SUPPORTING LANGUAGE EXTENSION AND SEPARATE COMPILATION BY MIXING JAVA AND BYTECODE

Lennart Kats

MSc Thesis
INF/SCR-07-02
Utrecht, August 2007

Center for Software Technology,
Institute of Information and Computing Sciences,
Utrecht University,
P.O. Box 80.089, 3508 TB,
Utrecht, The Netherlands.

Internal advisor: Prof. Dr. S.D. Swierstra
External advisors: Dr. E. Visser
Drs. M. Bravenboer

Abstract

Language extensions, such as embedded domain-specific languages, are often implemented by assimilating (rewriting) the extended language constructs to the host language. The result can then be compiled by a standard compiler. This approach is limited by the host language, which may not be designed with code generation in mind. An example of this is Java, which provides insufficient protection against name capture of host language identifiers, and does not provide the same low-level primitives that exist in the underlying Java Virtual Machine. For example, it does not provide equivalents for a *jump* or *jump subroutine* instruction, unbalanced synchronization, stack manipulation, or specification of debugging information. Code generated from a language that does not match Java's structure can therefore require inefficient or laborious alternatives instead.

We propose a new open compiler model to provide generators direct access to the underlying compiled code. With conventional open compilers, leveraging the bytecode-generating back-end is an intricate process, requiring adaptations tangled throughout the system. The result is hard to develop, understand, and maintain. By providing a mixed source language of Java and the underlying bytecode instruction language, we can provide access to the back-end at the source-level. Compiled instructions can be used in place of statements or expressions, which can aid language extensions but also applications of separate compilation. For example, it can simplify aspect weavers by enabling direct composition of source code aspects with compiled classes, or vice versa. As such, we also introduce a Java traits compiler that allows operation on classes and traits in both source and compiled form.

Acknowledgments

I would first like to thank my supervisors at Delft University of Technology, Eelco Visser and Martin Bravenboer, for their support and for creating the tools that made my work possible. Eelco Visser has given me very helpful suggestions and advice throughout this project, and provided me with the freedom to pursue directions that I found interesting. He also introduced me into the wonderful world of program transformations. I thank Martin Bravenboer, my daily supervisor, for his ideas and tremendous support, which had a major influence on this thesis. He devoted a lot of time to the various revisions I confronted him with, and provided extensive support with Dryad, the compiler front-end he created that formed the basis of my implementation. I really appreciate all of his feedback and guidance during this project. I also wish to express my sincere thanks and appreciation to Doaitse Swierstra, my internal advisor at Utrecht University, for his helpful comments and constructive suggestions.

Finally, special thanks are due to my parents and sister Merel, for their support and encouragement during my studies, thesis project, and other endeavors.

Contents

Chapter 1

Introduction

"Inside every large program is a small program struggling to get out"

– C. A. R. Hoare, Efficient Production of Large Programs (1970)

General-purpose programming languages offer numerous features that make them applicable to a broad scope of application domains. However, as program complexity increases, such languages lack the high-level formalisms required to adequately cope with this complexity. Through the introduction of **language extensions**, it is possible to add to the expressiveness of a language. By targeting a specific problem domain, language features can be introduced that allow for more concise, maintainable programs aimed at such a domain. Language extensions allow for static verification of correctness, security, and style constraints, specific for the associated domain. They also provide the opportunity for domain-specific program optimizations.

Domain-Specific Languages (DSLs) are languages aimed entirely at a specific domain, offering specialized functionality and syntax. Using DSLs, larger, more complex applications within a domain are easier to implement, and require less effort to develop and maintain. For example, DSLs exist for XML and text processing, and offer a very concise, natural way to program for such a domain. However, as DSLs lack the general capabilities of a general-purpose language (e.g., for creating a user interface), they are rarely used by themselves. Therefore, DSLs are often implemented as a language *extension*, in the form of an **embedded DSL**.

Key for implementing language extensions is the construction of **compilers** that generate code based on high-level program definitions. Currently, developers infrequently construct language extensions and DSLs, as building and maintaining such a compiler is onerous. If generative programming is to become a staple ingredient of the software engineering process, the construction of such tools should be as automated and easy as possible.

For implementing language extensions, it is clearly desirable to build upon the basis of an existing compiler for the base language. Extensible compilers are designed with this purpose in mind, and provide an extensible front-end: a parser and semantic analysis (type-checker) implementation that may be extended and reused for language extensions [34,10,66,51,52]. The acquired type information may be used in the extension, for error detection or to generate a properly-typed program, which is then further compiled by the compiler back-end. Only few compilers offer an extensible *back-end* [66,10,56], which emits the input program to a low-level language such as assembly or Java bytecode. Compilers that offer this functionality require adding a lot of additional code to different components of the compiler, to support even small extensions using the back-end. Such additions tend to get tangled throughout the various compilation stages, require profound knowledge of the compiler's internal structure, and create a large dependency on details of the base compiler's implementation.

Various languages and language extensions depend on implementation in a compiler back-end, as they need to directly generate code in the low-level output language. This enables the use of primitives not available in the base language, or is done for performance reasons [49,10]. Sometimes it is also applied for the purpose of separate compilation. For example, an **aspect-oriented programming language** allows separation of concerns by inserting separately defined code into existing classes [31,32,40]. Aspect weaving compilers manipulate the bytecode of existing, separately compiled classes to weave in additional compiled code, therefore making use of an existing compiler back-end [32].

An alternative to integrating with an existing compiler is developing a **preprocessor**. This is the starting point for many language extensions and DSLs. Preprocessors are stand-alone applications that rewrite source code to a high-level target language, such as Java, and require further compilation with an external compiler. This is a convenient, simple approach that takes advantage of the familiarity and high-level abstractions of the target language, without requiring integration with the target compiler [43,44,45,48]. Software composition systems are often implemented using this approach, as it requires significantly less effort than integrating with and extending a compiler. For example, existing compilers for the **traits language extension** operate as a preprocessor [6,23,24,26,37]. Unfortunately, they thereby forfeit operation on compiled code.

Java is a popular platform for application development, as it is a mature, widely used and available, cross-platform environment. Java programs can leverage the extensive Java standard library [33] as well as an overwhelming number of third-party libraries. The Java *language* is a strongly typed, general-purpose language. While it was not directly designed with the purpose of code generation in mind, it is applied for the implementation of various DSLs and language extensions [31,36,48,49,10]. The language's philosophy does not always match with that of a given language extension, or even with the notion of generating code in general. For example, Java allows identifiers of classes and methods to be referenced by their simple name (e.g., the simple name `List` for the full class name `java.util.List`). This can result in accidental **name capture** in generated code: A generated name may refer to a different variable, field, or class than intended. In fact, as the Java syntax does not distinguish between *package* identifiers and *class* identifiers, even using a fully qualified identifier does not rule out name capture. For example, even the fully qualified identifier `java.util.List` is highly ambiguous: `java` could be a package, a class, or even a local variable; likewise `util` could be an inner class or a field, etc. Naming conventions may help the human reader (i.e., packages are normally in lowercase), and may prevent name capture in *most* cases, but cannot be depended upon in a solid compiler. This could lead to confusing error messages for the user of the generator, or even to a potential security issue.

Java compiles to **bytecode**, the instruction language for the Java Virtual Machine. It is more low-level and more explicit than the Java language. As such, it does not suffer from ambiguity between class and package names. It provides low-level primitives such as a *jump* or *jump subroutine* instruction, unbalanced synchronization, and stack manipulation; features that are not available from the Java language. While these constructs are generally not missed in ordinary application programming, for code generation they can be very useful. They may make a close match with features in a specific source language that needs to be compiled for the JVM, or can provide a performance gain over Java language alternatives. Additionally, bytecode can contain debugging information, which cannot be directly controlled from Java source code.

1.1 Contributions

We present **the Dryad Compiler**, a compiler for a language formed by **mixing Java and bytecode**. We investigated how such a language can be leveraged for various generative programming tasks in the field

of language extensions and separate compilation. The mixed language combines its two constituents to the point where not only methods can be used interchangeably, but also Java statements and expressions can be mixed with bytecode instructions. We maintain type-safety through direct type-checking of the combined language, extending an existing Java type-checker. By imposing additional restrictions on the effect on the stack that bytecode fragments may have, we allow natural composition of fragments of code, ensure locality of errors, and maintain a high degree of language "sanity."

We investigate the implications of the combined language on different applications of code generation, building on our experience with the limitations imposed by code generation to mere Java. For developing **embedded DSLs** or other extensions of the Java language, it is possible to *assimilate* the extension by replacing the every extended construct with regular Java code. It is often necessary to assimilate a DSL *expression* to an implementation using several Java *statements*. Java however does not allow statements in place of an expression. The mixed Java/bytecode language makes it possible directly compile such expressions in-place to an inline sequence of bytecode instructions. Additionally, as we can place Java statements and expressions in place of such instructions, the DSL expression can be assimilated directly in-place to Java statements (see Section 5.1). A common task in assimilating languages, and code generation in general, is the use of **intermediate local variables**. The mixed language can be used to implicitly type such variables, using bytecode to forgo explicit declaration of the variable on the Java level. It also allows the use of (anonymous) **stack allocation** to prevent accidental name capture, while providing a potential performance benefit (see Section 5.2).

We also evaluate how bytecode primitives such as a low-level *jump* instruction can be used in for example **parser generators**. With only minimal changes to an existing parser generator back-end, the introduction of the jump instruction improves the run-time performance of lexical analysis and simplifies the generation of finite automata (see Section 5.3). Such primitives can also be used in the implementation of language features that extend the regular Java language, such as **extended control flow constructs**. We applied this to yield continuations, a control flow construct for defining iterators (see Section 5.4). Without the combination with bytecode in the output language, such tasks are considerably more complicated. The ability to emit bytecode instructions within Java code can also be applied for rapid prototyping of specialized compilers for a customized virtual machine with a **custom instruction set**. This can be used to support recent proposals [21,13,14,63] for enhancements of the Java Virtual Machine architecture (see Section 5.5).

As our compiler takes separate compilation of Java to a new extreme, it is a very suitable foundation for the implementation of **software composition** language extensions and systems. The mixed language unifies source code (Java) and compiled code (bytecode), and can therefore be used to compose programs in either representation. This allows a *traits* compiler or aspect weaver to operate solely as a compositional tool, rather than as a full-fledged compiler (see Section 6). For traits this painlessly introduces the notion of **separate compilation**, which means we can type-check trait definitions before composition, and enable distribution of traits in binary form in libraries. For aspects, this approach significantly simplifies the implementation effort compared to that of existing aspect-weaving compilers that support separate compilation by extending an existing Java compiler.

An important facility in generative programming in general, and software composition specifically, is the ability to include **debugging information** in generated code. Such information can be used for debuggers or run-time error messages, and implies awareness of the origin of generated code at all times. We provide syntax and library mechanisms to maintain this information during a transformation. While not a novel contribution by itself [50,65], it is prerequisite to high-fidelity **separate compilation**. Together with the mixed target language it provides a complete, flexible platform for fine-grained

separate compilation, enabling its wide use in various applications.

The Dryad Compiler allows language extensions to be implemented in loosely coupled, light-weight fashion similar to common preprocessors. By outputting to the Java/bytecode language, applications can benefit from use of both the front-end and back-end of the compiler. By providing facilities to maintain source code information, this approach does not suffer from the reliability issues that are normally associated with preprocessors (e.g., errors referring to line numbers of generated code rather than the original source code).

