in a CPU register to increase performance. It is only a hint to the compiler and may be ignored. In a conversion from C to Java, it can be removed from the source code.

The `static` specifier can be used in different contexts:

- If used in the top-level scope of a source file, it specifies that the identifier should be visible and accessible only in the current source file. Identifiers of the same name can be declared on the top-level of other source files without introducing naming conflicts.

  There is no directly equivalent construct in Java. Being in top-level scope of the source file, the declaration will be wrapped into a class whose name is derived from the source file during the source conversion. Thus, there will not be any naming conflicts with top-level identifiers in other source files. The identifier can be accessed from other source files though.

- If used for a field of a class, it specifies that this field exists only once for all instances of that class. The Java `static` specifier has the same effect.

- If used for a method of a class, it specifies that this method does not operate on an object of a class but provides a service on the class as a whole (possibly using static fields of the class). The Java `static` specifier has the same effect.

- If used for a local variable of a function, it specifies that the variable should be initialised at load time (for C and constant initializer expressions in C++) or only during the first invocation of the method (for

non-constant initialisers in C++) and retain its value between invocations of the method. Java does not support static local variables. In a conversion, they have to be moved from within a function to its enclosing class. If the initial value of the variable depends on the arguments passed to the function, additional code and variables may have to be added to emulate the behaviour of the C code. If several functions define static variables of the same name, some of the variables will have to be renamed. Figure 9.16 shows an example transformation for this scenario. C2J [50] performs the transformation similarly, while C2J++ [35, 65] does not handle this problem at all.

Original C++ Code

```
// file foobar.c
void foo() {
  static int i = 5;
  ... use i ...
}



void bar(int h) {
  static int i = h;

  ... use i ...


}
```

Transformed Code

```
class foobar {
  static int i = 5;
  static void foo() {
    ... use i ...
  }

  static int i1;
  static bool
     i1_initialized = false;
  static void bar(int h) {
    if (!i1_initialized) {
      i1 = h;
      i1_initialized = true;
    }
    .... used i1 ...
}
```

Figure 9.16: Conversion of local static variables

9.7.1.2 Function Specifiers (§r.7.1.2)

C++ function specifiers identify characteristics of functions that are defined. The `inline` specifier is a hint to the compiler that inline substitution of the function is to be preferred to the usual function call implementation. In Java, the compiler does not accept such hints but decides itself where inlining of

function bodies provides advantages. A Java Virtual Machine may even perform inlining during the execution of a program. In a conversion from C to Java, the inline keyword is removed from the source code.

The `virtual` specifier is used in the declaration of non-static class member functions. We discuss the difference between virtual and non-virtual functions and their transformations in Section 9.8.2.

9.7.1.3 The typedef Specifier (§r.7.1.3)

The `typedef` specifier is used to introduce a new type name for an existing fundamental or derived type. Its conversion is shown in Section 9.3.2.

9.7.1.4 The template Specifier (§r.7.1.4)

The `template` keyword specifies the declaration following it to be a parameterised type or function. Section 9.13 illustrates the transformation of templates.

9.7.1.5 The friend Specifier (§r.7.1.5)

The C++ `friend` specifier is used to give non-member functions and other classes access to the protected and private members of a class. There is no equivalent construct in Java. Java's default or `package` access class can be used to achieve a similar effect by giving access to class members to all other classes in the same package. This approach is less flexible though, since it gives access to an entire group of classes rather than to a specific set of classes or functions.

9.7.1.6 Type Specifiers (§r.7.1.6)

The `const` type specifier identifies a variable to be immutable after its initialisation. As such, it can be used in constant expressions (Section 9.5.6). The Java `final` keyword can be used to achieve the same effect. If the `const` type specifier is used for a formal parameter in a function declaration, this parameter cannot be changed by this function but only initialised from the actual

parameter upon function invocation.

C's `volatile` type specifier is a hint to the compiler specifying that a variable may change by means undetectable by the compiler. Compilers usually avoid aggressive optimisations of expressions involving such variables to ensure that they yield the expected result. As all storage accessible to a Java program is under the control of the Java Virtual Machine, the keyword is unnecessary in Java. In a conversion from C to Java, it may be removed. However, a comment should be added to notify the developer that this particular variable was marked as `volatile` in the original code to preserve the documentation aspect of this type specifier.

The type specifier together with the declarator that follows it also identifies the type of the variable or function to be declared or defined by mentioning the name of either a fundamental or predefined type. The following sections explain the different kinds of declarations and show how they are transformed in different contexts.

9.7.2 Enumeration Declarations (§r.7.2)

A C enumeration declaration declares a distinct integral type along with a number of named constants of that type. Values of the enumeration type are implicitly converted to `int` where needed, but explicit type casts are required to convert `int` to the enumeration type. Because of this convention, many legacy programs use unnamed enumeration declarations to declare a number of constants and assign their values to `int` variables in the program.

Java does not support enumeration declarations. One way to convert them from C to Java is to declare the enumeration constants in an interface that bears the name of the enumeration type. Variables of the enumeration type will have their type changed to `int`. Figure 9.17 shows an example using this conversion strategy.

This transformation is straight-forward and correct, but it causes the loss of some type information: in the C program, the variables `i` and `o` were of different type, in Java, they are both of type `int`. Though this difference does not cause any change in the behaviour of the program, the type information

Original C++ Code

```
enum OneTwo {
  one = 1,
  two
};
...
int i = one;
OneTwo o = two;
if (i == o) ...
```

Transformed Code

```
interface OneTwo {
  final static int one = 1;
  final static int two = 2;
}
...
int i = OneTwo.one;
int o = OneTwo.two;
if (i == o) ...
```

Figure 9.17: Conversion of enumeration declaration using an interface

lost in the conversion may have been an important clue to the developer of the program: an assignment of some arbitrary value to an integer variable does not look suspicious, but an assignment of a constant that is not a valid enumeration constant does. We therefore propose a slightly different conversion. The enumeration declaration is converted to a class declaration. The enumeration constants and an integral variable will become members of that class. The type of variables of the enumeration type does not have to be changed in the conversion. Figure 9.18 shows the result of this transformation.

Original C++ Code

```
enum OneTwo {
  one = 1,
  two
};


...


int i = one;
OneTwo o = two;

if (i == o) ...
```

Transformed Code

```
class OneTwo {
  final static int one = 1;
  final static int two = 2;
  int value;
  OneTwo(int src) {
    value = src;
  }
}
...
int i = OneTwo.one;
OneTwo o =
  new OneTwo(OneTwo.two);
if (i == o.value) ...
```

Figure 9.18: Conversion of enumeration declaration using a class

In a conversion with C2J [50], the original enumerated type definitions are lost. Uses of the enumerators are still visible in comments in the transliterated code. C2J++ [35, 65] does not handle enumerated type definitions at all.

9.7.3 Pointers, References, and Arrays (§§r.8.2.1, r.8.2.2, r.8.2.4)

The conversion of pointers, references, and arrays is discussed in detail in Chapter 8.

9.7.4 Functions (§§r.8.2.5 – r.8.3)

Original C++ Code

```
class A { };
A foo(A a1,
      A& a2,
      A* a3,
      int i1,
      int& i2,
      int *i3) {
  ...
  return a1;
}
...
A a;
int i;
a = foo(a, a, &a, i, i, &i);
```

Converted C++ Code

```
class A { };
...
A a;
int i;

A a1 = a;
A& a2 = a;
A* a3 = &a;

int i1 = i;
int& i2 = i;
int* i3 = &i;
...
a = a1;
```

Figure 9.19: A priori conversion of function parameters

The declaration and definition of functions is very similar in C and Java, and only a few transformations are necessary. Most of these happen in the context of conversions discussed in other parts of this chapter:

- Top-level functions need to be wrapped into classes.
- Parameters passed to functions and return values are converted as if they were used in assignments (§12.6.1), similarly to an explicit inlining of the functions called. Figure 9.19 illustrates the conversions applied for the purpose of defining the transformations of function parameters by showing the C++ code before and after these conversions.

  For the assignment of pointers, the conversions as defined in Section 8 apply. The techniques shown in Section 9.8 are used to convert the

assignment of objects.

C++ allows for a function's arguments to be given default values. If the arguments are not specified in a call to this function, these default values are used. Java does not have an equivalent construct. There are two possibilities for a conversion: either the default values are added to incomplete parameter lists in calls to the function, or additional functions are declared that take fewer arguments than the original function and pass the default values to the original function. Figure 9.20 shows this conversion.

Original C++ Code

```
void foo(int i = 0;
         float f = 5.5) {
  ...
}
```

Transformed Code

```
void foo(int i, float f) { ... }

void foo(int i) { foo(i, 5.5); }

void foo() { foo(0, 5.5); }
```

Figure 9.20: Conversion of default arguments

A second language element of C that allows for variable length parameter lists is the *ellipsis*. When the ellipsis is specified in the formal parameters of function, zero or more arguments of arbitrary types may be passed as actual parameters. Macros and functions defined in the C standard headers can be used to access these parameters within the function.

Ephedra defines an ellipsis class that replaces the C ellipsis in function declarations. Uses of macros and functions of the C standard headers are replaced by calls to members of the Ellipsis class. Parameters passed in the function call are wrapped into classes and passed to the ellipsis constructor. The ellipsis class provides a large number of constructors to allow for various argument lists, and stores the parameters in an array member. Figure 9.21 shows an example conversion. Note that there is a slight difference in the behaviour of the code in case of type mismatches: the C compiler and runtime have no possibility to check whether the parameters passed to the function satisfy the assumptions about their type and number. In Java, the bound checks on the array within the Ellipsis class and type checks performed before the execution of type casts will raise exceptions if the function attempts to

access extra or incorrectly typed parameters. The converted code is thus safer than the original code.

Original C++ Code

```
void foo(int num, ...) {
  va_list ap;
  va_start(ap, num);

  for (int i = 0; i < num; i++) {
    double d =
      va_arg(ap, double);
    ...
  }
  va_end(ap);
}
...
foo(3, 7.1, 8.2, 9.3);
```

Transformed Code

```
void foo(int num, Ellipsis __ellipsis) {
  Ellipsis ap = new Ellipsis();
  Ellipsis.operatorASSIGN(ap, __ellipsis);
  for (int i = 0; i < num; i++) {
    double d = ((EDouble) (ap).next()).get(0);
    ...
  }
}
...
foo(3,
    new Ellipsis(new pDouble(7.1),
                 new pDouble(8.2),
                 new pDouble(9.3)));
```

Figure 9.21: Conversion of ellipsis

C2J [50] performs the conversion of ellipses similar to Ephedra. However, it requires one statement for every actual parameter in the function call expression, while Ephedra requires only one expression for the entire ellipsis. C2J++ [35, 65] does not handle ellipses at all.

9.7.5 Initialisers (§r.8.4)

Initialisers are used to assign an initial value to a variable. Their conversion is simple for variables of fundamental data types. If these variables have their address taken or are passed by reference at any point in the program, their type will have been changed to that of a corresponding wrapper variable as explained in Chapter 8. Their initialisers will have to be modified accordingly.

For variables of structured types, additional steps are needed for the conversion of initialisers. A definition of a structured variable in C both allocates storage and possibly initialises the variable. A definition of a reference variable in Java does not allocate storage. When transforming C to Java, transformations that turn the implicit allocation of storage in C into an explicit Java storage allocation need to be applied.

Figure 9.22 illustrates the conversions applied on the initialisers of fundamental and structured data types.