1.2 Outline

The remainder of this thesis is organized as follows. In **Section 2** we provide essential background information on the Java Platform, the virtual machine, bytecode, and briefly discuss the notion of rewrite rules for program transformation. In **Section 3** we introduce the mixed Java/bytecode language, and discuss its design. **Section 4** discusses essential techniques required for compiling this language and enabling the proposed applications. In **Section 5** we demonstrate how the language can be applied in applications in the field of language extensions. In **Section 6** we describe applications in separate compilation and software composition, also applying some of the techniques of the preceding section. In **Section 7** we present the underlying compiler implementation, its design, and the tools used. **Section 8** discusses related work. Finally, **Section 9** presents our conclusion and points to directions for future work.

Chapter 2

Background and Preliminaries

2.1 The Java Platform

This section gives a basic introduction to the Java Virtual Machine architecture. For further information, readers are directed to the Java Virtual Machine specification [1], the official defining document of the Java Virtual Machine, as specified by Sun Microsystems.

At the heart of the Java Platform lies the **Java Virtual Machine, or JVM**. The virtual machine forms an abstraction over the actual hardware and operating system used by a system. Rather than executing natively compiled programs, it executes programs compiled for the virtual machine. Aimed at the virtual machine, such programs are platform-independent, and can run on different hardware and operating systems, as long as a virtual machine is available for it.

The virtual machine interacts with the specific operating system and provides an implementation of the Java **standard library**. This library provides essential functionality for writing programs, such as collections, I/O operations, networking, reflection, to name a few [33]. The JVM is essentially designed to be similar to a traditional, non-virtual computing machine. It has a fixed set of virtual machine **instructions**, and memory in the form of a **stack** and a **heap**. Notable differences to regular computing machines are that it has no notion of registers, and that memory on the heap applies automatic **garbage collection** to remove unused data. The basic components and behavior that every JVM implementation must offer are thoroughly specified, but not so restrictively that it prevents the implementors from making different choices for specific systems. This enables it to be implemented on a wide variety of platforms, ranging from mainframes to pocket devices. For example, floating-point calculations were loosened to allow slight differences in the results of calculations across different platforms (although regions of code can force strict results using the `strictfp` modifier, at the cost of performance).

In the past decade, the Java Platform has proven itself useful for deploying cross-platform applications, with support for thousands of high-quality libraries. At the end of 2006 there were over 21.000 open-source Java projects on the popular SourceForge collaboration project[1], making it the language most used on the site. Providing features such as automatic garbage collection, execution safety, and dynamic compilation, it makes it an excellent platform for developing new high-level languages [27,28]. In fact over 200 languages have been created targeting the JVM, including Jython (a Python implementation) [9], Kawa Scheme [10], Fortran [48,49], and Pizza [11].

[1]Sourceforge is a community of open-source projects and developers, that provides free web-based services at http://www.sourceforge.net/.

```
class Foo {
  String field;

  void method() {
    field = "hello world";
    System.out.println(field);
  }
}
```

Figure 2.1 A basic Java "hello world" class, with a single method and field definition.

2.1.1 Classes and interfaces

Java is an imperative, object-oriented language. The Java Virtual Machine is designed according to the same basic structure and type system, and as such is also fully object-oriented. Objects belong to a particular class. Classes are essentially object-oriented modules that may define methods and fields (see **Figure 2.1**). Java supports single inheritance for classes: Any class may inherit from (i.e., is a subtype of) a single other class. When a class inherits from another class, all its members (i.e., methods and fields) are also inherited: The subtype (or 'subclass') will have the same members as the inherited class, and may extend these with its own set of members.

The root of the inheritance tree is the `Object` type: All objects are instances of this type. Java provides native support for arrays, again instances of the `Object` type. Java requires explicit type specifications for class fields and local variables. Using the subtyping relation, it is also possible to use a **supertype** for such a reference:

```
String s = "Text";
Object o = s; // o now references the string (String is a subtype of Object)
```

In addition to class inheritance, Java also provides **interface inheritance**. Java interfaces are abstract types that do not define an implementation, but only define method signatures that a class must implement. Classes can be declared to implement one or more interfaces, if they provide an implementation for the interface methods. Like supertypes, interfaces can be used to reference an object of other types, as long as it implements the interface. This can be useful for accessing classes that are not in the same inheritance chain (and may only have the `Object` supertype in common, which does not define any type-specific methods). The Java standard library defines many such interfaces [33], to accommodate for the fact that classes may only inherit from a single superclass. An example is the `List` interface for lists of objects, which is implemented by different collection classes. It is also implemented by various classes that do not inherit from the collection classes, such as classes for graphical controls that inherit from other graphical controls.

2.1.2 Methods

In Java all methods are **virtual** by default. This means that when a class inherits from another class, it may **override** the methods defined by its superclass. For example, the `Object` class defines a `toString()` method that returns a string representation of an object. A list object inherits this method, and may override it to return a comma-separated string representing all the contained objects. Inheritance applied this way provides a way of code reuse and composition of class methods. In addition to virtual methods, methods may be sealed to prevent them to be overridden (using the `final` attribute).

Regular methods are bound to an instance of an object, but it is also possible to declare a method `static`

```
class Foo {
  int i, j;
  int k = 2;

  public Foo() { // called if a new Foo object is created (using 'new Foo()')
    i = 1;
  }
}
```

Figure 2.2 A simple constructor method. Constructors use the same name as their defining class, and do not have a return type. Creating a new instance of this class will initialize field `i`. Field `k` is also initialized, by means of a field initializer.

```
int i = 5;
Object o = i;            // auto-boxing to an object type
Integer wrapped = i + 2; // auto-boxing to an Integer
int j = wrapped;         // auto-unboxing of the primitive wrapper type
```

Figure 2.3 Using autoboxing and unboxing to implicitly convert between primitive and object types.

to bind them to a class. Static methods cannot be overridden, and cannot refer to a specific class instance (`this` in Java). Another special form of method is the **constructor** method, which initializes an instance of a class. Every time a new object of a class is created, a constructor (if defined) is called, along with any **field initializers** to initialize the class and its fields (see **Figure 2.2**).

Java uses **access modifiers** to protect class members from access by unauthorized, external classes. The most common forms of these are **public members** that are publicly accessible to any outside class, and **private members** that can only be accessed by the class that defines them. Unlike other types of methods, private methods cannot be accessed by subclasses.

2.1.3 Primitive types

In addition to objects, Java also supports primitive types. These can be used to store a single integral or floating-point value. Java supports a total of eight different primitive types: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and `char`. These types have a preset range (e.g., a byte ranges from -128 to 127) and space usage characteristics (e.g., an `int` occupies 4 bytes). Unlike objects, they are not referenced to using an object reference (pointer), but are stored in place in an object or on the stack (see Section 2.1.8). This means that primitive types have a smaller memory footprint than objects, and do not require garbage collection.

Since primitive values are not objects, they do not inherit from the `Object` type. Because they lack this subtyping relation, they cannot be assigned to a variable or field of the `Object` type, nor can they be stored in a collection of `Objects`. To overcome this limitation, Java introduced the notion of **autoboxing** and -unboxing. Using auto-boxing, any primitive type can be implicitly converted to a corresponding **primitive wrapper type** (or 'boxed type'). A primitive wrapper type is an object with a single, primitive field, and effectively encapsulates a primitive type in an object type. Examples of primitive wrapper types include the `Integer` class to store an `int` and the `Boolean` class to store a `boolean` value (see **Figure 2.3**).

2.1.4 Multi-threading

The Java Virtual Machine is designed specifically to support multi-threaded applications. The Java memory model is strictly defined to properly allow memory access in multi-threaded applications,

without leading to undefined situations. To ensure safe access of shared memory, the Java language provides **synchronization** constructs. Methods or blocks of code can be marked as `synchronized` to indicate they should be treated as a critical section of code. Threads that execute this code will implicitly acquire a monitor lock, ensuring only one thread can enter that section of code (for the given object instance) at a time. After the section is exited, the virtual machine ensures that monitor lock is always released, no matter the state of the execution. At that point another thread may enter the critical section.

Other than synchronization, the Java language offers little in the way of language support for handling multi-threading. However, the Java standard library [33] includes support for creating new threads, aborting threads, suspending threads, etc. While the virtual machine must support these library methods, this means there are no special virtual machine instructions for dealing with threads other than instructions for synchronization.

2.1.5 Basic control flow

As an imperative language, Java supports several control flow constructs, such as the `if` statement, the `while` loop, and a C-style `for`-loop and `switch` statement. Java does not provide support a `goto` statement, but does provide clauses for terminating a loop early (the `break` statement) or skipping the rest of the loop body and forcing the next loop iteration (the `continue` statement):

```
while (condition) {
  ...
  if (finished) break;        // exit the loop
  else if (restart) continue; // go to the beginning of the loop
  ...
}
```

2.1.6 Exception handling

The Java language and the JVM provide direct support for exception handling. This mechanism handles the occurrence of some condition that requires a change in the normal flow of execution. Such a condition is called an exception, and is used for signaling errors or other exceptional circumstances. For example, a method that reads a file may result in a `FileNotFoundException`, if the inputted file is not found. Java supports `throw` statement to raise an exception. Exceptions can be dealt with using the `try/catch` construct. If an exception occurs within the `try` clause, the `catch` clause will be executed to handle it (see **Figure 2.4**). If there is no `catch` clause that can be activated to handle the exception, the entire application will abort.

The `try/finally` construct is another way of managing exceptions. Regardless of whether an exception occurs in the `try` clause, the `finally` clause is always executed when the block is exited. It is commonly used to clean up resources, for example closing a file handle. After the `finally` block is executed, the control flow continues at the point where it normally would; in case of an exception it continues at the exception handler, if any is defined. **Figure 2.5** demonstrates this pattern.

Because of the multi-threaded nature of Java, it also allows asynchronous exceptions. These exceptions can be caused by another thread (using the deprecated, strongly recommended against `Thread.stop()` method). Alternatively they can also be thrown by an internal error in the JVM. Regular, synchronous exceptions occur at predictable places (i.e., when performing some operation that might result in an exception). Asynchronous exceptions may occur at any point during the execution. Fortunately,

```
class ExceptionDemo {
  void main() {
    try {
      String file = readfile("does not exist.txt");    // try to read a file
      ...
    } catch (FileNotFoundException exc) {   // catch a FileNotFoundException
      exc.printStackTrace();                // print details if it doesn't exist
    }
  }

  // readFile normally returns a String, but might also throw an exception
  String readFile(String filename) throws FileNotFoundException {
    File file = new File(filename); // create a new file object for the filename

    if (!file.exists())
      throw new FileNotFoundException();
    ...
  }
}
```

Figure 2.4 Throwing and catching exceptions in Java.

```
FileReader reader;
try {
   reader = new FileReader("file.txt"); // open (abort if an exception occurs)
   ...
} finally {
   if (reader != null) reader.close(); // close (unless it was never assigned)
}
```

Figure 2.5 Using the `try/finally` construct to ensure a file is always closed, even if an exception occurs.

asynchronous exceptions are very rare to nonexistent in most applications. However, they should never be completely ignored when analyzing or writing Java code.

2.1.7 The Java class file format

Though the JVM is designed for use with the Java programming language, it does not assume that the programs it executes were compiled from this language. As such, it can host other languages and extensions of the Java language. The JVM processes classes in a particular binary format, the **Java class file format**. Such a class file has the following components [1]:

- File **format version**. Version 50.0 for example means that Java 6 is required.
- **The constant pool**, a table contains the values for all constants. The constants can be used (and shared) throughout the class by referencing their index in constant pool. It includes numeric and string literal constants, as well as symbolic references (i.e., names of types or members).
- Type information and metadata for **the class**. This includes, among other things, the name and package name uniquely identifying the class, the superclass, and all implemented interfaces.
- Type information and metadata for **each class member** (i.e., method or field). For methods this includes a list of **bytecode instructions**; instructions for the virtual machine.

2.1.8 Bytecode instructions and the operand stack

Bytecode instructions are simple opcodes that are each represented as a single byte in a class file,

Instruction	Stack after execution
`iconst_1` (load each constant on the stack)	`[ 1 ]`
`iconst_2`	`[ 2, 1 ]`
`iconst_3`	`[ 3, 2, 1 ]`
`imul` (multiply stack values `3` and `2`)	`[ 6, 1 ]`
`iadd` (add stack values `6` and `1`)	`[ 7 ]`

Figure 2.6 Evaluation of the mathematic expression 1 + (2 * 3), using the stack and bytecode instructions for loading constants, multiplication, and addition.

followed by zero or more static arguments. All instructions take a fixed number of arguments (for example, `ldc`, an instruction to load a constant, requires the index of a constant pool entry). In addition to these arguments, the instructions can also process **operands** from the **stack**. The stack is essentially a simple list of values: values that result from operations (e.g., loading a constant), and values used as the input to other operations (e.g., adding two numbers). The instruction set includes support for arithmetic operations, such as additions, subtractions, divisions, etc. More advanced mathematic operations are supported by the library, through method calls. The operator instructions make intensive use of the stack: Given one or two values, they will return a single resulting value on the stack. For long expressions this can mean that a long stack of intermediate values is built up (see **Figure 2.6**).

Bytecode instructions are strongly typed. Many instructions encode the type they operate on in the instruction mnemonic (name): For example, the instructions in **Figure 2.6** operate on values of the `int` type. As such, their instruction mnemonic (name) is prefixed by an 'i'. Equivalent instructions for other types, such as `float` or `long` start with an 'f' or 'l', respectively. Other instructions, such as invocation instructions refer to the constant pool to specify their type information. Values on the stack are of course determined at run-time, but their type must also be statically determined at compile-time, to ensure type safety. Additionally, the stack for a given method has a predefined size; the stack cannot grow beyond this point.

Stack operations Because of the important role of the stack for bytecode, there are several instructions to directly manipulate it. Various instructions exist for pushing a constant on the stack, for different sizes and types of constants. For example, for pushing an integer value between 0 and 255, the `bipush` instruction can be used, as it can be represented by a single byte. For a larger integer, the `sipush` instruction may be used, and for a word-sized integer, the `ldc` instruction must be used. In the runtime these values have the same representation on the stack, regardless of their size. In this representation normal stack values require four bytes (one word), with the exception of `double` and `long` values that require two words.[2] The reason that the JVM uses these specialized constant loading instructions is that binary class files can be slightly smaller with this representation: A small integer does not require four bytes for its representation. Based on this principle, the JVM supports additional single-byte instructions for certain special cases: Instructions `iconst_1` up to `iconst_5` for example push an integer from 1 up to 5 on the stack. Since a bytecode instruction only occupies a single byte, there is only a limited number of these instructions, but they do need to be considered for code generation or analysis.

For regular bytecode instructions the stack can be viewed simply as a series of values of given types. A small set of special stack manipulation instructions exists that operate on the stack as raw values of a given width, without regard to type. For example, the `dup` instruction duplicates a single word on the

[2]JVM implementations for 64-bit machines may use a different internal representation, but are still required to interpret instructions as if they were using a word-based representation.

Instruction		Stack after execution
`iconst_1`		`[ 1 ]`
`dup`	(duplicate value)	`[ 1, 1 ]`
`istore 1`	(store as local variable #1)	`[ 1 ]`
`iload 1`	(retrieve local variable #1)	`[ 1, 1 ]`
`iadd`		`[ 2 ]`

Figure 2.7 Using the `dup`, `iload`, and `istore` instructions.

stack, and the `swap` instruction swaps two words on the top of the stack. These operations may not be used to break up two-word value types (which would not be a type-preserving operation). Attempting to do so results in an error. Instead, there are variants of the instructions to deal with such types, such as `dup2` which duplicates a two-word value (or two single-word values).

Local variables In addition to stack operands, Java also provides local variables to store runtime data. Local variables in bytecode are the direct equivalent of local variables in Java source code. Local variables are addressed by an offset, and exist separately from the operand stack. While the stack only allows limited access to its top elements and through stack manipulation operations, local variables can be directly accessed by their offset at all times. However, since nearly all instructions operate on the stack rather than on local variables, they must be placed on the stack to be manipulated or accessed. Using *load* and *store* instructions, they can be retrieved and pushed on top of the stack, or stored from the stack into a variable (see **Figure 2.7**).

Fields Similar to local variables, data in fields can be accessed using separate instructions: the `getfield` and `putfield` instructions. These are explicitly typed and require the complete package and class name of the retrieved field (this information is stored in the constant pool). Also unlike in the Java language, bytecode distinguishes between static and instance fields, and provides separate `getstatic` and `putstatic` instructions for static fields.

Method invocations While the Java language provides only a single syntax for invoking a method, in bytecode there are several instructions, depending on the sort of method: `invokestatic` for static methods, `invokeinterface` for interface methods, `invokespecial` for constructors, and `invokevirtual` for regular (virtual) methods. In Java source code it is optional to specify the complete package and class name for a class if this can be inferred from the context (such as the imported classes for the current source file). Bytecode, being more explicit, requires this to always be specified. A complete method invocation requires the package, class, and method name, as well as the parameter types and return type of a method.

Control flow Bytecode uses four basic operations to handle control flow: The jump instruction (`goto`), conditional jumps (similar to an `if` in Java), returning from the method, and exception handling. All the various Java control flow constructs can be rewritten to use these simple primitives.

Synchronization To provide for synchronization (see Section 2.1.4), the bytecode instruction set includes the `monitorenter` and `monitorexit` instructions. The `monitorenter` instruction is used to acquire a monitor lock for a critical section of code, while the `monitorexit` instruction is used to release the lock. A lock is always established on a given object, such as the current object instance. As such, both instructions require an object instance on the stack on which the lock is acquired.

```
int fac (int i) {
   if (i == 0) {
      return 1;
   } else {
      return i * fac(i-1);
   }
}
```

```
int fac(int)
  Code:
   Stack=4  // maximum stack depth
   Locals=2 // number of local variables ('this' and i)
   0:   iload_1          // load i on stack
   1:   ifne 6           // if (i == 0)
   4:   ldc #1           // load constant #1
   5:   ireturn          // return stack value
                         // else
   6:   iload_1
   7:   aload_0          // load 'this'
   8:   iload_1
   9:   ldc #1
   10:  isub             // substract stack values
   11:  invokevirtual #2 // call Method fac:(I)I
   14:  imul             // multiply stack values
   15:  ireturn
  LineNumberTable:
   line 1: 0
   line 2: 4
   line 4: 6
```

Figure 2.8 A factorial function, represented as Java source code and as bytecode. The bytecode method specifies the type signature, the maximum stack depth and the number of local variables. It also includes a table with the bytecode instructions, and an (optional) "line number table" mapping lines from the source code to the bytecode positions. The `invokevirtual` instruction makes a recursive method call, and is parameterized with constant pool entry #2.

```
Exception in thread "main"
  java.io.FileNotFoundException: bar.txt
    at Foo.myMethod(Foo.mylang:10)
    at Foo.main(Foo.mylang:5)
```

Figure 2.9 An exception trace generated by a program compiled from a source file named "Foo.mylang".

Figure 2.8 illustrates a complete bytecode method with some of the most commonly used instructions. In section 3.3 we give a further overview of the instruction set, and how they are used in our combined language. A complete overview of all instructions can be found in the Java Virtual Machine specification [1].

2.1.9 Debugging information

Class files as generated by the Java compiler can contain symbolic debug information. This information contains the original source line numbers for each instruction and name of the source file. It can also include the names of all local variables. This information is stored in the form of tables that reference instruction locations or local variable indices, and accompany each method (see **Figure 2.8**). These tables can be used by external debuggers to allow the programmer to step through the code and inspect local variables. The information is also used in exception traces: For every exception thrown, a stack trace is generated complete with the original source filenames and line numbers (see **Figure 2.9**). It is displayed on screen when the exception is not caught, or can be used by the application.

2.1.10 Execution safety

Java's security model is focused on protecting end users from possibly malicious or erroneous programs that could be downloaded from across a network from untrusted sources. This allows Java program to be hosted on a website and run inside a web browser. Java provides fine-grained security policies and authentication of digitally signed code.

The Java security model is integrated into the lowest levels of the JVM. To enforce it, the JVM does not

allow any kind of direct access to memory of the underlying system. This prevents Java programs from interfering with the native hardware and operating system. Pointer arithmetic is simply not supported; all memory access is through type-safe references. Garbage collection is used to prevent memory leaks and ensures that unused (or previously unused) memory is never referenced.

The JVM must ensure that all executed code is **type-safe**, that referenced members and variables are valid and accessible, that objects are not accessed before initialization, and that all jumps are to valid locations. If these conditions are not met, execution would leave the runtime in an undefined state. The JVM includes a **verifier** that analyzes all code before execution to ensure the conditions are met, and that security policy is adhered to [8,15]. It will use a **static analysis** to confirm that all instructions (and their operands) are valid.

Java performs most safety checks statically at load-time, allowing it to safely omit most runtime checks. Other checks, such as type casting checks and array bounds checks, must still be performed at runtime.

2.1.11 Dynamic class loading

The Java Virtual Machine is responsible for loading all required classes and interfaces from the class files. This loading is deferred until they are required for execution or verification of classes. Delaying this as long as possible allows reduced memory usage and improved system response time.

All references to other classes, methods or fields are located in the constant pool, and are represented by an identifier in the form of a string. The JVM uses it to build an implementation-specific runtime constant pool. When a method is first executed, any symbolic references used from instructions are then replaced by direct references (such as a pointer or offset to the class, field, or method). Classes will be loaded if they were not already at this point. If the referenced class is not valid or does not exist, an error will be thrown, allowing the application to recover gracefully.

Since classes are loaded as separate modules after compile-time, they are effectively **dynamically linked**. Dynamic linking means that classes may be updated or modified after compilation, perhaps because they are distributed to an installation that has a different library version. Java uses load-time verification to verify that any updated classes comply to the interface requirements established at compile-time, and will throw an error if this is not the case. Other compilers often do linking compile-time, where all modules are combined into a single file. This simpler model allows the compiler to make more assumptions about the included modules at runtime, for example allowing optimizations such as inlining. (Java does such optimizations at load-time; see Section 2.1.12.)