Original C++ Code	**Transformed Code**
`// address not taken`	`// address not taken`
`int i1;`	`int i1;`
`int i2 = 5;`	`int i2 = 5;`
`// address taken`	`// address taken`
`int i3;`	`pInt i3 = new pInt();`
`int i4 = 5;`	`pInt i4 = new pInt(5);`
`int *i5 = &i3;`	`pInt i5 = new pInt().set(i3);`
`int *i6 = &i4;`	`pInt i6 = new pInt().set(i4);`
`// structured`	`// structured`
`class A { };`	`class A { }`
`...`	`...`
`A a;`	`A a = new A();`

Figure 9.22: Conversion of initialisers for variables of fundamental types

9.7.5.1 Aggregates (§r.8.4.1)

In C and C++, a so called *aggregate* is an array or an object of a class with no constructors, no private or protected members, no base classes, and no virtual

functions. A *brace list initialiser* (a list of initialisers enclosed by braces), may be used to initialise the elements of the array or the fields of the class. In multi-level brace list initialisers, lower-level braces may be omitted in some cases. In Java, brace list initialisers are supported only for the initialisation of arrays. For the initialisation of classes, constructors need to be used. Figure 9.23 shows the simplest case of a conversion.

Original C++ Code

```
struct S {
  int i;
  float f;
};

int i[4] = { 1, 2, 3, 4 };
S s = { 5, 7.7 };
S t[2] = { s, { 6, 8.8 } };
```

Transformed Code

```
class S {
  int i;
  float f;
  S(int i, float f) {
    this.i = i;
    this.f = f;
  }
  S(S src) {
    i = src.i;
    f = src.f;
  }
};

int[] i = { 1, 2, 3, 4 };
S s = new S(5, 7.7);
S[] t = { new S(s), new S(6, 8.8) };
```

Figure 9.23: Conversion of brace list initialisers

In C, brace list initialisers may be incomplete and initialise only some of the storage of the variable to be initialised. The remaining storage will be initialised using zero values. In a conversion, one could either include the missing initialisers in the brace list or provide additional constructors and support routines to perform the initialisation, possibly requiring auxiliary variables. Ephedra chooses the second option as it keeps the initialisers easier to read. In particular with large arrays, the first option becomes impractical. Figure 9.24 shows a transformation.

Difficulties arise in the transformation of brace list initialisers when aggregates are nested. It is possible to use one C brace list initialiser to initialise an object that contains an array. Since a brace list initialiser cannot be passed as a parameter to a constructor in Java, an auxiliary variable needs to be used

Original C++ Code

```
struct S {
  int i;
  float f;




};


int i[3] = { 1, 2 };
S s = { 4 };
```

Transformed Code

```
class S {
  int i;
  float f;
  S(int i) {
    this.i = i;
    this.f = 0;
  }
};

int[] temp = { 1, 2 };
int[] i = pInt.opNewPrimitiveARRAY(3, temp);
S s = new S(4);
```

Figure 9.24: Conversion of incomplete brace list initialisers

in the converted code, as shown in Figure 9.25.

Original C++ Code

```
struct S {
  int array[5];




};

...

S s = { 1, 2, 3, 4, 5 };
```

Transformed Code

```
class S {
  int[] array;
  S(int[] array) {
    this.array = array;
  }
}
...
final int[] temp =
  { 1, 2, 3, 4, 5 };
S s = new S
  (pInt.opNewPrimitiveARRAY(5, temp));
```

Figure 9.25: Conversion of nested brace list initialisers

C2J [50] creates a special method for every variable that is brace-list initialised in the original C code to initialise its storage area. The semantics of the initialisation are not apparent from the code any more. C2J++ [35, 65] fails to convert code involving brace-list initialisers completely.

9.7.5.2 Character Arrays (§r.8.4.2)

Both C and Java provide *string-literals* for the initialisation of strings. In C, string-literals are character arrays, and the C compiler appends a null char-

acter to the end of every string-literal. In Java, string-literals are instances of the Java `String` class. String-literals in the transformed program need to be converted to character arrays. Ephedra uses a function of the standard Java API to perform this conversion, as illustrated in Figure 9.26. C2J [50] uses a custom function to perform the same conversion, while C2J++ [35, 65] converts C character arrays to Java strings, so no conversion of string literals is necessary.

Original C++ Code	**Transformed Code**
`char c[] = "message";`	`byte[] c = "message\0".getBytes();`

Figure 9.26: Initialisation of character arrays

9.7.5.3 References (§r.8.4.3)

As explained in Chapter 8, C++ references are converted to Java references in the same way pointers are converted. For most C variables, initialisations and assignments have identical semantics in that the value of the variable is changed. References are special: if we consider references to be a kind of pointer, initialisations define the address stored in the pointer while assignments change the contents of the storage addressed by the pointer. In C++, there is no syntactic difference in the source code that shows the semantic difference. As the concept of Java references is different and not compatible to that of C++ references, the source code has to be changed to achieve the same effect.

Unlike pointers, references can seemingly refer to literals: the C++ compiler creates auxiliary objects to represent these literals. Figure 9.27 shows the transformations necessary to emulate the initialisation of references.

9.8 Classes (§r.9)

C supports three kinds of classes: the members of classes defined using the keyword `class` are private by default and those of classes defined using the

Original C++ Code	Transformed Code
`int i;`	`int[] i = ...`
`// initialisation` `int &r = i;`	`// initialisation` `int[] r = i;`
`// assignment` `r = i;`	`// assignment` `r[0] = i[0];`

`// generation of`
`// auxiliary variable`
`const int &h = 5;`
	`// generation of`
	`// auxiliary variable`
	`final int[] h = { 5 };`

Figure 9.27: Initialisation of references

keyword `struct` or `union` are public by default. Unions hold only one member at a time.

C2J++ [35, 65] converts all structured data types to Java classes. It correctly translates access specifiers, but fails translating some other specifiers such as `static`. It ignores the difference between virtual and non-virtual function overriding and most of the more intricate problems of the language conversion. C2J [50] does not create any structured data types in the transliterated source. The code it generates looks much like unstructured assembly code in Java syntax in that it directly accesses certain storage regions without using symbolic names. Since C2J does not handle C++, we will not mention it in the rest of this conversion catalogue. The conversions it applies to the non-C++ specific language features are correct, but through its use of a dedicated storage area and the abandoning of structured data types, the original data structures of the C code are lost.

9.8.1 Class Members (§§r.9.2, r.9.4)

Members of classes are converted in the same way top-level declarations are converted, except that they need not be wrapped into enclosing classes, since they are already members of a class. In C, the definitions of member functions and static member variables can, and in some cases must, occur outside the class declaration, in the top-level scope of a compilation unit. In Java, they

have to be in the same place as the declarations.

The static keyword, when used for class members, has the same meaning in C++ and Java. No conversions are necessary to handle this language construct.

9.8.2 Non-Static Member Functions (§§r.9.3, 10.2)

C++ distinguishes between two kinds of member functions, non-virtual and virtual. Stroustrup explains the difference as follows [61]:

> If a class `base` contains a `virtual` function `vf`, and a class `derived` derived from it also contains a function `vf` of the same type, then a call of `vf` on an object of class `derived` invokes `derived::vf` (even if the access is through a pointer or reference to `base`). The derived class function is said to override the base class function.

In contrast, if the function had been declared non-virtual, the call would have invoked `base::vf`.

When talking in C++ terminology, functions in Java always override functions of the same type in base classes: all non-static Java functions are virtual. Therefore, non-virtual C++ functions need to be transformed to emulate the behaviour of the C++ code in Java. Ephedra converts non-virtual C++ functions to static Java functions. Since static functions operate on a class rather than on an object, they are not implicitly passed a reference to an object. It has to be passed explicitly. Accesses to the fields and functions of the originally implicitly passed object need to be changed to refer to the explicitly passed object. Figure 9.28 illustrates the necessary conversions.

There are drawbacks to this solution: It is not customary to declare functions to be static and then to pass an object that the function is to operate on in mainstream Java programs. An experienced Java programmer would design his or her algorithms differently to take advantage of Java's function calling semantics. Thus, the transformed code will look unfamiliar and possibly even confusing to Java programmers. On the other side, the transformed code may perform better than code using non-static Java functions. There is a run-time

Original C++ Code

```
class A {
  int j;
  virtual void f1(int i) {
    // call to non-virtual
    f2(this->j);
  }
  void f2(int i) {
    // call to virtual
    f1(this->j);
  }
};

class B : A {
  void f1(int i) { ... }
  void f2(int i) { ... }
}
```

Transformed Code

```
class A {
  int j;
  void f1(int i) {
    // call to non-virtual
    f2(this.j);
  }
  static void f2(A _this, int i){
    // call to virtual
    _this.f1(_this.j);
  }
};

class B extends A {
  void f1(int i) { ... }
  static void f2(A _this, int i)
    { ... }
}
```

Figure 9.28: Conversion of member functions

overhead when non-static Java functions are called since the program needs to determine at run-time which function to invoke. It is difficult for compilers and Java Virtual Machines to optimise code using this calling technique.

9.8.3 Unions (§r.9.5)

Unions are classes that hold only one of its members at a time. There is no equivalent language construct in Java. Section 7.2 describes a technique for turning unions into hierarchies of classes. A different approach is to turn unions into regular classes. Since Java classes provide storage for all their members all the time, they provide all the features provided by a union in C. As C compilers optimise storage allocation for unions, taking advantage of the fact that they need to provide storage for only one of their members at a time, this conversion may cause a large memory overhead if the union has many fields. A software engineer should evaluate which approach is better for a specific problem.

9.8.4 Bit-Fields (§r.9.6)

C bit-fields are integer variables of a specified size. They have to be members of classes. Ephedra converts them to corresponding integral Java types. If there is no exact match, Ephedra uses the next larger integral type. As with the conversion of other integral types, there is the danger that the converted code does not perform correctly because of assumptions about the size and signedness of the original data type. The software engineer needs to review the converted code to make sure that the assumptions still hold.

9.8.5 Nested and Local Class Declarations (§§r.9.7, r.9.8)

Both C and Java allow class declarations to be nested. Nested C++ classes correspond to *staticly* nested classes in Java, thus, in a conversion, the `static` keyword needs to be inserted, as shown in Figure 9.29.

C also allows a class to be declared within a function. It is then declared in the scope of that function only. There is no equivalent language feature in Java, but it may be emulated by including the declaration of the local class in C as a static nested class declaration in Java. Care must be taken to resolve naming conflicts because the class declarations move from an inner scope to an outer scope in this conversion. Figure 9.29 illustrates the transformations.

9.9 Derived Classes (§r.10)

As an object-oriented programming language, both C++ and Java provide mechanisms for deriving one class from other classes. Java's language constructs are more restrictive than those of C++, so transformations are necessary in some cases. As Ephedra's focus is mainly on the conversion of procedural C code, it does not perform automatic transformations to handle the differences between C++ and Java inheritance. Instead we provide some suggestions and references to related work.

Both C++ and Java support the definition of functions with no implemen-

Original C++ Code

```
class A {
  class B {
  };
  void f1() {
    class C {
      int i;
    };
    C c;
  }
  void f2() {
    class C {
      float f;
    };
    C c;
  }
}
```

Transformed Code

```
class A {
  static class B {
  }
  static class f1_C {
    int i;
  }
  void f1() {
    f1_C c;
  }
  static class f2_C {
    float f;
  }
  void f2() {
    f2_C c;
  }
}
```

Figure 9.29: Conversion of nested class declarations

tation as members of classes. In C++, they are called *pure virtual functions*, in Java *abstract methods*. A class with such a function is called *abstract class*. It is not possible to create an object of an abstract class, but only of subclasses of it that provide an implementation for all of its abstract methods. In C++, programmers do not need to explicitly identify a class as being abstract. In Java, one needs to use the `abstract` keyword in the definition of the class. Figure 9.30 shows the necessary conversions.

Original C++ Code

```
// class implicitly abstract
class A {
  // pure virtual function
  virtual void f() = 0;
};
```

Transformed Code

```
// class explicitly abstract
abstract class A {
  // abstract method
  abstract void f();
}
```

Figure 9.30: Conversion of abstract classes

C++ supports multiple implementation inheritance, allowing one class to have several base classes. Java supports single implementation inheritance and multiple interface inheritance (i.e., a class can have at most one base class and can implement several interfaces). Interfaces are restricted kinds of classes

that can contain only abstract methods and static fields. Wen examined the use of multiple inheritance in C++ and developed techniques for its conversion using the inheritance schemes provided by Java [68].