2.1.12 JIT verification and compilation

Just like loading classes, verification and compilation can also be done “just-in-time” (or simply “JIT”). Modern JVM implementations include a **JIT compiler** that can compile methods into native machine code. The JIT compiler can apply several optimizations, such as using static analysis to remove unneeded array bounds checks, or inlining of invoked methods. This can significantly increase the performance compared with plain interpretation of bytecode.

JIT compilation can also slightly increase the time required to load each method. To prevent this, advanced JVM implementations use runtime profiling to determine what methods are used most often. It can use this information to selectively compile methods that run many times during the lifetime of a program, and for inlining frequently called methods.

2.2 Program transformation with rewrite rules

For automatically rewriting or compiling a program, we make use of rewrite rules. Rewrite rules provide a simple, almost mechanical formalism for describing such transformations. A basic rewrite rule consists of a left-hand side (the source) and a right-hand side (the target):

```
desugar-while: ⟦ if (e) stm ⟧ → ⟦ if (e) stm else ; ⟧
```

In this rule we rewrite the one-armed `if` statement to a regular `if` with two arms. Essential in this rule, and rewrite rules in general, is the notion of **variables**. The rewrite rule above applies to *any* `if` statement for a given condition *e* and a statement *stm*. To denote that these are variables rather than static clauses, variables will be printed using *italics* throughout this thesis.

The rewrite rule above is an example of **desugaring**: The one-armed `if` construct, which is essentially syntactic sugar for the regular `if`, is hereby eliminated. Other examples of syntactic sugar are the `while` construct that may be rewritten to a `do-while` construct, and operators such as `+=`, which adds and assigns a value to a variable. By desugaring, other rewrite rules may focus on the resulting, simplified language, and proceed to rewrite the different language constructs to bytecode instructions. This practice is commonly applied in complex rewriting systems, to reduce the number of rewrite rules required for the base language.

2.2.1 Rewriting using rules in Stratego

For our implementation we use the Stratego transformation language (see Section 7.1), and as such we lend some of the basic syntax of rewrite rules from this language to illustrate selected concepts throughout this thesis. We should note that while we use it in examples, our system does not actually depend on the use of Stratego for the implement of the applications. They may be implemented separately from the Java/bytecode compiler, using any language or system that can produce an input for it.

Rewrite rules in Stratego may be written using **concrete syntax**, i.e., using the syntax of the language that is being rewritten [4,55]. For this, quotation delimiters (⟦, ⟧) are used to distinguish concrete syntax fragments. Alternatively, rewrite rules could rewrite the underlying data structures that represents the parsed form of the language constructs.

An additional construct that Stratego adds to the basic rewrite rule is the `where` clause. It can be used to specify any information needed in the right-hand side of the rule:

```
evaluate-plus:
  ⟦ e1 + e2 ⟧ → ⟦ e3 ⟧
  where e3 := <add-ints> (e1, e2)
```

This rule invokes the `<add-ints>` function which adds two integers from the left-hand side, and assigns the resulting value to *e3*, which is used as the right-hand side. (Note that this is not how the operator is implemented in Java; the rules in this section serve merely as an example.) The `where` clause can also be used to specify a condition for a rule to succeed: If it fails, the rule will also fail and will not be applied. This means that if the input to the rule above is a concatenation of two strings (also denoted by a + in Java), the `<add-ints>` invocation will fail, and so therefore the rule will not apply.

The concept of failure has a central place in Stratego, much like exceptions or return values in other languages. As such it has different operators to deal with this condition. One such operator we use here in concrete code examples is the `not` operator, which succeeds if a given function call fails. This allows us to write a rule that depends on something to fail:

```
desugar-plus:
  ⟦ e1 + e2 ⟧ → ⟦ new StringBuilder().append(e1).append(e2) ⟧
  where  not(<is-primitive> e1);
         not(<is-primitive> e2)
```

In this example, the `where` clause will only succeed if the two invocations of `<is-primitive>` fail. If that is the case, the input will be rewritten to a string concatenation using the Java `StringBuilder` class. In Java, the concatenation operator (+) can be applied to objects of any type. The `append()` method accepts any object as its input, and uses the standard `toString()` method to get its string representation. The Java concatenation operator may also be applied to an object and a primitive type:

```
"String" + 5
```

This expression results in `"String5"`. To support the case of a single primitive type and an object type we need to adapt our rewrite rule, using the choice operator (<+):

```
desugar-plus:
  ⟦ e1 + e2 ⟧ → ⟦ new StringBuilder().append(e1).append(e2) ⟧
  where  not(<is-primitive> e1) <+ not(<is-primitive> e2)
```

The <+ operator works much like a standard 'or' operator: It first tries to evaluate its left-hand side, and in case of failure it will evaluate the right-hand side. By applying this operator, our rule now succeeds if *either* of the two conditions succeeds; the expression will be rewritten in case it has one or two non-primitive type inputs.

This concludes our brief introduction to rewrite rules. In the remainder of this thesis we make use of such rules to illustrate selected concepts, and use pseudocode in lieu of more advanced Stratego constructs (or will explain the constructs as they are applied). For a complete reference of the Stratego language, in particular about the topic of strategies (i.e., functions in Stratego) we refer the reader to the Stratego documentation [55].

Chapter 3

Language Design

"Anything can be made to work if you fiddle with it.
If you fiddle with something long enough, you'll break it."

– Murphy's technology laws

The mixed Java/bytecode language lies at the heart of our compiler design. In this chapter we discuss the language design and the requirements that defined it. The language essentially consists of three components: The regular Java language, a textual representation of the binary bytecode language, and a small number of quotation constructs that aid in the integration of the two base languages. Given that the standard Java language is well defined [2], we limit our discussion here to the bytecode representation (Section 3.2) and the integration of the two languages (Section 3.5).

3.1 Safety first

Java's strongly typed language design allows for a great deal of compile-time checking, which makes it harder for developers to shoot themselves in the foot. When extending Java, care must be taken to preserve this property. When Java is combined with a low-level language such as bytecode, this does not come naturally. Direct access to bytecode potentially opens the doors for stack overflows and underflows, type errors, and other threats. Fortunately, the built-in verifier in the Java Virtual Machine prevents such errors from resulting in system crashes and security problems, but they still lead to a program that does not work. In addition to type-checking the mixed language to prevent such problems, we impose certain restrictions on the effect on the stack that embedded bytecode may have, to avoid stack mismatches, underflows, or overflows. This way errors can be intuitively prevented by programmers, or are picked up at compile-time by the compiler at the fragment of code where these constraints are violated. (The alternative, reporting the error at the location where the program actually fails to type-check, makes it significantly harder to find the cause of such errors.) In other words, in our design the language we try to provide the same safety (or "sanity") as provided by the regular Java language.

3.2 Bytecode representation

The Java class file format was originally designed to be compact enough to allow it to be transferred over the Internet in the mid-1990s. As such, it is a compact binary format that is not easily edited by human beings. Many textual representations of bytecode exist to aid in this. For example, Sun provides the `javap` disassembler tool, which uses a language that is very close to the binary representation (see

```
void say(java.lang.String text) [
  catch (handler: Exception) [
    getstatic java.lang.System.out : java.io.PrintStream;
    load text;
    invokevirtual java.io.PrintStream.println(java.lang.String : void);
    return;
  ]

  handler: // (handle exceptions at this label)
]
```

Figure 3.1 Example of a bytecode method that calls `System.out.println()` and handles any exception it may throw.

Figure 2.8 on page 21). Other representations, most notably **Jasmin**[3] and its derivatives, provide a somewhat more abstract representation. Jasmin provides assembler-like syntax that uses the canonical instruction names from the Java Virtual Machine specification. Unlike the `javap` representation, it provides symbolic labels to serve as targets for branch instructions and exception clauses. The underlying binary format uses relative and absolute *offsets* instead, which must be updated with any changes in a program. Jasmin also abstracts from the constant pool (see Section 2.1.7), allowing every constant to be placed inline instead of as a constant pool reference. With this abstraction, constants can be easily added and can be directly modified in-place, without corrupting other references to the same constant.

Jasmin provides a solid basis, but in this project we have additional requirements for the bytecode representation. We need it to allow tight integration of bytecode with Java source code, and it must allow painless composition of fragments of code. We use the fundamental principles from Jasmin together with some adjustments (see **Figure 3.1**):

- Similar to labels for jump locations, we use names for local variables. Normally, these are represented by an offset. These can be automatically generated instead, and the compiler can reserve a proper 1 or 2-byte slot for each variable. Using names allows variables to be shared between Java and bytecode. As long as variable names are unique or scoped (see Section 36), this can also help ensure that variables do not clash when two methods or fragments of code are combined. The number of used local variables and maximum height of the operand stack can be inferred automatically, therefore we eliminate the need to specify them manually. (Specifying this would be harder for combined Java and bytecode, but is rather straightforward for the compiler once the method is compiled to pure bytecode.)
- Exception handler locations (`try/catch` blocks in Java) are specified in the method header in Jasmin. Inlining these can help code generators since they can directly emit exception handlers *inline* rather than generating a separate table. (Note that this representation means that exception blocks are always properly nested, which is also demanded by the JVM specification.) This also helps with composing or manipulating bytecode fragments, as it eliminates the need to update the exception handler tables.
- Instead of specifying debug symbols in a separate table, we include references to the original code *inline*. Again, this helps method composition and manipulation. We also apply this information to generate compile-time error messages that trace back to the original source code (see Section 4.2).

[3] http://jasmin.sourceforge.net/.

3.3 Bytecode instructions and pseudo-instructions

The Java Virtual Machine knows over two hundred instructions. The instructions range from very low-level operations, such as manipulation of the stack, to instructions with direct ties to the Java language, such as loading a field or local variable, to high-level operations that offer support for synchronization. Many of these instructions are highly specialized for specific uses (see Section 2.1.8). For example, there are eight different instructions for loading a value from an array, depending on its type. For code generation this means that a slightly different sequence of instruction may need to be generated for the same source code fragment, depending on the context. We introduce a set of generalized **pseudo-instructions** as alternatives to some of the more specialized instructions.

The use of pseudo-instructions is common practice in compilers to delay the instruction selection process to a later phase of the compilation. Sometimes they are used as an intermediate representation to support emitting to multiple instruction sets for different processors in the back-end. Other times they are applied purely to encapsulate the instruction selection process. For Java compilers, it is not a very common practice to do this, but one existing system for bytecode abstraction is **Soot** [35]. Soot defines multiple levels of abstractions over bytecode, for the purpose of analysis and optimization. The abstractions introduced in Soot are at a relatively high level, and are designed to avoid having to deal with bytecode as stack machine code[4]. In our approach we stay very close to the original set: Every regular bytecode instruction can be mapped directly to an alternative in the reduced instruction set. Moreover, we maintain compatibility with the standard instruction set[5], although in this thesis and the provided syntax definition we focus on the reduced set. In the remainder of this subsection we will discuss the various categories of bytecode instructions (see **Figure 3.3**), their uses, and abstractions we introduced.

3.3.1 Local variables

Local variables are loaded and stored using a two-byte operation: one byte for the instruction (e.g., `iload` to load an integer), and one byte for the index of the local variable. To safe space, the JVM also provides some instructions for specific indices, only requiring a single byte of space, such as `iload_1`. In our approach we introduce two overloaded pseudo-instructions: `load` and `store` that take a *named* local variable as their argument. The instruction specialization, implemented in the compiler, will then make sure that the more space-efficient variants are used when possible. (See also the implementation discussion about overloaded instructions in Section 7.5.)

3.3.2 Arrays

While arrays are regular objects in Java, several bytecode instructions are provided to access and create arrays, rather than using method calls. Like for local variables, there are specialized *load* and *store* instructions available for arrays. We introduced overloaded `aload` and an `astore` instructions as alternatives to the regular `<t>aload` and `<t>astore` instructions.

---

[4] The *baf* representation level is an exception to this, and introduces very little abstractions other than names for local variables [35].

[5] Actually, we omitted the `jsr/ret` ("jump-subroutine") instructions, that were used for exception handling blocks in older versions of Java, but are currently deprecated. Equivalent programs without these instructions can be produced by inlining code.

General	p	::= $\overline{cd}$	
	T	::= C | void | int | long | double | float | boolean | char | short | byte	
Java	cd	::= class C extends C { $\overline{fd}$ $\overline{md}$ $\overline{cd}$ }	Class declaration
	md	::= C ($\overline{T\ x}$) { super ($\overline{e}$); $\overline{s}$ } | $T\ m$ ($\overline{T\ x}$) { $\overline{s}$ }	Method/constructor declaration
	fd	::= $T\ f$;	Field declaration
	s	::= { $\overline{s}$ }	Statement block
		| e;	Expression statement (eliminates result)
		| $C\ x = e$;	Local declaration
		| if (e) s else s | while (e) s | for (e; e; e) s	Control flow
		| throw e;	Throw exception
		| return e;	Return
		| try { $\overline{s}$ } catch ($C\ x$) { $\overline{s}$ }	Exception catch block
		| l:	Label
	e	::= $x = e$ | x	Local variable access
		| $e.f = e$ | $e.f$	Field access
		| (T) e	Cast
		| $e.m$ ($\overline{e}$)	Method invocation
		| new C ($\overline{e}$)	Object creation
Bytecode	cd	::= classfile C extends C fields $\overline{f}$ methods $\overline{md}$	Class declaration
	md	::= C ($\overline{T\ x}$: T) [$\overline{I}$] | <init> ($\overline{T\ x}$: C) [$\overline{I}$]	Method/constructor declaration
	fd	::= f : T	Field declaration
	I	::= catch ($\overline{l : C}$) [$\overline{I}$]	Exception catch block
		| *(see Figure 3.3)*	Bytecode instruction
Mixing	s	::= [$\overline{I}$]	Bytecode statement
	e	::= [$\overline{I}$]	Bytecode expression
	I	::= stm: s | expr: e	Java as instruction
Tracing	s	::= new code (o) [$\overline{s}$]	Trace *[generated/new code]* back to *(old code)*
	e	::= new code (o) [e]	(see Section 4.2)
	I	::= new code (o) [$\overline{I}$]	
	o	::= $\overline{s}$ | e | $\overline{I}$	Source code; may contain position annotations

Figure 3.2 Syntax for mixed Java/bytecode programs (some advanced Java constructs are not included here).

Math operators
```
add
div
mul
neg
rem
sub
shl
shr
ushr
xor
and
or
inc
dec
```

Comparison operators
```
lt
gt
eq
le
ge
ne
```

Stack operations
```
ldc     c
ldc2_w  c
new     C
pop
dup
dup_x1
pop2
dup_x2
dup2
dup2_x1
dup2_x2
swap
```

Primitive conversions/ trunctations
```
x2i
x2l
x2d
x2f
i2b
i2s
i2c
```

Arrays
```
aload
astore
arraylength
newarray        T
multianewarray  T n
```

Local variables
```
load  x
store x
```

Fields
```
getstatic C.f : T
putstatic C.f : T
getfield  C.f : T
putfield  C.f : T
```

Method invocation
```
invokespecial   C.m(T̄ : T)
invokevirtual   C.m(T̄ : T)
invokestatic    C.m(T̄ : T)
invokeinterface C.m(T : T)
```

Control flow
```
ifeq l
ifne l
goto l
l:
athrow
return
xreturn
tableswitch
  n to n
  l̄
  default: l
lookupswitch
  n : l
  default: l
```

Miscellaneous
```
checkcast  C
instanceof C
monitorenter
monitorexit
nop
breakpoint
```

Figure 3.3 The reduced bytecode instruction set, with a total of 67 instructions. (Note that terminating semicolons are *optional* in this syntax definition, unlike for Java statements.)

3.3.3 Accessing fields

Instructions for accessing fields are `getfield`, `putfield`, `getstatic`, and `putstatic`. These respectively push a field value on the stack, assign the value on the stack to a field, or perform these operations for static fields. While it is possible to infer the signature of a field, and determine whether or not it is static, we feel that specifying this adds to the expressivity and explicitness of bytecode. The field accessing instructions take the complete declaring class name and package of a field as their argument. In case of instance fields they also require an object instance on the stack:

```
load this
getfield mypackage.MyClass.fieldName : String
```

This operation loads the current class instance (local variable `this`) and retrieves the value of the `fieldName` field.

3.3.4 Method invocation

As with fields, bytecode method invocations require a complete method signature, class name, and package specification (see Section 2.1.8). In our syntax definition we do not diverge from this. For example, the `println()` method of the `PrintStream` class in package `java.io` can be invoked using:

```
invokevirtual java.io.PrintStream.println(java.lang.String : void)
```

Bytecode allows to distinguish between package names, names inner classes, and field names. To instead invoke the `println()` method as defined by a third-party `PrintStream` class, defined as an inner class of class `io`, which in turn is defined as an inner class of class `java`, the invocation would be:

```
invokevirtual java$io$PrintStream.println(java.lang.String : void)
```

In this invocation the `$` signs are part of the name of the actual class file (which does not belong to a package). In regular Java it is not possible to distinguish between names of such inner classes or classes in packages (or even fields), and this information must be inferred from the context. In bytecode this ambiguity is eliminated through the use of a different syntax for inner classes, and a separate instruction for handling fields. This can be applied to advantage in code generators to eliminate any risk of name capture.

3.3.5 Stack operations

Stack operations include instructions for loading constants, such as `ldc` for loading a one-word constant, or `bipush` for a single-byte constant (see also Section 2.1.8). In total there are 20 different instructions for loading constants that can be used depending on the value and type of the constant. In our approach we defer this specialization to the assembler, and only require the use of the `ldc` and `ldc2_w` instructions. These are used respectively for one-word and two-word types, allowing us to distinguish between single-word `int` and `float` values and double-word `long` and `double` values (which share the same syntax). In regular bytecode, the two instructions require the use of the constant pool, which may be more costly in terms of space usage (and execution time, in case of a bytecode interpreter). However, since we can specialize the instructions in a later compilation stage, we need not be concerned with this overhead when these instructions are emitted.

Instruction	Stack after execution

`ldc "value"`
`dup`
`putstatic MyClass.staticField : java.lang.String`
	`[ "value" ]`
	`[ "value", "value" ]`
	`[ "value" ]`

Figure 3.4 Assignment of a field, while maintaining the assigned value on the stack. The `dup` instruction duplicates the topmost stack value.

Stack operations distinguish between one-word and two-word types, as they operate on the raw words rather than on values of a given type (see Section 2.1.8). This means that a bytecode generator needs to be aware not only of the current values on the stack, but also of their width on the stack. To fully appreciate the importance of stack instructions and the fact that they operate on words rather than values, let us look at an example of static field assignment (see **Figure 3.4**). A Java field assignment is compiled to a sequence of bytecode instructions. This sequence must maintain the value of the assignment on the stack, as Java allows nested assignments (e.g., `variable = field = "value"`). Since the assignment of the field takes the value off the stack, the value must be duplicated using the `dup` instruction. However, in case the field is of a two-word type, the `dup` instruction – operating on words – is not allowed and will result in an error. In our approach we *do* allow the `dup` instruction to be used in such cases, and rewrite the it to a two-word `dup2` instruction during the specialization process. Similarly, we allow the use of the `pop` and `dup_x1` instructions on two-word types, where they would fail before. For the other (more rarely used) stack operations we do not introduce such semantics as they operate on two-word types, and thereby also work on single-word types (albeit with different semantics).

3.3.6 Arithmetic operators

The arithmetic operators in regular bytecode are specialized to operate on a specific type (see Section 2.1.8). We introduce overloaded instructions as a more convenient alternative (see **Figure 3.3**). To support the common operation of incrementing and decrementing values, we also introduce the `inc` and `dec` pseudo-instructions.

3.3.7 Primitive conversions and truncations

Since Java supports various primitive types, it also supports operations to convert between these types: `l2i` converts a `long` to an `int`, `i2l` converts an `int` to a `long`, etc. We introduce overloaded conversion instructions that convert any inputted primitive value to a target type (e.g., `x2i` converts any primitive to an integer). In addition to conversions, Java supports truncation of an `int` to a `byte`, `short`, or `char`. These instructions – respectively `i2b`, `i2s`, and `i2c` – truncate an integer to the range of the specified type.

3.3.8 Control flow

Java control flow builds on very basic primitives such as **jumps** and **conditional jumps** (see also Section 2.1.8). Conditional jumps are essentially the bytecode equivalent of the Java `if` statement. Java supports a plethora of conditional jump instructions, specialized for various comparison operators and types: For example, the `if_icmpgt` instruction can be used to jump if the second `int` value on the stack is greater than the first value (see **Figure 3.5**).Other conditional jump operations include `if_fcmpgt` for `float` values, `if_icmpeq` for 'equals', etc. To simplify the bytecode interface, we introduced several overloaded comparison operators that can be used as an alternative. Combined with the existing `ifeq` ("if true

```
iconst_1
iconst_2
if_icmpgt else // jump to label "else" if 1 > 2
  ...
else:
  ...
```

Figure 3.5 A simple "`if`" statement expressed as regular bytecode.

```
iconst_1
iconst_2
gt
ifeq else // jump to label "else" if 1 > 2
  ...
else:
  ...
```

Figure 3.6 A simple "`if`" statement expressed using the reduced instruction set.

jump") and `ifne` ("if false jump") instructions, these can replace the standard conditional jump instructions. For example, the overloaded `gt` comparison instruction can be used to replace the highly specialized `if_icmpgt` instruction (see **Figure 3.6**). Although this representation means that we now require one more instruction in the intermediate representation, it also means that we only need a handful of different instructions to express all combinations of primitive types and comparison operators.

We distinguish between a **return instruction** that returns a value (`ireturn`, `lreturn`, etc.) and a return instruction that returns no value (`return`). An additional pseudo-instruction, `xreturn`, can be used to return a value of *any* type.

3.4 Rewriting using the reduced instruction set

Using the reduced instruction set means that code generation becomes less complicated, as it relieves the code generator from specializing the emitted instructions. For example, rewriting the minus operator to bytecode can be expressed as a single, **type-agnostic** rewrite rule:

```
emit-minus: ⟦ e1 - e2 ⟧ → ⟦ <emit-java> e1; <emit-java> e2; sub ⟧
```

This rule emits a subtraction operation on any two expressions *e1* and *e2*. For the right-hand side of the rule, the `<emit-java>` function is called to emit instructions for the two sub-expressions. The resulting bytecode will push these two values on the stack, which are then consumed by the overloaded `sub` pseudo-instruction.