C++ distinguishes between public, protected, and private inheritance. Java knows only public inheritance. The conversion from private or protected inheritance to public inheritance does not usually cause a problem.

9.10 Member Access Control (§r.11)

C++ and Java both allow members of classes to be declared as private, protected, or public members, thus restricting other classes' access to these members. Java's protected access is less restrictive than the one of C++, since all classes in a package are permitted to access each other's protected fields. This difference does not cause a problem in the conversion.

We discussed access control with respect to inheritance (Section 9.9 and the `friend` specifier earlier (Section 9.7.1.5.

9.11 Special Member Functions (§r.12)

Both C++ and Java use special member functions of classes during object creation and deletion. C++ also defines special functions for copying or converting objects. These special member functions are frequently called and even generated implicitly.

9.11.1 Constructors (§r.12.1)

Constructors are used to initialise newly created objects. Both C++ and Java compilers create *default constructors* to assign default values to the fields of an object, if no constructors for this class have been defined in the source code. C++ also creates a *copy constructor* in this case. It copies all fields of one object to those of another object: for fields of class types, this is done by invoking the copy constructors of these fields; for fields of fundamental data types or aggregates, a binary copy is created. As discussed in Section

9.7.5, additional constructors may have to be created to convert brace list initialisers. Ephedra generates the source code for all necessary constructors. *Copy constructors* are discussed in more detail in Section 9.11.4.

In C++, parameters for the initialiser of base classes are specified in the header of the constructor of the derived class. In Java, a call to the super class' constructor has to be the first statement in the constructor's body. Figure 9.31 shows the conversions applied.

Original C++ Code

```
class A {
public:
  A(int) { }
};
class B : A {
public:
  B(int i, float f) : A(i) {
  }
};
```

Transformed Code

```
class A {

  A(int) { }
};
class B : A {
  B(int i, float f) {
    super(i);
  }
};
```

Figure 9.31: Conversion of base class initialisers

9.11.2 Conversions (§r.12.3)

C++ applies constructors and special conversion functions implicitly to perform necessary type conversions. Java does not support these conversion functions, and does not apply any constructors implicitly. During the conversion from C++ to Java, the conversion functions need to be renamed to use valid Java identifiers as names as outlined in Section 9.5. Implicit calls to these conversion functions or constructors need to be made explicit. Figure 9.32 illustrates the transformations.

9.11.3 Destructors (§r.12.4)

The conversion of destructors has already been mentioned in connection with the `delete` operator (Section 9.5.2.4). Since C++ destructors are special functions and do not bear names that are valid in Java, Ephedra names them

Original C++ Code

```
class X {
public:
  X(int);

  operator int();
};
...
  X a = 2;

  int i = a;
```

Transformed Code

```
class X {

  public X(int);

  public int toInt();
};
...
  X a = new X(2);

  int i = a.toInt();
```

Figure 9.32: Implicit type conversions using constructors and conversion functions

`DESTRUCT()` in the converted code. In all other aspects, including the conversion of call semantics in the case of non-virtual destructors, they are converted like regular functions.

9.11.4 Copying Class Objects (§r.12.8)

In addition to the copy constructor mentioned earlier, C++ uses the assignment operator function to perform assignments between objects of classes. Under certain conditions, one or both of them are automatically generated by a C++ compiler. As Java compilers do not generate these functions, they need to be coded explicitly during the conversion from C to Java. As the assignment operator function bears a name that is not a valid Java name, it needs to be renamed as outlined in Section 9.5.

It is important to recall the different assignment semantics in C and Java. Assignments between structures in C will cause all the structures' fields to be assigned. With a Java assignment statement, only the references to the classes will be assigned. The C assignment semantics have to be coded explicitly in Java to ensure that variables of reference types and arrays are copied properly. Ephedra emulates these assignment semantics by replacing the assignment by an automatically generated assignment function. The constructor will also have to allocate storage for the variables; in case of arrays of primitive variables, Java's `clone()` method can be used to do this in an efficient way. Figure 9.33

illustrates the transformations on a few examples.

9.12 Overloading (§r.13)

C++ and Java use mostly the same rules for overloading functions. However, problems can arise if a function is defined for two distinct C++ types that are identical after the conversion to Java. In this case, some or all of the overloaded functions need to be renamed, as illustrated in Figure 9.34.

Java does not support operator overloading. Operator functions need to be renamed to bear valid Java names, and implicit calls to these functions need to be turned explicit by changing the application of operators to function calls as shown in Section 9.5. This can have the unfortunate effect that the transformed code becomes difficult to read.

9.13 Templates (§r.14)

Ephedra does not provide a good solution for the conversion of templates (also called generic or parameterised types) yet. In the conversion from C++ to Java, one distinct class is created for every instantiation of a template. As this is exactly what a C++ compiler does during the compilation of a program, there is no performance overhead in this approach. However, since there are now multiple copies of the template's code in the source, the code is more difficult to maintain.

There are several Java dialects that support templates, and efforts are underway to create a standard way for their definition in an upcoming release of the Java programming language [54, 48, 64]. A future release of Ephedra could take advantage of these language extensions to achieve a better transformation of templates from C++ to Java.

Original C++ Code

```
struct A { int i; };
struct B { A a; };
struct C : A { int i[5]; A a[5]; };
```

Transformed Code

```
class A {
  public int i;
  public A(A src) { i = src.i; }
  public static A operatorASSIGN(A _this, A src) {
    _this.i = src.i; return _this;
  }
}
class B {
  public A a;
  public B(B src) { a = new A(src.a); }
  public static B operatorASSIGN(B _this, B src) {
    A.operatorASSIGN(_this.a, src.a); return _this;
  }
}
class C extends A {
  public int[] i;
  public A[] a;
  public C(C src) {
    super(src);
    i = (int[]) src.i.clone();
    a = new A[5];
    for (int i = 0; i < 5; i++) { a[i] = new A(src.a[i]); }
  }
  public static C operatorASSIGN(C _this, C src) {
    A.operatorASSIGN((A) _this, src);
    for (int i = 0; i < 5; i++) { _this.i[i] = src.i[i]; }
    for (int i = 0; i < 5; i++) {
      A.operatorASSIGN(_this.a[i], src.a[i]);
    }
    return _this;
  }
}
```

Figure 9.33: Copy constructors and assignment operators

Original C++ Code	Transformed Code

`// sufficiently different`
`void f1(int);`
`void f1(double);`
	`// sufficiently different`
	`void f1(int) { ... }`
	`void f1(double) { ... }`

`// identical after conversion`
`void f2(signed int);`
`void f2(unsigned int);`
	`// identical after conversion`
	`void f2_signed(int) { ... };`
	`void f2_unsigned(int) { ... };`

`void f3(int& i);` `void f3(int* i);`	`void f3_ref(EInt i);` `void f3_ptr(EInt i);`

Figure 9.34: Conversion of overloaded functions

9.14 Exception Handling (§r.15)

In C++, variables of any data type can be used to raise an exception. In Java, only objects of the `Throwable` class and its subclasses can be used. C++ functions need not declare which exceptions they can possible raise, while Java functions can raise only exceptions that are mentioned in their declaration or caught within their body. An exception to this rule is Java's `RuntimeException`. An exception of class `RuntimeException` or one of its subclasses can be thrown without being caught in a function's body or declared in its definition.

In a transformation from C++ to Java, all C++ classes that are used in a `throw` or `catch` clause could be changed to extend Java's `RuntimeException` class. Since in C++ it is possible to throw variables of any data type, Ephedra's primitive wrapper classes and possibly even pointer classes would have to be changed to extend the `RuntimeException` class. We conducted experiments using this approach that showed that it results in a significant performance overhead. Objects extending Java's `Throwable` class, which is a superclass of the `RuntimeException` class, require a large amount of memory and processing time upon their creation, so programs run about two to four times slower than if a better approach is used.

We therefore developed a more efficient and more effective way to convert C++ exceptions to Java. Ephedra defines *one* exception class that extends

`RuntimeException`. It acts as a wrapper for the C++ data types used for raising an exception. Where a `throw` clause occurs in the original C++ code, it is replaced by a `throw` clause raising an exception of Ephedra's exception wrapper class. The original `throw` argument is passed to the constructor of the exception class. All `catch` clauses of a `try` statement are combined into a single `catch` clause that tests for Ephedra's exception wrapper class. Java's `instanceof` operator is then used to test the type of the class wrapped by Ephedra's exception class. Figure 9.35 illustrates the conversion.

The conversion we propose is correct and does not cause performance or safety problems during the execution of a converted program. However, it is not optimal because it does not conform to traditional Java coding standards. It is therefore confusing for software engineers who are not familiar with Ephedra. In the long term, converted code involving exceptions should be reviewed and manually modified by software engineers to conform to Java coding standards.

9.15 Preprocessing (§r.16)

In C programs, `#define` directives are frequently used to define constants. As C++ and Java provide better means for declaring constants, `#define` directives should be converted to C++ constants before the code is converted to Java. In other cases, the `#define` directive is used to define very short functions. These functions should be converted to regular C++ functions before the conversion to Java as well. Ephedra does not provide tools for the automatic conversion of these language constructs yet.

9.16 Summary

In this chapter, we presented a detailed catalogue showing the code transformations that need to be applied in a source code migration from C to Java. Following the structure of the C++ reference manual, we explained the options for the conversion of each C++ language element along with their

Original C++ Code

```
class MyException { };
...
  try {
    throw 5;
    throw MyException();
  } catch (int i) {
  } catch (MyException e) {
  } catch (...) {
  }
```

Transformed Code

```
class EphedraException extends java.lang.RuntimeException {
  Object target;
  EphedraException(Object o) {
    target = o;
  }
}

class MyException { };
...
  try {
    throw new EphedraException(new pInt().set(5));
    throw new EphedraException(new MyException());
  } catch (EphedraException ee) {
    if (ee.target instanceof pInt) {
      int i = ((pInt) ee.target).get(0);
    } else if (ee.target instanceof MyException) {
      MyException e = (MyException) ee.target();
    } else {
    }
  }
```

Figure 9.35: Conversion of C++ exceptions

consequences. The remaining chapters of this dissertation evaluate Ephedra by presenting tools that automate Java to C source transformations and case studies performed using these tools according to the Ephedra approach.

Part III

Evaluation

In the previous chapters of this dissertation, we presented the Ephedra approach for converting C and C++ programs to Java source code. In Chapter 5, we had prioritised goals and requirements for the Ephedra approach, which we would like to reiterate here:

- maintainability,
- functional equivalence,
- high automation,
- efficiency of generated code, and
- efficiency of tools.

To show that the Ephedra approach satisfies these requirements and to evaluate its correctness and practicality, we implemented tools to perform most of the code transformations we described. We tested the tools, and thus the Ephedra approach, on several representative C programs and evaluated the quality of the converted source code by assessing its maintainability, conformance with Java coding standards, and performance as compared to the original C code.

The subject systems we used for the evaluation cover a large part of the C and C++ programming languages, but are not exhaustive in the sense of exercising the complete set of language features offered by C and C++. In the future, we will conduct more case studies to validate Ephedra more comprehensively. However, the results of the initial case studies are encouraging and show that our overall conversion strategy is feasible.

Chapter 10

Implementation

One of the goals of Ephedra is a high degree of automation in the transliteration of the original Code to the Java Virtual Machine. It is therefore important to ensure that the generated code is correct — otherwise, the software engineer will have to proof-read and correct the code.

To achieve this goal, we decided not to do a lexical transliteration of the source code, as is done by C2J++ [65], but to perform a complete parse of the source code into an abstract semantic graph with subsequent transformation of that graph. This parsing process is very difficult to implement for a complex language like C++. To be able to focus our attention on the language conversion aspect of the migration, we decided not to design a custom parser for the transliteration tool, but to interface with the parser of an existing C++ compiler. At the time the project was started, IBM had made the interface to their VisualAge C++ compiler and related documentation available to universities, so we chose this product.

Section 10.1 provides some information on IBM VisualAge C++ and the interface of its compiler. Section 10.2 describes the tool to build class hierarchies out of type cast relationships. Section 10.3 explains the implementation of the source code transliteration tool. Considerations on future implementation changes are discussed in Section 10.4.

10.1 IBM VisualAge C++

C and C++ sources are usually broken up into two kinds of files. One kind, so called *header files*, contains the declarations of classes, data types, functions, and global variables. The second kind contains the definitions for these declarations. To ensure the correctness of a program, a developer needs to ensure that the declarations match the definitions at all times. This is traditionally done by recompiling the changed source files and all other sources files that depend on the declarations in the changed source file. Often, a lot of time is lost by recompiling code that was not affected by the changes but resides in a file that contains other code needing re-compilation. Also, header files are usually included in many other source files and get compiled during the compilation of every one of these source files.

The *incremental C++ compiler* of the IBM VisualAge C++ development environment takes a different approach: while the sources are still kept in regular files, the compiler keeps all declarations, definitions, and their dependencies in a database, the so called *CodeStore* [32, 49]. Whenever a source file changes, the compiler determines which of the declarations or definitions in that file was changed, and recompiles only this one and any other declarations or definitions that are affected by the change. The compiler thus recompiles only a minimal set of affected source code parts rather than whole source files.

IBM decided to document and publish the interface to the compiler's CodeStore, so software engineers could use the information collected by the compiler to build tools that depend on such information, such as computer aided software engineering (CASE) and computer aided software re-engineering (CARE) tools. Tools can also request the compiler to recompile selected function bodies or variable initialisations to provide further information about these. Such information can be queried at various stages of the compilation, such as before or after type analysis, optimisation, or code generation, and a complete abstract semantic graph (ASG) can be obtained from CodeStore. Tools can also insert additional passes into the compilation process, for example to insert profiling or memory management instructions into the generated code [58, 30]. Software engineers can also extend the IBM VisualAge C++ integrated de-

```
CS_Declaration *d;
for (d = code_store->globalDeclarationStore().firstDeclaration();
     d; d=d->next()) {
  if (d->declarationKind() == CS_Declaration::IsVariable)
    cout << d->fullyQualifiedName() << endl;
}
```

Figure 10.1: Querying the IBM VisualAge C++ CodeStore

velopment environment (IDE) by adding customised views or metrics of the programs compiled.

To illustrate the CodeStore interface of IBM VisualAge C++, Figure 10.1 shows the essential parts of a C++ program that uses this API to print the names of all global variables of a program.

10.2 Type Cast Analyser Tool

The normalisation step of the Ephedra Approach (Section 7.2) deals with the restructuring of data types in the original source code. There are tools for visualising class hierarchies that can be used to remove multiple inheritance from the code, and even simple Unix tools such as *grep* can be used to identify uses of multiple inheritance.

However, type casts and data structures related through type casts are difficult to identify manually and tools do not usually handle or visualise them. As traditional C compilers do not enforce the use of explicit type casts, many of them are implicit and thus cannot be detected by lexical parsing approaches. Because related data structures may reside in different source files or even subsystems, it may be difficult for software engineers to locate them. They might also not have sufficient domain knowledge to associate them.

We therefore decided to implement a tool to aid the developer in locating type casts and data structures related through type casts. As the CodeStore of IBM VisualAge C++ contains not only syntactic but also semantic information, all type casts, including implicit ones, are represented in it. Thus it was relatively easy to implement the tool using the CodeStore API, by first obtaining and then traversing the abstract semantic trees for all functions and

initialisers. We first developed a stand-alone tool and then integrated it into the IBM VisualAge C++ IDE. The tool implementation closely follows the algorithms as presented in Section 7.2.2 with the difference that Steps 1 and 2 of the algorithm are combined to increase performance. They do not have side effects and can thus safely run concurrently. The complete tool consists of about 1500 lines of code, of which about 400 provide the integration into the IDE. Figures 11.3 and 11.4 show screen shots of the tool running within the IDE.

10.3 Source Code Transliteration Tool

In the translation step of the Ephedra approach, the C/C++ source code that was improved and corrected in the normalisation step is transliterated to Java source code. Ephedra provides a tool that automates the transliteration. In the following sections, we describe the overall architecture and the experiences gained in the implementation of the tool using the IBM VisualAge C++ CodeStore API.

10.3.1 Implementation Strategy

The IBM VisualAge C++ CodeStore provides information on a C++ program through an API. Using this API, one can query the abstract semantic graph (ASG) that the C++ compiler built during the lexical and semantical analysis of the program. VisualAge can supply this graph during various stages of the compilation process, such as before and after type analysis or optimisation. To perform a correct transformation, we need the ASG to be annotated with exact type information, so we chose to query the graph after type analysis. As we wanted to generate Java source code that closely resembled the original C code, we then chose to query the graph before the optimisation phase of the compiler.

The overall strategy was to traverse the C++ ASG and to create a corresponding Java ASG that could be written out to Java source files. We therefore designed data structures for the storage of the Java ASG and an API for the

manipulation of this ASG. Since we had found VisualAge's CodeStore API to be well designed and usable during the implementation of the type cast analysis tool (Section 10.2), and to facilitate the conversion of the C++ ASG to a Java ASG, we modelled the Java ASG's API after the CodeStore API (Section 10.3.2). In contrast to the CodeStore API which was designed solely for turning source code into an ASG, we included functionality in the Java ASG API to write the ASG to Java source files.

The transliteration tool would perform the ASG conversion in two steps. In the first step, most conversions explained in the detailed conversion catalogue (Chapter 9) were to be applied. The tool would mark variables of primitive type whose address was taken in the original C++ source code for later wrapping into objects. In the second step, these previously marked variables would be wrapped. This *marking* of variables cannot be expressed in a strict Java ASG. We therefore designed our Java ASG to be extensible and permit non-Java constructs to be present in the graph. In our approach, the first step in the ASG conversion would convert the C++ ASG to a Java ASG with some non-Java constructs, while the second step would remove the non-Java constructs from the ASG to replace them by Java constructs. The final result would be a Java ASG that could be written into Java source files.

10.3.2 Java ASG and API

Using the VisualAge C++ CodeStore API as a model, we designed data structures and an API to store and manipulate an abstract semantic graph for Java source code. We created a hierarchy of connected C++ classes that reflect the structure of the Java ASG. Figure 10.2 shows the class hierarchy representing the various kinds of declarations in Java source code. The `WrapperClassDeclaration` class (drawn on shaded background) is an example of a class that does not occur in a strict Java ASG but only in the process of a conversion from C to Java. Similar class hierarchies exist for Java expressions, statements, and type descriptors.

The declaration, expression, and statement classes provide functions for printing the artifacts they represent as Java source code. The code is formatted

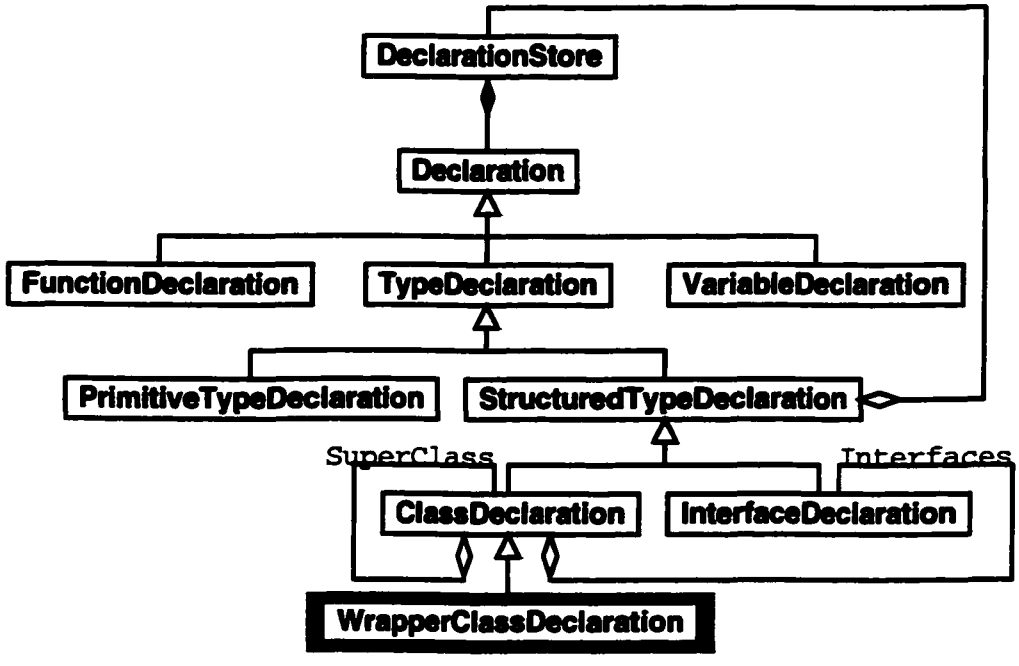

Figure 10.2: Java ASG and API — classes representing declarations

and indented in a well readable style. Comments can be attached to source code artifacts and are printed along with these artifacts when the source code is generated. The expression classes automatically add parentheses where they are required to change the default evaluation order.

10.3.3 ASG Conversion

As most of the code conversions explained in the conversion catalogue (Section 9) are simple, the implementation of the C++ ASG traversal performing the conversions appeared to be of only little complexity, and initial partial implementations of the conversions supported this assumption.

The conversion of expressions and initialisers proved to be much more difficult than expected. The representation of the C++ source code in VisualAge's ASG was sometimes counter intuitive. The ASG correctly represented type casts that the compiler had to insert into the C++ source code. In some cases, however, type casts present in the source code were discarded by the compiler even though they were of importance. Figure 10.3 shows two very similar C expressions with their ASG representations. In the first case, a cast statement is added by the compiler, in the second case, the cast statement is discarded. Details of the CodeStore ASG like these were not explained in the documentation and thus often led to confusion and errors in the conversion tool.

Another problem with VisualAge CodeStore was that, though we queried the ASG before the optimisation step of the compiler, some optimisations and code transformations had already been performed on the code. For some storage allocation and deallocation operations, the ASG did not show `new` and `delete` expressions but complex expressions involving auxiliary variables, calls to constructors and destructors, and undocumented functions of VisualAge's run-time library. These transformations were not documented in the CodeStore API, and therefore had to be reverse engineered. They apparently depend on properties of the data types involved, such as whether they have non-default constructors and destructors. They also depend on the version of the VisualAge compiler and the platform the compiler is compiling for. In

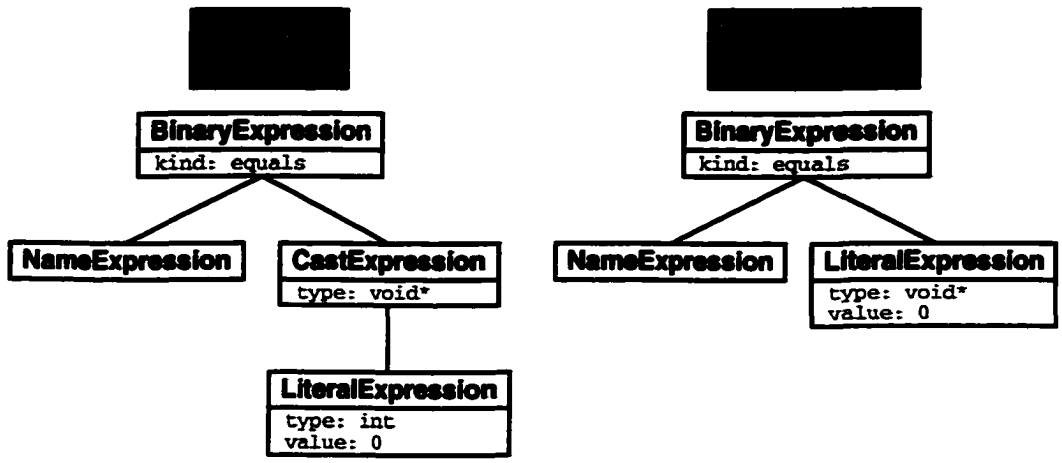

Figure 10.3: CodeStore ASG — counter intuitive representation

the conversion of the ASG, we had to revert the complex storage allocation and deallocation expressions to the original `new` and `delete` expressions before they could be transformed to Java ASG elements. To identify these complex expressions, we developed algorithms that try to match subgraphs of the C++ ASG.