To generate regular, type-specialized bytecode, the given `emit-minus` rule would not suffice. Instead, it would need to check the operand types to select the properly typed instruction variant. Using overloaded instructions, we can express the minus operator and twenty other Java operators using single-line rewrite rules instead. While the instruction selection logic must still be implemented in the compiler (as discussed in Section 7.5.2), this technique allows us to encapsulate this aspect. This simplifies the compiler implementation, and relieves external code generator applications from this task.

Multiple type-specific instructions are often required for code generation tasks that require more complicated bytecode patterns, such as iteration, creating a new array instance, or modifying a field. Such patterns often use local variables (`load/store`) or stack operations to maintain state. For example,

Java expression	Regular bytecode	Reduced instruction set
`intVar++`	**`iload_1`**; **`dup`**; **`iconst_1`**; **`add`**; **`istore_1`** or, more specifically: **`iload_1`**; **`iinc`** `1 1`	**`load`** `intVar`; **`dup`**; **`inc`**; **`store`** `intVar`
`longVar++`	**`lload_2`**; **`dup2`**; **`lconst_1`**; **`ladd`**; **`lstore_2`**	**`load`** `longVar`; **`dup`**; **`inc`**; **`store`** `longVar`

Figure 3.7 Instructions generated for seemingly similar Java expressions. This example assumes that there are two variables declared: `longVar` of type `long` and `intVar` of type `int`.

the Java increment operator makes use of such operations (see **Figure 3.7**). To implement this operator in bytecode, a local variable needs to be loaded, incremented, and stored. As an increment expression may appear as a subexpression, it must additionally maintain (i.e., duplicate) the value on the stack. In regular Java bytecode, these four operations require type-specific instructions, and must carefully be selected depending on the type of the variable in question. Using the reduced instruction set however, the operations can be performed in a type-agnostic fashion. This results in the same generated bytecode, regardless of whether the variable is of type `int` or `long`. Therefore, this operation, too, can be compiled using a single rewrite rule:

```
emit-increment: ⟦ x++ ⟧ → ⟦ load x; dup; inc; store x ⟧
```

Additional rules are still required for the related pre-increment operation (i.e., ++x), pre-decrement operation, and post-decrement operation. Furthermore, such rules are not just required for local variables, but also for arrays, static fields, and instance fields[6]. Using the regular bytecode instruction set, this would lead to 60 rewrite rules that all generate a subtly different sequence of instructions. Implementing this would be very tedious and error-prone. An abstraction could be sought over such rules, to limit the amount of coding work, but would still require intensive unit testing for the different cases. The reduced instruction set on the other hand already defines a well-encapsulated abstraction for this, and is available for direct use in code generation within the compiler and for external preprocessor applications.

3.5 Integrating Java and bytecode

Bytecode has many of the same object-oriented units of code as Java: classes, methods, and fields. In the combined language, we want these to be interchangeable: Allow bytecode classes with Java methods, Java classes with bytecode fields, etc. These units of code are straightforward to integrate in the syntax, and they can simply be handled and compiled separately by the compiler.

Mixing the two languages beyond the method level is less obvious, and required us to make some design choices. In Java, a method body is a treelike structure of statements that may contain leafs of expression trees. Bytecode method bodies can be described as a flat list of instructions (that may include nested exception handling sections). Perhaps the most simple form of mixing at this level would be to allow bytecode fragments in place of statements, and vice versa. This – amongst other things – allows separate compilation of a DSL statement (directly to bytecode), or basic insertion of source code into compiled

[6] In our actual implementation, we abstract over store and load operations for these entities, but this illustrates the number of different bytecode sequences may have to be generated for a similar-looking expression.

```
class Foo {
  // A bytecode method with embedded Java
  add1(int i, int j : int) [
    stm: return i + j;
  ]

  // A Java method with embedded bytecode
  void add2(int i, int j) {
    [ load i; load j; add; xreturn ]
  }
}
```

Figure 3.8 Example of embedded bytecode and embedded Java: Two equivalent methods that add two integers, defined using different syntax forms.

methods. For other purposes, such as handling DSL *expressions,* it can be advantageous to extend support for mixing to the *expression level*. This is the approach we take here.

An overview of the complete syntax of the mixed language can be found in **Figure 3.2**. Essential are the definitions that allow class members, defined in either base language, to be included in classes of either language. At a lower level, statement- and expression-level mixing is achieved using square brackets (`[,]`) to inline bytecode in a Java method body. The `stm:` and `expr:` constructs embed Java fragments inside bytecode (see **Figure 3.8**). As will become evident in Section 3.5.2, this allows natural interoperability between constructs of the two base languages, adding to the use of local variables.

3.5.1 Stack-neutrality of Java statements

While Java expressions make intensive use of the stack (see Section 2.1.8), piling up intermediate values and popping off input values, this is not the case for statements. Java statements are compiled to a **stack-neutral** bytecode sequence: The stack after the execution of a Java statement is the same as it was before its completion. This ensures that all Java statements can be safely placed in a loop construct, without the risk of a stack overflow or underflow. Even more so, the JVM actually *requires* that methods have a fixed maximum stack depth, disallowing any loops with increasing stack depths.

For compound statements (e.g., `for`, `while`), stack-neutrality extends to the contained statements: The input stack of all nested statements must be the same as the stack outside the containing statement. This restriction goes hand in hand with the JVM restriction that at any point in a method, the types of the values on the stack must be known in advance (a requirement for static verification; see Section 7.5). For control flow this means that jumps to a location are only allowed if all incoming edges of the control flow graph have the same input stack. For statements and nested statements this is always true based on the property of stack-neutrality: A jump from one statement to another is therefore always allowed. Jumping into or out of expressions is not, unless the stack is explicitly made the same.

3.5.2 Interoperability between Java and bytecode fragments

The semantics of bytecode fragments inside Java, or vice versa, can be surprisingly simple and natural. To understand this, we will draw a comparison between the embedded code fragments and composition of regular Java statements and expressions. In regular Java, the stack only plays a role for the compiler, to combine expressions. In the bytecode language, this role is much greater and more direct, which would seem to cause a discrepancy. In the combined language, we will see that this role is also expanded, to allow interoperability between the two base languages.

```
// A bytecode expression in Java:
int i = [ ldc 1 ];

// A Java expression in bytecode:
[ expr: i + 1;
  store j;
]
```

Figure 3.9 A bytecode expression in Java, and vice versa.

```
while (x > 2) {
  [ ldc 1 ];
  if (x < 5) [ pop ];
  else return;
}
```

```
while (x > 2) { // <- stack mismatch at this point
  [ ldc 1 ];
  if (x < 5) [ pop ];
  else continue;
}
```

Figure 3.10 Using bytecode in place of statements: The program on the left could be compiled into a legal bytecode sequence, while on the right a program that is illegal since it increases the stack depth each loop iteration (and thus will not pass the Java verifier). Evidently, statements that modify the stack can lead to non-local errors that cannot be spotted at first sight. Because of this, we disallow such statements, and consider *both* these programs illegal.

Java expressions use the stack to interoperate: The stack is used for the input of operands, and to store the results of the expression (see Section 2.1.8). All Java expressions return a *single* value on the stack (with the exception of invocations of methods with a `void` return type). This means that it is possible to use a bytecode sequence that pushes a single value on the stack in place of a Java expression. We call this a **bytecode expression** (see **Figure 3.9**).

We also introduce the notion of **bytecode statements**; bytecode fragments that can be used in place of Java statements. For these, we pose requirement of stack-neutrality, which also exists for regular Java statements (see Section 3.5.1). Although this restricts certain otherwise valid programs from being compiled, we feel that the potential problems eliminated by this constraint outweigh this concern (see **Figure 3.10**). Requiring stack-neutrality of bytecode statements eliminates potential problems with statements that mutate values on the stack, and cause errors in other places in the method. It does not pose any substantial restriction when compiling to bytecode: Statements can still make use of the stack for intermediate values, as long as they restore the stack after completion.

Java can also be embedded in bytecode. Here the stack again has an essential role. In our syntax definition we distinguish between embedded statements and expressions, as Java statements have no (lasting) effect on the stack, while expressions push a value onto it (**Figure 3.9** shows a Java expression in bytecode). Besides readability, this distinction allows the use **promoted expressions** as both expressions and statements. Promoted expressions are expressions that may exist both in expression or statement form, such as method calls or assignments (**see Figure 3.2**). In statement form, these have no stack effect, while in expression form they may leave a return type on the stack.

In conclusion, using the syntactic and semantic constraints posed on embedded Java and bytecode, we believe that the result is a safe, yet powerful combination of the two base languages. The stack plays an important role in the interoperability of the two, as well as in the type safety of embeddings.

3.5.3 Shared local variables

Both Java and bytecode support the notion of local variables. As such, they can be used to share data between fragments of the two languages, adding to the interoperability available by use of the stack. We defined local variables to be referenced by name in the bytecode syntax, as is already the case in standard

```
void foo() {
  { // a Java block construct introduces a new local variable scope
    [ ldc 1; store var ];
  }
  // variable is now out of scope

  int var = 2; // defines a new variable
}
```

Figure 3.11 Local variables shared between in Java and bytecode. In this example, the Java scoping mechanism ensures there are two distinct variables declared: one scoped inside the block construct, and one defined after the block.

```
void foo() {
   [ stm: int var = 1; ]

   var++; // variable is still in scope
}
```

Figure 3.12 Embeddings do not create a scope by themselves.

Java (see Section 3.3.1). This allows us to enable sharing of local variables, simply by referencing the same name from within a bytecode and a Java fragment.

Scoping rules apply to local variables in Java. Using different scopes, distinct variables can be declared using the same name, possibly with different types. This means that the Java scoping rules are essential for proper sharing of local variables between Java and embedded bytecode. Therefore, we also apply the Java scoping rules to variables declared in bytecode (see **Figure 3.11**).

In standard bytecode, there is no notion of variable scoping (that is, they may be used anywhere within a single method). For embedded bytecode, we define that no scope is introduced, other than by the standard Java scoping constructs. Similarly, no scope is introduced by embedded Java, unless a block construct is explicitly used for scoping (see **Figure 3.12**).

3.5.4 Extended control flow mechanisms

In Section 3.3.8 we discussed that the bytecode control flow primitives, which cannot be mapped one-to-one to the control flow statements in the Java language. As such, they provide additional expressiveness to the language, allowing forms of control flow not allowed in mere Java code.

The most basic form of control flow in bytecode is the `goto` instruction, which can jump to any point in a method. The JVM specification requires that the types on the stack are the same for every entry point to a bytecode instruction. Jumps to a point in the program with a different stack height are simply not allowed. While this can pose some limitations on jumps to individual *instructions* – which have to be dealt by giving a proper verifier error message – this is not the case for jumps to Java *statements*. This is because Java statements are stack-neutral (see Section 3.5.1): The stack is the same at the beginning of each statement, including statements inside compound statements. Java already allows the placement of labels in front any statement (normally for use by `break` and `continue`). This means that we can also use these for jumps from inline bytecode to *any* Java statement (see **Figure 3.13**).