Brace list initialisers were also a problem, in particular if they were nested. The VisualAge C++ compiler needs to analyse brace list initialisers to identify which parts of an aggregate they initialise. The result of the analysis is available through the CodeStore API. Unfortunately, this part of the API is poorly documented, and a lot of guess-work was necessary to extract this vital information successfully.

The VisualAge C++ Integrated Development Environment (IDE) caused many delays in the development as well. Occasionally, the incremental compiler would generate incorrect code during the compilation of the conversion tool, causing the tool to expose strange errors. We often wasted hours searching for errors that disappeared once we recompiled the code from scratch.

10.3.4 Testing

To facilitate testing the generated Java code, we implemented some optional functionality in the conversion tool. For the automation of our regression test suites, we added an option to write out all classes of a project into one Java file only that could be independently compiled. For larger projects, the conversion tool generates *make files* that can be used to compile, run, and debug the generated source code, even if it is spread over several source files.

We verified the correctness of the implementation of the code transformations using small sample C++ source files. We initially reviewed the generated Java code for correctness and later implemented automated tools to compare expected results to the actual source code generated by the conversion tool. By re-running these tests after each change to the conversion tool, we could quickly determine whether errors were introduced to the tool and eliminate them early on.

Errors we overlooked in the manual review of code or manifested themselves only in more complex programs such as the subject systems of our case studies (Chapter 11), usually resulted in exceptions being raised by the Java Virtual Machine. We were therefore able to trace their origins effectively.

10.4 Future Tool Development

The choice of IBM VisualAge C++ as front-end for the tools developed had both advantages and disadvantages. It significantly helped in the development of the tools, in particular, since no other equally well documented and powerful C++ parser was available at the time the project was started. Unfortunately, IBM has dropped support for many of the platforms VisualAge C++ used to run on, so we can only deploy the Ephedra tools on the IBM AIX operating system.

In the meantime, other C++ compilers have been used to extract information on source code [42, 15, 18], and a number of stand-alone C++ parsers have become available [21, 2]. Work has been done to standardise the output of parsers and extractors to facilitate interoperability of tools using the *Graph*

Exchange Language (GXL) [3, 22].

It therefore appears beneficial to make our tools independent of IBM VisualAge C++ and to read and write GXL instead. A new release of the tool could thus be split into the following parts:

1. *GNU protoize*
2. C/C++ to GXL fact extractor, based on VisualAge C++ or any of the other parsers available
3. type cast analyser, reading and emitting GXL
4. source code transliterator, reading and emitting GXL
5. Java source code generator, reading GXL

After the completion of the transliteration, the GXL representation of the program contains a valid Java program. It would be possible to add additional optional parts before the Java source code generation, for example to experiment with optimisations or refactorings. For example, tools could replace C library calls by Java API invocations or change error handling to leverage Java exceptions.

Parts or all of the tool could be integrated into current integrated development environments. An integration would be particularly beneficial if the IDE supported both C++ and Java, such as Eclipse [19].

10.5 Summary

In this chapter, we described the implementation of tools for the automation of Ephedra. We explained the role of the IBM VisualAge C++ CodeStore API in the development of these tools and presented some details on their overall architecture. We discussed some of the drawbacks of our implementation approach and plans for a more flexible future implementation of our tools. The next chapter describes the experiences gained in the application of our tools while conducting several case studies migrating C to Java.

Chapter 11

Case Studies — Migration of Various C Programs

To validate the Ephedra approach, we conducted several case studies by selecting typical C sources for migration to Java. To provide a usable migration environment, Ephedra offers automatic migration of frequently used C standard library functions. Many of them can be implemented quite easily and effectively using similar methods from the Java APIs. Some of them however are rather complex (e.g., the `fprintf()` function) and we decided to transliterate them from an existing C implementation rather than writing them by hand in Java. This transliteration served as the first case study.

For the second case study, we selected a small stand-alone game program that is in many aspects similar to the programs we expect to transliterate using Ephedra. It is written in K&R style C, uses pointers, function pointers, and `goto` statements, and contains little documentation.

The sample program from Figure 7.5 was used in the third case study. The run-time performance of the generated Java code was evaluated in a fourth case study.

11.1 Conversion of a Non-Trivial C Library Function

While looking for an existing, open source implementation of `fprintf()`, we found an ANSI C implementation, approximately 1,000 lines of C source code, written by Eberhard Mattes for his DOS and OS/2 port of GCC *emx* [45]. The implementation consists of one primary function and several subroutines. As the source code is written in ANSI C and does not contain any problematic type casts, the Ephedra normalisation step was not necessary.

A first attempt to transliterate the code failed. The IBM VisualAge C++ front-end could not compile the source code since it depended on some external functions, mostly for integer to string conversions, defined by emx-specific header files. We identified these functions and their purpose and decided to implement them as stubs first and to replace calls to them after the transliteration by calls to similar methods of the Java APIs.

The second transliteration attempt produced a syntactically correct Java class. Inspection of the Java sources showed that there were many casts between integer and boolean values, making the code quite difficult to read. Apparently, some of the variables and function return values used in the source assumed only boolean values, even though, because of the lack of a boolean type in C, their actual type was `char`. To optimise the code, we identified all of these variables in the original C source code and changed their type to `bool`.

After the third transliteration using Ephedra, we obtained a readable and syntactically correct Java class. After replacing the calls to stubs created after the first transliteration attempt by calls to similar Java API methods, we were able to gain confidence in the correctness of the code using a few regression tests.

Even though our new Java implementation of `fprintf()` was now working correctly, the readability of the Java source code was not perfect. `fprintf()` returns the total number of characters written as its result, and so it has to check for I/O errors in various places to guarantee that its return value is

exact. Most of the subroutines signal through their return values whether an I/O error occurred during their execution, and thus these return values have to be checked as well. These checks hide the algorithm behind the implementation and make the code difficult to read. In the original C source, these checks had been hidden using macros.

As file I/O is an ideal scenario for the use of exceptions, the Java APIs use them extensively for file access. We decided to modify the generated Java code to use exceptions to handle I/O errors. Since we do not know of existing tools to automate this process and we do not yet have sufficient experience with case studies to provide a general solution to this problem, we performed these modifications manually. In the end, we were able to replace all checks for I/O errors by a single `try/catch` statement in the primary `fprintf()` function. Moreover, the return type of all subroutines could be changed from `bool` to `void`. Through these modifications, the code size was reduced by about 25%.

The result of the transliteration process exceeded our expectations. Not only did we achieve the goal of obtaining a functional Java version of `fprintf()`, we also completed the task quickly, and the resulting Java code was more readable than the original C code. Neither did we have to derive an algorithm from the specification of the `fprintf()` function nor did we have to re-engineer it from the C implementation. There were no bugs, since the original C code had been well tested and all transformations performed were deterministic.

We do however have to note a problem with the optimisation step. Since this step is performed manually, it has to be repeated whenever the Java code is regenerated from the original C code. As we were dealing with C code that does not change any more, this was not a problem in this case study. When we are dealing with original code that still changes, we have to be careful to record all optimisations so they can be re-applied whenever the Java code is regenerated.

11.2 Conversion of a K&R-Style C Game Program

During the development of the Ephedra source code transliteration tool, we wrote various small C programs as a regression test suite to test certain aspects of the transliteration. To validate the correctness of the many applied transformations in the context of a real program that uses a large range of C language features, we chose a stand-alone *monopoly* game program, which we had used in previous studies [59, 60]. This program is written in K&R style C code and uses complex data structures, pointers, and function pointers. As such, it constitutes a solid test case for the normalisation step and the more difficult transformations of the translation step of the Ephedra approach.

11.2.1 Insertion of C Function Prototypes

In a preparatory step, we compiled and ran the original monopoly program to get an impression of its functionality. According to Ephedra's normalisation step, we then used the *GNU protoize* tool to transform the program from K&R style C to ANSI C. The monopoly program was then ready to be imported into and compiled in the IBM VisualAge C++ IDE. The first attempt to compile the program failed — the C++ compiler detected some errors that the C compiler had been unable to discover due to the lack of prototypes in the K&R style C program. Using the browsing capabilities of the IBM VisualAge C++ IDE, detection of the source of these errors was easy.

In some cases, extra parameters were passed to a function (Figure 11.1). We determined that these extra parameters were indeed not needed. We had the choice of either correcting all invocations of the function in question, or adding an extra default parameter to the function prototype that would be ignored in the function body. We decided for the latter to minimise source code changes (Figure 11.2). In retrospect, we should have chosen the other option. The addition of the superfluous parameter did not correct the errors in the code, but rather obscured them and might also confuse future developers working on the code.

```
/*
 * This routine executes a truncated
 * set of commands until a "yes or
 * "no" answer is gotten.
 */
getyn(char *prompt) {
  [...]
}

[...]
  // correct invocation;
  getyn("enter yes or no");

  // compiler detects mismatch
  char *yn[];
  getyn("enter yes or no", yn)
```

Figure 11.1: Error in monopoly program

```
/*
 * This routine executes a truncated
 * set of commands until a "yes or
 * "no" answer is gotten.
 */
getyn(char *prompt, void* = 0) {
  [...]
}

[...]
  // both invocations now correct
  getyn("enter yes or no");
  char *yn[];
  getyn("enter yes or no", yn)
```

Figure 11.2: Corrected example code

The compiler also flagged discrepancies between declared return types and actual return values of functions. None of the functions in the code had a return type explicitly specified, and thus they all defaulted to `int`. Some of the functions did not actually return a value and were flagged as incorrect by the compiler. We corrected them by giving them an explicit return type of `void` in the original code.

The compiler detected more problems in the functions responsible for loading and saving games. Apparently, the code used some Unix system calls to save and load its data area. Since we used the Windows version of the IBM VisualAge C++ compiler, these system calls were unknown to the compiler. After looking at the code, we found it difficult to understand and decided that it would be better to re-implement the load and save functions from scratch in Java rather than to emulate the Unix system calls used by their current implementations. We therefore replaced these functions by stubs. The code now compiled without errors.

The automated tool to detect problematic type casts did not locate any such casts. Since the program was written in C rather than C++, we did not have to transform any instances of multiple inheritance, and thus, the normalisation step was complete.

11.2.2 Transliteration of Source Code

The Ephedra source code transformation tool performed most of the work necessary for the translation step. The tool does not yet transform `goto` statements, so these had to be converted manually. The rest of the transformations were done automatically, and the generated Java source code then compiled correctly. In a final step, we re-implemented the load and save functions that were removed earlier in Java. The `java.io.Serializable` interface helped to make this implementation simple.

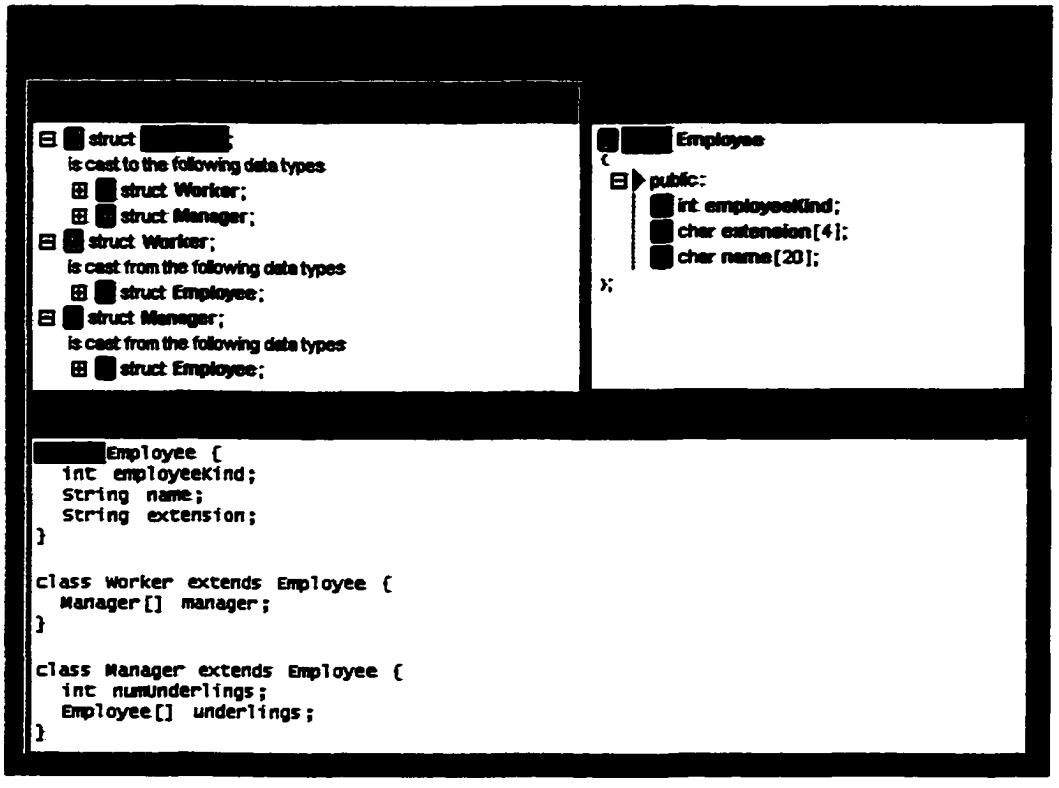

Figure 11.3: Ephedra migration tool — transliteration of data structures

11.3 Conversion of Program with Problematic Type Casts

For this case study, we converted the example program from Figure 7.5. This was done primarily to check whether the result of the automated conversion is close to the manual conversion we proposed in Section 7.2. As the program was written in the ANSI C dialect, we did not have to insert prototypes.

11.3.1 Data Type and Type Cast Analysis

Figure 11.3 shows the VisualAge IDE after importing the example program and switching to the *migration page* that presents a view generated by the type cast analysis tool. In the top left window, the page shows all data structures

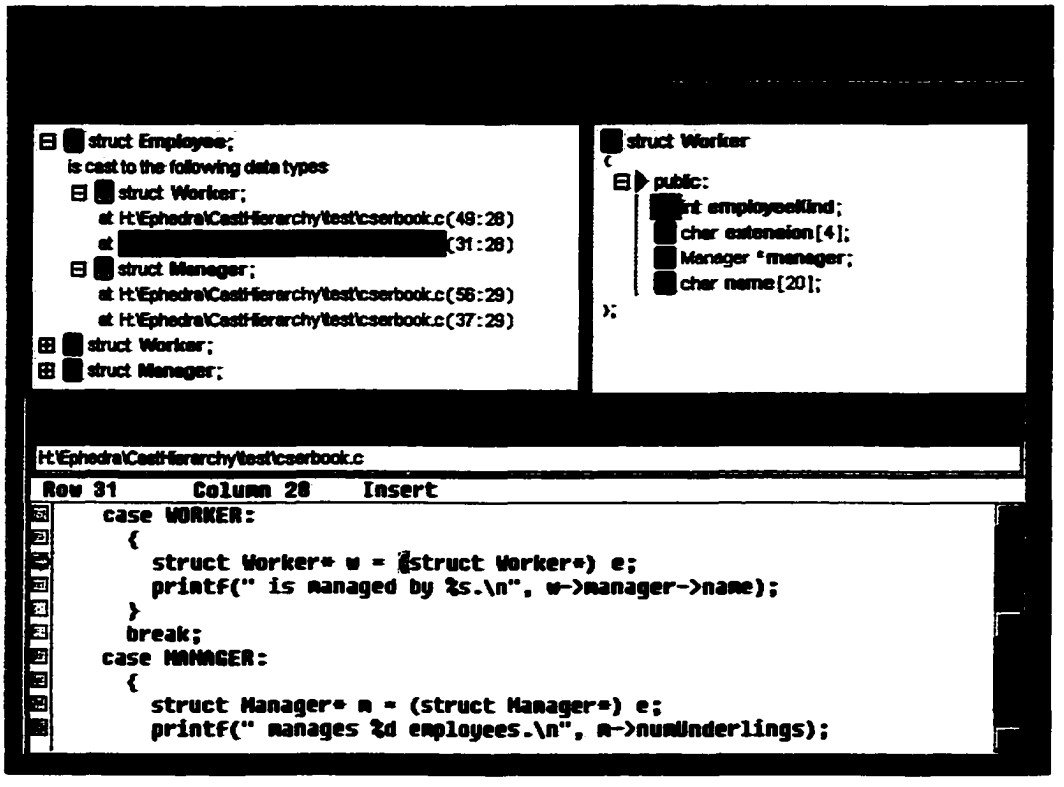

Figure 11.4: Ephedra migration tool — analysis of type Casts

that have type cast relationships to other data structures. For each of these data structures, it also shows which data structures are directly related to it. Upon the selection of one of the data structures, the top right window displays the current C language implementation of that data structure, the bottom window displays the C++ class hierarchy built using the type cast relationships.

The tool correctly identified the `underlings` field of the `Manager` data structure as unused and therefore removed it in the transliteration. Since a complete application would use this field, the sample program was modified to include a function accessing this field so it would appear in the view.

The view can also be used to determine where in the source code type casts have been applied to data types (Figure 11.4). The software engineer can use this information to locate parts of the source code that can be coded more

elegantly in C++ by exploiting the class hierarchy that has been built (as in Figure 7.6). The standard set of views and search tools in the VisualAge for C++ IDE can be used to explore the code and the usage of the data structures of interest further.

11.3.2 Transliteration of Source Code

The source code from Figure 7.6 was transliterated using the automated transliteration tool. The transliteration result is shown in Figures 11.5 and 11.6. The generated code is longer than the original source code, since default and copy constructors as well as copy operators that replicate the behaviour of their C++ equivalents had to be added to the source code.

11.3.3 Manual Optimisation of the Code

In a further step, we optimised the generated code: the `WORKER` and `MANAGER` constants clearly belong to the `Employee` class, so they were moved there from the `EmployeeType` interface. Similarly, the `show()` method operates on employees and thus should also be a member of that class. The transliteration tool transformed the C character arrays to Java byte arrays as it was the closest transliteration. Our domain knowledge tells us that these arrays always contain strings, so it is safe and reasonable to change their type to the Java `String` class. Figure 11.7 shows the code.

Repetitive manual modifications in source code, as were needed in our case study, are usually comparatively labour intensive and error prone. To relieve the software engineer from some of the burden of performing these modifications, research has been done into categorising and formalising classes of changes. *Refactorings* are source code changes that are proven not to change the behaviour of the source code if certain conditions are met [23, 52]. Tools to check whether these conditions are met in a given program and to perform the refactorings exist for Java and can help automate this optimisation step.

Figure 11.8 shows a further transformation of the example from Figure 11.7. To decrease coupling between the classes and increase coherence within the classes, the `show()` member method of `Employee` has been split; it now

```
class Employee extends Pointer {
  public int employeeKind;
  public EByte name;
  public EByte extension;
  public Employee() {
    employeeKind = 0;
    name = (EByte) EByte.opNewARRAY(20, new EByte());
    extension = (EByte) EByte.opNewARRAY(4, new EByte());
  }
  public Employee(Employee src) {
    employeeKind = src.employeeKind;
    name = (EByte) EByte.opNewARRAY(src.name);
    extension = (EByte) EByte.opNewARRAY(src.extension);
  }
  public static Employee operatorASSIGN(Employee _this, Employee src) {
    _this.employeeKind = src.employeeKind;
    _this.name.opAssignARRAY(src.name);
    _this.extension.opAssignARRAY(src.extension);
    return _this;
  }
};
class sample {
  public static void showEmployee(Employee e) {
    stdio.printf(EString.toBytes("%s\0"),
 new Ellipsis(new MultiPointer(e.name)));
    switch (e.employeeKind) {
    case EmployeeType.WORKER:
      {
        Worker w = (Worker) e;
        stdio.printf(EString.toBytes(" is managed by %s.\n\0"),
     new Ellipsis(new MultiPointer(w.manager.name)));
      }
      break;
    case EmployeeType.MANAGER:
      {
        Manager m = (Manager) e;
        stdio.printf(EString.toBytes(" manages %d employees.\n\0"),
     new Ellipsis(new EInt(m.numUnderlings)));
      }
      break;
    }
  }
};
```

Figure 11.5: Transliteration of code from Figure 7.6 (Part 1 of 2)

```
interface EmployeeType {
  public static final byte WORKER = 0;
  public static final byte MANAGER = 1;
};
class Worker extends Employee {
  public Manager manager;
  public Worker() {
    manager = null;
  }
  public Worker(Worker src) {
    super(src);
    manager = src.manager;
  }
  public static Worker operatorASSIGN(Worker _this, Worker src) {
    Employee.operatorASSIGN((Employee) _this, src);
    _this.manager = src.manager;
    return _this;
  }
};
class Manager extends Employee {
  public int numUnderlings;
  public Employee underlings;
  public Manager() {
    numUnderlings = 0;
    underlings = null;
  }
  public Manager(Manager src) {
    super(src);
    numUnderlings = src.numUnderlings;
    underlings = src.underlings;
  }
  public static Manager operatorASSIGN(Manager _this, Manager src) {
    Employee.operatorASSIGN((Employee) _this, src);
    _this.numUnderlings = src.numUnderlings;
    _this.underlings = src.underlings;
    return _this;
  }
};
```

Figure 11.6: Transliteration of code from Figure 7.6 (Part 2 of 2)

```
class Employee {
  static final int WORKER = 0;
  static final int MANAGER = 1;

  int employeeKind;
  String name;
  String extension;

  void show() {
    System.out.print(name);
    switch (employeeKind) {
      case WORKER:
        {
          Worker w = (Worker) this;
          System.out.print(" is managed by ");
          System.out.print(w.manager.name);
          System.out.println(".");
        }
        break;
      case MANAGER:
        {
          Manager m = (Manager) this;
          System.out.print(" manages ");
          System.out.print(m.numUnderlings);
          System.out.println(" employees.");
        }
        break;
    }
  }
}
class Manager extends Employee {
  int numUnderlings;
  Employee[] underlings;
}
class Worker extends Employee {
  Manager manager;
}
```

Figure 11.7: Manual optimisation of code from Figures 11.5 and 11.6

```
class Employee {
  String name;
  String extension;

  void show() {
    System.out.print(name);
  }
};

class Manager extends Employee {
  int numUnderlings;
  Employee[] underlings;

  void show() {
    super.show();
    System.out.print(" manages ");
    System.out.print(numUnderlings);
    System.out.println(" employees.");
  }
}

class Worker extends Employee {
  Manager manager;

  void show() {
    super.show();
    System.out.print(" is managed by ");
    System.out.print(manager.name);
    System.out.println(".");
  }
}
```

Figure 11.8: Further optimisation of code from Figure 11.7

contains only the code common for all employees. The new `show()` member methods of `Manager` and `Worker` handle the specialised cases.

Further analysis of the code reveals that the `employeeKind` variable is a constant for any given class, and its value has no meaning except for differentiating the distinct classes. Therefore, the Java comparison operator `instanceof` can be used instead of the `employeeKind` variable (it is not needed in the sample code, but it may be needed in other parts of the program) [29].

11.4 Conversion of two CPU Intensive Algorithms

To evaluate not only the correctness but also the efficiency of the generated code, we chose to convert two CPU intensive programs. These programs are part of *Sgraph* and implement *Spring* and *Sugiyama* graph layout algorithms [31, 4]. The programs work as filters (i.e., they read their input from standard input and write the results to standard output). The graphs they operate on are stored in a proprietary graph format. Lex and Yacc are used to parse this format.