Java labels As demonstrated, the bytecode control flow instructions integrate well with the regular Java statements. This is in part because the regular Java language already provides support for labels, to support the labeled `break` and `continue` statements. These statements allow, under certain conditions, jumps to labeled locations. We extended the syntax definition to allow labels after a statement, rather

```
void findValue(int[][] matrix, int value) {
  for (int i = 0; i < matrix.length; i++) {
     for (int j = 0; j < matrix[i].length; j++) {
       if (matrix[i][j] == value) {
         System.out.println("Found it!");
         [ goto done ]
       }
     }
  }
  done:
}
```

Figure 3.13 Using an embedded `goto` instruction in Java. (Here, the `goto` is used as an alternative to standard Java `break` statement, but as demonstrated in Section 5.3 and 5.4, it also has more sensible applications.)

than only at the beginning of a statement. One minor issue with this language feature was the fact that Java allows duplicate labels to be defined, and only requires them to be unique with respect to nested statements. In other words, the following is legal in a standard Java method:

```
label1: int i = 1;
label1: i++;
```

As we extend the semantics of labels to allow jumps, we however require labels to be unique across the entire method. The fact that Java does not demand this can be considered an extremely esoteric language feature, which is rarely used in normal Java programs (we certainly have never encountered such a program). As such, we feel that the restriction to allow only unique labels in the Java/bytecode language is a necessary step that does not form a impediment for regular applications.

Try/finally and jumps Another issue we ran into with the mixed language is the semantics of control flow instructions used within the Java `try/finally` statement. The `try/finally` statement (see Section 2.1.6) ensures that the `finally` block of code is always is executed when the `try` block is exited. This can be because the `try` block is completed, or because an exception occurred. For the mixed language, this means that similar semantics must also be applied for the `goto` instruction and other control flow instructions. For both types of jump instructions, this can be effectively done by first jumping to the `finally` block, before the jump is made to actual the target destination (see **Figure 3.14**)[7]. For conditional control flow instructions, such as `ifeq`, this means that the jump is only "redirected" if the jump is actually made, otherwise the control continues normally.

Synchronized and jumps The Java `synchronized` block (see Section 2.1.4, 2.1.8) shares some of the semantics of the `try/finally` block. It always closes a monitor if a block is exited in any fashion. As such, the statement also shares its issue with bytecode control flow instructions. However, using the Java/bytecode language, we can define the `synchronized` statement using the following rewrite rule:

```
desugar-synchronized:
  ⟦ synchronized (e) bstm ⟧ →
  ⟦ try {
      Object locked = e;
      [ expr: locked; monitorenter ];
      bstm
    }
```

[7] Older Java compilers made use of the *jump-subroutine* (`JSR`) instruction for this, which is currently considered deprecated.

```
try {
  foo();
  ...
  [ goto end ];
} finally {
  bar();
}

end:
```

```
catch finally: * [ // if any exception occurs, goto finally
  invokestatic MyClass.foo();
  ...
  goto end_redirect;
]

end_redirect: // redirected jump to first execute "finally"
  invokestatic MyClass.bar();
  goto end;

finally:
  invokestatic MyClass.bar();

end:
```

Figure 3.14 Compilation of a mixed program with a `try/finally` statement and an embedded `goto` instruction (left) to pure bytecode (right).

```
    finally {
      [ expr: locked; monitorexit ];
    }
  ⟧
```

In this rule, the `monitorenter` and `monitorexit` instructions are used to respectively acquire and release the lock. Java's `try/finally` construct, which has no direct equivalent in bytecode, is used to ensure that it is always released. Essentially, this rule lifts the `synchronized` statement to a form of syntactic sugar. As syntactic sugar, it can make use of the implementation of the `try/finally` statement, sketched out above, which eliminates any issues with bytecode control flow inside the statement. (It should be noted that the given rewrite rule declares a variable `locked`, which may conflict with other variables in the generated code; in Section 5.2 we discuss how this issue can be handled.)

Chapter 4

Compiler Implementation Requirements

A regular Java compiler operates by rewriting Java source code to bytecode. Not all compilers for the JVM actually write the result directly to the binary class file format; sometimes an intermediate representation is used, such as the format provided by Jasmin or Soot [35]. The final stage of the compilation – writing this representation to the binary class format – is then carried out by a **bytecode assembler**. A good bytecode assembler provides access to the complete feature set of the class file format, and may provide class verification to aid bytecode engineering. The compiler for the mixed Java/bytecode language essentially unifies a regular Java compiler with such a bytecode assembler: It rewrites all Java code to bytecode, and acts as an assembler for both the resulting and inputted bytecode. In the remainder of this chapter we will discuss the requirements this poses on the compiler architecture. Furthermore, we will discuss the required techniques posed by the desire to allow separate compilation and composition of programs. In Section 7 we will discuss the actual design and implementation that follow from these techniques and requirements.

4.1 Bytecode verification

All bytecode compiled or written for the JVM has to pass the verifier (see Section 2.1.10). Implementors have to consider the security-related constraints the JVM poses on the generated code, as well as general type safety and stack consistency. The verifier included in the JVM will ensure that type errors and stack inconsistencies are detected before a program is executed. However, this is only intended as a means of providing security and reliability for the end-user, and not so much to help find programming errors. Many JVM implementations also allow the user to switch off the verifier, in exchange for a small performance boost. The JVM verifier only verifies a method "just-in-time" (i.e., when it is first executed), and may not verify methods that are never executed. Furthermore, the error messages produced by the JIT verifier are not very user-friendly, as they typically lack context information necessary to clearly identify the problem. For example, a typical verifier error message is "Expecting to find integer on stack." This message does not include the offending instruction, or its exact location in the program. Additionally, the produced message may differ across JVM implementations. All this makes the built-in JVM verifier not directly useable for writing or modifying programs in bytecode (i.e., **bytecode engineering**).

4.1.1 External verifiers

In addition to the JIT verifier that exists in the JVM, stand-alone verifiers also exist, which are more useful for bytecode engineering. A notable verifier in this category is **JustIce** [16], a verifier included in BCEL, an open-source project sponsored by the Apache Foundation. This verifier can report verbose error messages, as well as warning messages. As such, including it in a compilation chain can help prevent errors, and provides feedback to the programmer.

4.1.2 Compiler-integrated verifier

A verifier may also be coupled with a Java compiler. The advantage of this integration can be that the verifier has more context information at its disposal. For this project, a tightly integrated verifier enables us to extract typing information from fragments of bytecode. These fragments make use of context information of the surrounding code: Defined fields, methods, local variables, etc. Type-checking bytecode fragments is essential for the stack type restrictions we placed on embedded bytecode (see Section 3.5.2). Furthermore, this type information is essential for the code generation process, supporting for example method overloading resolution.

The context provided for the verifier can also be in the form of information about the original, uncompiled code. Any generated error messages generated can then refer back to the original code, as well as the generated bytecode. Because of the large focus on bytecode in our project, there are two roles for the verifier. One is to aid in error detection in bytecode in the source files. The other is to detect errors in the *generated* bytecode. The latter is especially useful in the development of the standard Java code generation component of the compiler. The additional context information also allows static analyses to be performed at the bytecode level on entities that normally only exist at compile-time, such as the read-only (`final`) local variables or anonymous classes.

An integrated verifier can also be more easily adapted to process a modified form of bytecode. For the purpose of generating and composing code, we introduced a set of pseudo-instructions (see Section 3.3). These abstractions are compiled to regular bytecode in the final phase of the compiler. The integrated verifier either can run before this transformation, or it can be used as a part of it. We have chosen for the latter approach; using the verifier to provide typing information for the compiler. More details on the principles and actual implementation of this inferencing process can be found in Section 7.5.

4.2 Integration of debugging information

A Java compiler may include debugging information about source files in compiled classes. This information is used, among other things, to allow debuggers to step through the source code and to provide tracing information for runtime exceptions (see Section 2.1.9). Since debugging information is optional, many third-party compilers do not generate it. For a preprocessor, which generates Java code and uses a separate compiler to compile the result (see **Figure 4.1**), including the correct line numbers is generally not possible. Instead, debugging information is included for the *generated* code. As a single line of code in the original source file can expand to a large number of generated lines of code, the position information of the generated code is of little value to the end-user of a language extension. Accordingly, any compile-time errors will refer to the generated code (see **Figure 4.2**), as will any run-

Foo.mylang:

```
1 public class Foo {
2   void bar(BarList list) {
3     foreach (Bar b in list) {
4       b.print();
5     }
6   }
7 }
```

Foo.java:

```
1 public class Foo {
2   void bar(BarList list) {
3     Iterator it = list.iterator();
4     while (it.hasNext()) {
5       Bar b = it.next();
6       b.print();
7     }
8   }
9 }
```

Figure 4.1 A preprocessor rewrites a `foreach` (left) statement to regular Java code (right). (Note that Java 5 actually introduced similar syntactic sugar in the form of the *enhanced for loop;* see Section 5.4.)

```
Foo.java:5: cannot find symbol: method next()
```

Figure 4.2 A compile-time error from the Java compiler, directed to the generated code. This may confuse the end-user, as the original user code did not include a call to `next()` on line 5.

```
class Merged {
  trace (void method1 ... @ File1.java:10:2) [
    void method1() {
      ...
    }
  ]
  trace (void method2 ... @ File2.java:6:2) [
    void method2() {
      ...
    }
  ]
}
```

Figure 4.3 Explicit source location information included within a Java file. This Java file is generated from two input files, which is explicitly noted in the `trace` tags. This information is usually included at all levels of the program (i.e., at every class, method, statement, and expression).

time errors. One workaround that is sometimes applied by Java preprocessors is distributing whitespace so that the intermediate code uses the same line numbers. This is however limited to in-place code generation and can only preserves line numbers (not column numbers). When more than a few lines of code needs to be generated however, this approach also becomes less manageable and harder to debug.

Other third-party compilers, such as Kawa [10], compile directly to bytecode and directly generate proper debugging information. Our mixed language compiler needs to allow various separate compilation scenarios, perhaps forming the back-end of a preprocessor. Therefore, rather than extracting the debugging information from the source file, we allow the inclusion of explicit source information as annotations in the language. This **source tracing information** can be automatically generated and maintained throughout the compilation process [50]. This makes it is possible to generate debugging information that refers to the original source code rather than to the intermediate source code. For third-party compilers this simplifies the process of including debugging information, as they are no longer required to directly generate Java bytecode with debug symbol tables.

Compilers operate by rewriting source code to compiled code in multiple steps. Every source construct undergoes several analysis and transformation steps, such as desugaring, before it is finally translated to compiled code. During every such translation step a reference to the original source code location must be maintained with the translation result. For preprocessors this information must be maintained as metadata accompanying the generated code. This information can be maintained in separate tables, mapping line numbers from the source to the generated code. However, as with exception handler tables (see Section 3.2), it is more convenient to place this information *inline*. This enables quick manipulation of the source code, without having to maintain and update separate tables. Like for exception handlers in bytecode, we introduce a new language construct to specify this information (see **Figure 4.3**).

In conclusion, we require the compiler to maintain source code information during the complete compilation chain. This allows for generation of debugging information, and as we have seen in Section 4.1.2, this context information can be used by the verifier and code generator to generate detailed error messages. These can be provided with the original source file, line, and column number, and can include the offending source and generated code. These features are essential for high-fidelity compilation.