As the graph format is rather simple and we did not want to transform Lex or Yacc in this case study, we decided to re-implement the parser without Lex and Yacc for testing purposes. This new parser was implemented in about 100 lines of C code.

The Rigi graph editor uses the Spring and Sugiyama stand-alone programs for generating graph layouts [47, 51]. As the layout programs did not come with test suites, we used this graph editor to generate a few test cases.

11.4.1 Insertion of C Function Prototypes

As the programs were written in K&R style C, we used *GNU protoize* to transform them to ANSI C. While compiling the programs using the VisualAge C++ IDE, the compiler found a few problems. As in the monopoly program, the compiler noticed discrepancies between declared return types and actual

return values of functions. Some of the functions in the code had no return type explicitly specified, and thus their return types defaulted to `int`. As they did not actually return a value, they were flagged as incorrect by the compiler. We corrected them by giving them an explicit return type of `void`. The compiler also encountered an incorrect function call: an extra parameter had been specified.

11.4.2 Data Type and Type Cast Analysis

The Sgraph data structures used a *union* to save attributes of the graph. These unions could contain either one integer or an untyped pointer. According to the Ephedra approach, this union should be converted to a hierarchy of classes with an abstract base class and concrete subclasses for the different types of attributes.

Since the fields of the union were rather small, we decided to take a different approach and transform the union to a regular data structure with one field each for the integer and pointer value. We felt that the little extra amount of memory used in this approach was preferable to the loss in performance that would result from type casts or calls to virtual functions when strictly following the Ephedra approach.

Ephedra's source code transliteration tool performs this transformation automatically for all unions that are still present in the code. Thus, we did not have to make any manual changes to the code.

11.4.3 Transliteration of Source Code

The transliteration of the code went smoothly, and no major problems were encountered. The programs used some C standard library functions, which had not been implemented yet. Once they had been implemented, the Spring layout program performed correctly. There were minor differences in the output of the original and transformed program. They were caused by differences in how C and Java handle floating point numbers.

The Sugiyama layout program aborted with a null pointer exception while printing the results. Investigating the problem revealed a severe error in the

original C program: some memory was released before it was last used. As the program did not allocate any new dynamic storage after this release of storage, the storage was not overwritten and the C program still functioned properly. In a different environment, the program might have produced incorrect results or caused harm to other parts of the system. In the Java transliteration, the de-allocation of memory is emulated by setting references to storage that is to be de-allocated to `null`. A subsequent access to those references therefore resulted in a null pointer exception, and thus revealed the error that was hidden in the original C program.

11.5 Summary

In this chapter, we presented several case studies on the conversion of example C programs stemming from various application domains to Java. We explained the steps necessary to perform the migration and assessed the usefulness of the tools we implemented for this purpose. We are satisfied with the reliability and the level of automation achieved by the Ephedra tool set. We are particularly pleased that Ephedra helped to locate some serious errors in the subject systems of our case studies. While this chapter focused on the conversion process, the following chapter considers the conversion result by assessing the quality of the code generated during the conversion.

Chapter 12

Quality of Generated Code

We evaluated the quality of the code generated by the Ephedra source code conversion tools by assessing several properties of the code: its maintainability, as characterised by its readability and conformance with Java coding standards, and its efficiency.

12.1 Readability

From a subjective point of view, we are generally pleased with the readability of the code generated by the Ephedra conversion tool. It compares very favourably to its only competitor for large scale automated translation of source code, C2J [50]. To obtain a more objective measure of the readability of the generated code, we need to gather opinions and experiences of other developers. For this purpose, we have created a Web site that allows Internet users to perform source code translations with the Ephedra tools on their own systems. As this project has only recently been started, we have not been able to gather results yet. We are also releasing stand-alone versions of the Ephedra tools to interested parties.

As expected, the two different mapping schemes for pointers we developed exhibit both advantages and disadvantages with respect to the readability of the code they generate. By converting most C pointers to Java references, the first mapping produces very well readable code where pointers in the C code are

used to implement linked data structures or for call-by-reference parameters. However, code responsible for allocation and traversal of arrays suffers because the mapping does not employ standard Java arrays.

The opposite situation applies to the second mapping approach. Since it maps C arrays to Java arrays, array definitions and traversals are very readable. Pointer arithmetic is clearly recognisable because of the dedicated pointer classes. The need for these classes and the creation of pointer objects in various places makes the code less readable where pointers are used simply for the implementation of linked data structures or call-by-reference parameters.

Unfortunately, we do not currently see a way to reconcile the two approaches to mapping pointers to achieve a unified approach that provides optimal code in all situations. The software engineers will have to decide which mapping approach to employ depending on the nature of the program to be converted by choosing the appropriate executable of the Ephedra conversion tool.

12.2 Conformance, Integration

One of the primary goals of Ephedra was to enable the integration of large volumes of source code written in C into mainstream Java programs. With Novosoft's C2J translator, this is difficult, because C2J's storage management scheme is incompatible with that of regular Java programs.

Ephedra uses the regular Java data types and classes, so the generated code interfaces nicely with manually written Java source code. Where arrays or pointers are part of the interfaces to the generated code, the software engineer should have a basic knowledge of the Ephedra approach, but a deep understanding is not required.

Classes generated by Ephedra are not currently designed for serialisation using the standard Java serialisation techniques. As C data structures are not designed with the Java serialisation capability in mind, they might yield unexpected results, in particular where these data structures contain pointers or arrays. Ephedra supports a distinct serialisation strategy that closely models the behaviour of serialisation in C using C standard library functions.

Ephedra's approach is not inherently incompatible to Java's serialisation facility, so a software engineer can add the required functionality after reviewing the source code to ensure that the desired effects are achieved.

12.3 Performance

Our case study of the conversion of two CPU intensive algorithms (Section 11.4) allowed us to evaluate the efficiency of the code generated by the Ephedra conversion tools.

For small input files, the transformed versions of the Spring and Sugiyama graph layout algorithms performed worse than the original program. It was quickly determined that this is mainly due to the long start-up time of Java programs.

To analyse the performance of the algorithms on larger graphs, we created a graph of 560 nodes for the Spring algorithm, and a graph of 270 nodes for the Sugiyama algorithm. The difference in execution time becomes smaller, but is still significant. The performance also depends on the pointer mapping scheme used (Chapter 8). Code converted using the first mapping scheme was about five times slower than the original C code for the Spring algorithm, and about ten times slower for the Sugiyama algorithm. In our tests, the second mapping scheme produced code that ran about 30% faster than code produced with the first mapping scheme. The C code had been compiled with run-time optimisations enabled.

To gain a better understanding of the causes of the loss of efficiency, we measured the time the converted programs used for I/O operations (loading and saving the graph files) and the execution of the algorithms. We noticed that the Java programs needed a significantly longer time for I/O than the C programs. This was no big surprise: the Java programs need to pass the data from Ephedra's run-time library through the Java API's to system functions, while the C programs could access these system functions directly. Also, the *Just-In-Time* (JIT) compilers built into the Java Virtual Machines we used were apparently unable to compile or optimise the code performing the I/O; a Java to machine code compiler that produced overall worse results than the

Java Virtual Machines produced significantly better performing code for file I/O.

The bad performance of I/O also explains the difference in the efficiency of the converted Spring and Sugiyama algorithms. The graph used to evaluate the Spring algorithm was roughly twice as big as that of the Sugiyama algorithm. However, the Spring algorithm has a significantly higher complexity than the Sugiyama algorithm, so the time the Spring algorithm spent with I/O was a smaller fraction of the total execution time than for the Sugiyama algorithm. Both algorithms were thus only about three to five times slower than the original C code, depending on the Java Virtual Machine and the pointer mapping scheme used.

As we could verify in a small additional case study using a heap sort algorithm, converted programs using little or no pointer arithmetic and mostly fundamental data types can perform almost as well as the original C programs. In our tests, the Java code ran faster than C code compiled without optimisations enabled, and about half as fast as C code compiled with optimisations enabled.

The Sugiyama algorithm was a good example to illustrate the effect of an inefficient conversion of exceptions from C++ to Java. If a straightforward transformation (Section 9.14) is used with the first pointer mapping scheme, the Sugiyama algorithm requires enormous amounts of memory. For the example graph of 270 nodes, the algorithm required 80 to 160 MB of memory instead of the original 2 MB and ran about two to three times slower than with the optimised exception transformation. Some Java Virtual Machines were not able to execute the test case at all due to this inefficient conversion of exceptions, while they had no problems running the test cases with the optimised exception transformation.

12.4 Summary

In this chapter, we evaluated the quality of the Java source code resulting from a conversion of a C program using the Ephedra approach. According to the goals set out in 5, we analysed the converted code with respect to

its readability and performance. To assess how well the migrated code can be integrated into mainstream Java programs, we checked its conformance with Java coding standards. We are pleased to have developed a conversion approach that does not have major negative impacts on any of these criteria. The following chapter ends this dissertation by summarising the main problems and our contributions to their solutions as well as open problems and future work.

Chapter 13

Conclusions and Future Work

13.1 Summary

Since electronic commerce over the Internet plays a prominent role in today's business environment, customers expect integrated Web-based services historically provided by legacy information systems. One way to provide these services to the users quickly and effectively is to integrate the legacy C code of such information systems with newly developed Java programs. The use of Java as the programming language also opens a software project to a larger group of developers, because universities nowadays mostly teach Java in their undergraduate curricula.

In this dissertation, we presented several integration strategies by discussing their characteristics, strengths, and weaknesses. We also introduced the Ephedra approach for migrating C programs to Java using the Ephedra software migration environment. Several different migration case studies were used to evaluate this approach to migrating C/C++ code to Java.

13.2 Major Contributions

The survey of current integration and migration studies presented in Chapter 4 documents and evaluates the approaches for integrating C/C++ source code into Java programs that existed prior to the development of Ephedra. It pro-

vides valuable information on the strengths and weaknesses of these approaches and explains in which scenarios they yield best results.

Part II of this dissertation presents Ephedra, a new approach for migrating C/C++ source code to Java. It was designed with specific goals in mind, in particular the ease of integration of the converted source code into mainstream Java programs. While these design goals were met, the approach is general enough to permit the migration of a wide range of programs. In particular, we developed two strategies for mapping C pointers to Java, that are targeted towards different kinds of applications. In contrast to previous C to Java migration strategies, Ephedra provides a structured approach and a manual that clearly describes the conversion steps and their underlying rationale.

In Chapter 10, we showed that the Ephedra approach can be automated by implementing a migration environment according to this approach. Some parts of this environment are relatively self contained while providing sufficient functionality to be used in other software engineering tool-sets. In particular, the type cast analysis tool (Section 10.2) and the Java ASG API (Section 10.3.2) should prove very useful in the construction of reverse and re-engineering tools.

13.3 Future Work

Some of the case studies showed that debugging the generated code seems to be easier than debugging C code compiled to native machine code. The run-time checks performed by the Java Virtual Machine help greatly in locating faults that may go undetected in C programs under certain conditions. It may be possible to decrease development time and cost for new C/C++ programs by transliterating them to Java and debugging them using the Java Virtual Machine.

We are pleased with the Java code generated by the transliteration tool. The generated code seems to be correct and reasonably efficient. If certain guidelines are followed in the development of new C++ code (e.g., portable type casts, single inheritance), no manual intervention will be required to migrate such C++ code to Java. Moreover, it is quite likely that faults detected in the Java code are faults in the original C code. Debuggers could be instru-

mented to hide the intermediate Java code from the developers and make it appear as if they were debugging the original C/C++ code. A study comparing the development cost when using this approach, as opposed to the cost of traditional development, might be interesting. This might be a viable and useful option for C development environments. We did not anticipate this application when we began our research.

The monopoly application was used in previous case studies to evaluate the usability of re-engineering tools in a structured experiment [59, 60]. This experiment could be repeated to determine whether the source code migration helps or hinders the software engineer in understanding the program. Such studies would also help us to evaluate how readable and maintainable the generated code is. As it is difficult and not very objective to judge the readability of code by simply looking at the code, we have consciously avoided conclusions about the readability of the generated code in this dissertation. Since we are familiar with the conversions applied in these case studies, we rate the code as more readable compared to Java developers who are not familiar with the code.

In Section 10.4, we laid out a strategy for a new and more flexible implementation of the Ephedra conversion tool. In addition to the structural changes, this new implementation should perform conversions that currently have to be done manually, such as the elimination of `goto` statements. For the tool to be usable in large and critical projects, an important requirement is the preservation of the original comments in the source code. The current implementation of the tool already contains some provisions for representing comments in the Java source code. We need to develop strategies for identifying to which token a particular comment in the source code really refers. There is some related work that might prove useful for this purpose [7].

As discussed in Section 7.3.1, C macro definitions pose a problem for our conversion tool and for many automated reverse engineering tools, because they cannot be expressed as part of the C grammar. The development of an automated method for converting certain macro definitions to C++ language constructs such as constants and inline functions would be a notable contribution to the research community.

Bibliography

[1] American National Standard X3.159-1989.

[2] Datrix Home Page. http://www.casi.polymtl.ca/casibell/datrix/, November 2001.

[3] GXL Home Page. http://www.gupro.de/GXL, November 2001.

[4] Sgraph Standalone Programs. http://www.infosun.fmi.uni-passau.de/GraphEd/download/sgraph-standalone/, December 2001.

[5] Peter Aiken, Ojelanki K. Ngwenyama, and Lewis Broome. Reverse-engineering: New systems for smooth implementation. *IEEE Software*, pages 36–43, March 1999.

[6] Keith H. Bennett. Do Program Transformations Help Reverse Engineering? In *Proceedings of the International Conference on Software Maintenance (ICSM) 1998*, Washington, DC, November 1998.

[7] Peter L. Bird and F. Andy Seidl. Reengineering Legacy C and C++ Applications, June 1998. Whitepaper, Genitor Corp.

[8] Maria Istela Cagnin, Rosângela Penteado, Rosana T. V. Braga, and Paulo C. Masiero. Reengineering using Design Patterns. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 118–127, Brisbane, Queensland, Australia, November 2000.

[9] Gerardo Canfora, Aniello Cimitile, and Malcolm C. Munro. An improved algorithm for identifying reusable objects in code. *Software Practice and Experiences*, 26(1):24–48, 1996.

[10] Oscar Cepeda. *Open32 Developer API Extensions for OS/2 Warp.* IBM Corporation, 1996.

[11] Aniello Cimitile, Andrea De Lucia, Giuseppe A. Di Lucca, and Anna Rita Fasolino. Identifying Objects in Legacy Systems. pages 138–147, May 1997.

[12] Internet Software Consortium. Internet Domain Survey, January 2001. http://www.isc.org/ds/hosts.html, April 2001.

[13] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Software Engineering by Source Transformation — Experience with TXL. In *Proceedings of the 1st International IEEE Workshop on Source Code Analysis and Manipulation*, pages 168–178, Florence, Italy, November 2001.

[14] Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. The Simula67 Common Base Language. Technical report, Oslo, Norway, October 1970.

[15] Thomas R. Dean, Andrew J. Malton, and Richard C. Holt. Union Schemas as a Basis for a C++ Extractor. In *Proceedings of the Eight Working Conference on Reverse Engineering*, pages 59–67, Stuttgart, Germany, October 2001. http://swag.uwaterloo.ca/~cppx/Union.pdf, November 2001.

[16] Erik D. Demaine. Converting C Pointers to References. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, 1998.

[17] Adolfo Di Mare. C Iterators. Technical report, Universidad de Costa Rica, April 1999. http://www.di-mare.com/adolfo/p/c-iter.htm.

[18] James Michael DuPont. GCC XML Node Introspector. http://introspector.sourceforge.net/, March 2002.

[19] Eclipse.org. Eclipse Web Site. http://www.eclipse.org.

[20] Hakan Erdogmus and Oryal Tanir, editors. *Advances in Software Engineering: Topics in Comprehension, Evolution, and Evaluation.* Springer-Verlag New York, Inc., December 2001.

[21] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss, and Mikko Tarkiainen. Columbus — Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of SPLST 2001*, pages 16–27, Szeged, Hungary, June 2001. http://ferenc.rgai.hu/research/ferencr_columbus.pdf, November 2001.

[22] Rudolf Ferenc, Susan E. Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a Standard Schema for C/C++. In *Proceedings of the Eight Working Conference on Reverse Engineering*, pages 49–58, Stuttgart, Germany, October 2001. http://ferenc.rgai.hu/research/ferencr_scheme.pdf, November 2001.

[23] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman, Inc., 1999.

[24] Free Software Foundation (FSF). GCC Home Page. http://www.gnu.org/software/gcc/gcc.html, October 2001.

[25] Harald Gall and René R. Klösch. Finding objects in procedural programs: an alternative approach. In *Proceedings of 2nd IEEE Working Conference on Reverse Engineering*, pages 208–216, Toronto, Canada, July 1995. IEEE Computer Society Press.

[26] Joseph George and Bradley D. Carter. A strategy for mapping from function-oriented software models to object-oriented software models. *ACM Software Engineering Notes*, 21(2):56–63, March 1996.

[27] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman, Inc., 1983.

[28] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley Longman, Inc., 1996.

[29] James Gosling, Bill Joy, and Guy Steele. *Type Comparison Operator instanceof*, chapter 15.19.2. In [28], 1996.

[30] Jennifer Hamilton. Montana Smart Pointers: They're Smart, and They're Pointers. In *Proceedings of the Conference on Object-Oriented Technologies and Systems*, pages 21–39, Portland, OR, 1997.

[31] Michael Himsolt. Sgraph Programmer's Manual, 1993.

[32] Michael Karasick. The Architecture of Montana: An Open and Extensible Programming Environment with an Incremental C++ Compiler. In *Proceedings of the Conference on Foundations of Software Engineering*, Orlando, FL, November 1998.

[33] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[34] Kostas Kontogiannis, Johannes Martin, Kenny Wong, Richard Gregory, Hausi A. Müller, and John Mylopoulos. Code Migration Through Transformations: An Experience Report. In *Proceedings of CASCON '98*, pages 1–13, Toronto, ON, 1998.

[35] Chris Laffra. C2J, a C++ to Java translator. (web site no longer available).

[36] M. M. Lehman. On Understanding Laws, Evolution and Conversation in the Large Program Life Cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[37] M. M. Lehman. Programs, Life Cycles and Laws of Software Evolution. *Proceedings of the IEEE Special Issue on Software Engineering*, 68(9):1060–1076, September 1980.

[38] Sheng Liang. *The Java Native Interface, Programmer's Guide and Specification*. Addison-Wesley Longman, Inc., 1999.

[39] Frank Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, Inc., 1999.

[40] Sying-Syang Liu and Norman Wilde. Identifying objects in a conventional procedural language: an example of data design recovery. In *Proceedings of IEEE Conference on Software Maintenance*, pages 266–271, San Diego, CA, November 1990. IEEE Computer Society Press.

[41] Panos E. Livadas and Theodore Johnson. A new approach to finding objects in programs. *Journal of Software Maintenance: Research and Practice*, 6:249–290, 1994.

[42] Andrew J. Malton. CPPX Home Page. http://www.swag.uwaterloo.ca/~cppx/, November 2001.

[43] Andrew J. Malton. The Migration Barbell. First ASERC Workshop on Software Architecture, August 2001. http://www.cs.ualberta.ca/~kenw/conf/awsa2001/papers/malton.pdf, November 2001.

[44] Johannes Martin and Hausi A. Müller. *Discovering Implicit Inheritance Relations in Non Object-Oriented Code*, chapter 11. In Erdogmus and Tanir [20], December 2001.

[45] Eberhard Mattes. emx 0.9d (GCC and tools for DOS & OS/2). http://archiv.leo.org/pub/comp/os/os2/leo/gnu/emx+gcc/, October 2001.

[46] Hausi A. Müller. Understanding Software Systems Using Reverse Engineering Technologies — Research and Practice. Tutorial presented at the 18th International Conference on Software Engineering, March 1996. http://www.rigi.csc.uvic.ca/UVicRevTut/UVicRevTut.html.

[47] Hausi A. Müller and Karl Klashinsky. Rigi — A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, 1988.

[48] Andrew Myers, Barbara Liskov, Jed Liu, Grant Wang, and Nick Mathewson. PolyJ — Java with Parameterized Types. http://www.pmg.lcs.mit.edu/polyj/, March 2002.

[49] Lee R. Nackman. CodeStore and Incremental C++. *Dr. Dobb's Journal*, pages 92–95, December 1997.

[50] Novosoft. C2J — C to Java translator. http://www.novosoft-us.com/NS2B.nsf/w1/C2J, September 2001.

[51] University of Victoria. Rigi Web Server. http://www.rigi.csc.uvic.ca, June 2001.

[52] William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992. http://st.cs.uiuc.edu/pub /papers/refactoring/opdyke-thesis.ps.Z.

[53] Prashant Patil, Ying Zou, Kostas Kontogiannis, and John Mylopoulos. Migration of Procedural Systems to Netowkr Centric Platforms. In *Proceedings of CASCON 1999*, pages 68–82, Toronto, ON, November 1999.

[54] Pizza Group. GJ — A Generic Java Language Extension. http://www.research.avayalabs.com/user/wadler/pizza/gj/, March 2002.

[55] Chris Ricker. Linux 2.0.31 Change Notes, August 1997.

[56] NUA Internet Services. NUA Internet How Many Online. http://www.nua.ie/surveys/how_many_online/index.html, March 2002.

[57] Harry Sneed. Validating Functional Equivalence of Reengineered Programs via Control Path, Result, and Data Flow Comparison. *Software Testing, Verification and Reliability*, 4(1):33–44, March 1994.

[58] Danny Soroker, Michael Karasick, John Barton, and David Streeter. Extension Mechanisms in Montana. In *Proceedings of the 8th IEEE Israeli Conference on Computer Systems and Software Engineering*, Herzliya, Israel, June 1997. IEEE Computer Society Press.

[59] Margaret-Anne D. Storey, Kenny Wong, Philip Fong, David S. Hooper, Kory Hopkins, and Hausi A. Müller. On Designing an Experiment to Evaluate a Reverse Engineering Tool. In *Proceedings of the 3rd Working Conference on Reverse Engineering*, Monterey, CA, November 1996.

[60] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 12–21, Amsterdam, Holland, October 1997.

[61] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman, Inc., Reading, MA, 1986.

[62] Andrey A. Terekhov. Automating Language Conversion: A Case Study. In *Proceedings of the International Conference on Software Maintenance (ICSM) 2001*, pages 654–658, Florence, Italy, November 2001.

[63] Andrey A. Terekhov and Chris Verhoef. The Realities of Language Conversions. *IEEE Software*, pages 111–124, November 2000.

[64] The Java Community Process (CP) Programm. JSR14: Add Generic Types to the Java Programming Language. http://jcp.org/jsr/detail/014.jsp, March 2002.

[65] Ilya Tilevich. Translating C++ to Java. *First German Java Developers' Conference Journal.* http://sol.pace.edu/~tilevich/c2j.html.

[66] Robert Tolksdorf. Languages for the Java VM. http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html, November 2000.

[67] Trent Waddington. Java Backend for GCC. http://archive.csee.uq.edu.au/~csmweb/uqbt.html#gcc-jvm, November 2000.

[68] Tracy Wen. Translating C++ to Java: Resolution of Multiple Inheritance. Master's thesis, University of Victoria, 2000.

[69] Kazuki Yasumatsu and Norihisa Doi. SPiCE: A System for Translating Smalltalk Programs into a C Environment. *IEEE Transactions on Software Engineering*, 21(11):902–912, 1995.

[70] Alexander S. Yeh, David R. Harris, and Howard B. Reubenstein. Recovering abstract data types and object instances from a conventional

procedural language. In *Proceedings of 2nd IEEE Working Conference on Reverse Engineering*, pages 227–236, Toronto, Canada, July 1995. IEEE Computer Society Press.