Chapter 5

Application in Developing New Languages

In this chapter, we will show how the mixed Java/bytecode language can help implement various language features and applications. The language can be used to provide certain features only available in bytecode, without introducing the complexity of full-fledged bytecode compilation. This can be of benefit for the complexity of the code generator or the performance of the generated code. We will also introduce patterns that rely on the interaction between the two base languages, to help accomplish common tasks in developing new languages or language extensions.

5.1 Language embedding

5.1.1 Motivation

Domain-Specific Languages (DSLs) offer specialized functionality and syntax for a specific domain. They provide a more natural way of programming for such a domain, and tend to be less verbose than general-purpose languages. This leads to an improvement in maintainability, reliability and developer productivity. Furthermore, a good code compiler can apply domain-specific optimizations to the code to achieve improvements in performance and resource utilization [61,69]. However, DSLs lack the general capabilities of general-purpose language. A language for text-processing and manipulation may lack the capability to create a graphical user interface, for example. For this reason, they are rarely used by themselves, but instead in combination with other languages.

A technique to combine the expressiveness and the general capabilities of different languages is through embedding a language: A DSL may be introduced as a language construct within a general-purpose language. In addition to extending the language syntax, its semantics will also have to be adapted to support the new constructs. Embedded DSL constructs need to be **assimilated** to be integrated into the host language: They must be rewritten to language constructs in that language, such as library calls or perhaps an outright translation. Since the DSL expressions were not directly written in terms of the general-purpose language, it is possible to perform optimizations or verifications at this stage. Standard libraries for queries on XML, for example, can only validate and interpret API calls at run-time, rather than at compile-time.

5.1.2 Applications

Embedded domain-specific languages can come in various forms, such as embedded database queries, or direct support for regular expressions (**Figure 5.1**). This can mean advantages in terms of performance and reliability, as compile-time optimizations and verifications specific for the DSL can be performed. **MetaBorg** [4,47] is an approach that provides generic facilities and techniques for embedding languages. It uses a modular syntax definition formalism (SDF [67]) to provide grammars extensions, and

```
String getShortDomain(String url) {
  return url ~= /
    ^(http|ftp):\/\/ # ignore protocol (e.g., http://)
    [\w\.]*          # ignore subdomains (e.g., www. or www.forum.)
    \.\w+\.\w+       # match second-level domain name
  /w;
}
```

Figure 5.1 An embedded regular expression in a Java method. It is implemented in a fraction of the Java code that would be required for this, and may have better performance characteristics than a hand-coded alternative with (expensive) string operations in Java.

methodologies for extending the type checker of the host language. Recent applications of MetaBorg included **JavaSwul** for assimilating a user-interface language into Java, which provided a concise new way to access the user-interface API. Another recent application was **JavaRegex**, which integrated regular expression into Java, with support for compile-time verification. MetaBorg is aimed at assimilating the embedded domain code to existing API calls and other constructs in the host language, while leaving the host language undisturbed. This transformation allows programs to be compiled by an existing compiler for the host language.

5.1.3 Challenges

One immediate complication that often arises with language embedding and assimilation, is the fact that DSL *expressions* often need to be translated to multiple *statements* in the host language. For example, given a regular expression (see **Figure 5.1**), the assimilation could produce several specialized statements, intermediate variables, and method calls. The originating regular expression itself however is placed in a Java expression, which cannot contain any statements. In other words, the expression cannot be simply replaced in-place in the generated code.

In many imperative languages, including Java, statements are more *expressive* than expressions. Among other things, they allow variable declarations and can use various control flow constructs. Statements may embed other statements or expressions, while expressions can only contain other expressions. Still, language embedding at the expression-level is often required, to return a value from the embedded language context. An expression can directly return the result of a database query or regular expression, for example, which would be tedious to do at the statement level.

One technique to overcome the problem of expression-level embedded language constructs is **Statement-lifting**. This is a transformation that moves the translated form of an expression to the statement level. Sometimes this process is performed in a separate transformation step, rather than on-the-fly, to avoid complication of the assimilation process. MetaBorg provides a systematic approach for this technique in the form of **eblocks** [47], a technique that allows a programmer to use an intermediate language formalism to include statements in expressions. An eblock is a language construct that can be used in place of an expression, and takes the following form:

```
{| statements | expression |}
```

From this expression, the *statements* component allows statements to be used in place of the expression. The resulting value of the eblock expression is the value in the *expression* component. To transform this to regular Java, the embedded statements are simply lifted to the statement-level, before the statement in which the eblock occurs. We applied this technique to assimilate the C# `as` type-casting operator into Java (see **Figure 5.2**). This operator can be rewritten to a single statement and an expression. While in

```
// 1. 'As' operator
String s = e as String;
```

```
// 2. 'As' operator rewritten inline using eblock
String s =
{| Object temp = e;
   String lifted = temp instanceof String ? (String) temp : null;
 | lifted
 |}
```

```
// 3. 'As' operator rewritten to pure Java
//    (using statement-lifting to rewrite it to separate statements)
Object temp = e;
String lifted = temp instanceof String ? (String) temp : null;
String s = lifted;
```

Figure 5.2 Rewriting the `as` operator to pure Java using statement-lifting. The `as` operator [42] performs a checked cast operation on an expression *e*. If *e* does not have the expected type, it returns `null`. When expressed in pure Java, a temporary variable is declared to store expression *e*, ensuring it is only evaluated once.

```
// 1. 'As' operator in for loop
for (String s = start; s != null; s = next() as String) {
  ...
}
```

```
// 2. 'As' operator rewritten to pure Java
//    (using simple statement-lifting, the resulting program may be incorrect)
Object temp = next();
String lifted = temp instanceof String ? (String) temp : null;
for (String s = start; s != null; s = lifted) {
  ...
}
```

Figure 5.3 Statement-lifting can be problematic, as the unlifted expression is semantically in a different context. In this example the `as` operator cannot be directly moved out of the loop, without changing the meaning of the program.

principle this process of **statement-lifting** is a simple transformation, it requires consideration of all statements and expressions the eblock could be embedded in. They may appear as subexpressions, but also as the condition of a loop construct or `if` statement, as the argument of a `throw` or `return` statement, etc.

Even when considering all the places that an expression may appear in, simply moving the statements for an assimilated expression to the statement level still has its limitations. Another complication is that by moving the translated code, the execution order changes. Because of side-effects that the code may exhibit, this can lead to undesirable results. For example, the lifting pattern from **Figure 5.2** cannot be safely applied inside a looping statement, as illustrated in **Figure 5.3**. Another language construct that this technique is not be applicable is inside short-circuiting operators: Operators such as the Java 'and' (`&&`) and 'or' (`||`) operator will only evaluate their contained expressions if they require its value. A further (non-trivial) transformation is required to prevent the lifted statement from being evaluated in these cases. In other words, not only does proper statement-lifting require syntactic knowledge of the complete language, it also requires profound knowledge about its semantics.

```
String s = [
  expr: e;
  dup;
  instanceof java.lang.String;  // determine if 'e' is a subtype of String
  ifeq else;
    checkcast java.lang.String; // if it is, cast to String
    goto end;
  else:
    pop;                        // otherwise, replace it by null
    ldc null;
  end:
];
```

Figure 5.4 Inline compilation of the `as` operator to a bytecode expression.

```
String s = [
  stm:  Object temp = e;
  expr: temp instanceof String ? (String) temp : null;
];
```

Figure 5.5 The **as** operator, expressed as Java embedded at the bytecode-level. The inner Java expression determines the value (and type) pushed on the stack, which in turn is the value assigned to string *s*.

5.1.4 Implementation

In bytecode, there is no statement level or expression level. This distinction only exists in Java source code, and is simply a consequence of the Java syntax. By embedding bytecode in Java source code, we can effectively eliminate this limitation without further effort. Bytecode expressions can make variable declarations and other features available that are normally unavailable to expressions. Based on this, we can directly compile the operator to bytecode *inline*, using embedded bytecode (see **Figure 5.4**). The Java/bytecode compiler does not change the execution order, but will simply type-check the expression and integrate the bytecode with the surrounding code at compilation time (see Section 3.5.2).

Building on the principle of embedded bytecode, we can go one step further and also apply embedding of Java source code in bytecode: We can place Java statements and expressions inside a bytecode block. This means we can inline Java statements in bytecode expressions, effectively allowing statements in place of expressions. Using this pattern, we can successfully achieve the effect as we were trying to accomplish with eblocks (see **Figure 5.5**). This pattern avoids the problems of statement-lifting, as it can be applied in place of *any* expression, and does not require changes in the execution order. The compiler will simply compile the code in-place to bytecode, without any re-ordering. Furthermore, the syntax is arguably as convenient in use as the eblock syntax, even though it only makes use of the standard embedding constructs we defined in Section 3.5. The implementation follows naturally from the mixed language concept, and does not require specific handling for special cases as required with statement-lifting.

There are a number of special forms of the eblock construct. For example, the *post-eblock* can be used in cases where it is desirable to evaluate the expression before, rather after than the statement. Another form first evaluates a statement, then an expression, and then another statement. With the bytecode-embedding syntax, these forms can be expressed using the same basic embedding syntax. The only restriction is that such a bytecode expression must push only a single value on the stack (which can be in the form of a single expression). This is enforced by the type-checker, rather than by the syntax definition. Statements on the other hand, have no effect on the stack (see Section 3.5.2), which means a bytecode expression may include as many statements as desired, and in any order.

In conclusion, we can assimilate the `as` operator using a single rewrite rule. The mixed language concept ensures proper integration in the surrounding code, without changing the execution order:

```
assimilate-operator:
  ⟦ e as t ⟧ →
  ⟦ [  stm: Object temp = e;
       expr: temp instanceof t ? (t) temp : null;
    ]
  ⟧
```

5.2 Local variables in generated code

5.2.1 Motivation

A classic issue in code generation is the use of local variables. Even for assimilating small fragments of embedded code, the use of intermediate local variables may be a necessity. Java is a statically typed language that does not support name shadowing (i.e., declaring a new variable with a name that is already used in an outer scope). Generated local variables must therefore be provided with a unique name (or at least a name that does not shadow other variables), and must be explicitly typed.

5.2.2 Applications

As our next running example, we will take a stab at another operator from the C# language, which shows similarities with the `as` operator in Section 5.1. The null coalescing operator, or simply the **coalescing operator**, is specifically designed to handle conversions of **nullable types** [42]. Nullable types are types that may be assigned the `null` value, and are often used in databases. In Java, all object types are nullable, but the numeric, primitive types are not. Therefore, types provided by Java as equivalents to those in the database world may not always be nullable. Java however does provide primitive wrapper types that may be used as an object reference (i.e., nullable) representation of the primitive types (see Section 2.1.3). Due to performance constraints however, and the fact that most library functions use primitive types rather than their wrappers, having to convert them to their primitive equivalent is usually unavoidable. This conversion is facilitated by Java's support for auto-boxing and -unboxing (see Section 2.1.3). This makes it possible to directly and implicitly convert from a primitive wrapper type to a regular primitive type:

```
Integer i = new Integer(3);
int j = i;
```

Unfortunately, a limitation of auto-boxing is that if the converted wrapper type is `null`, there is no corresponding primitive value. Since there is no general, safe default value that could be used instead, and because this is an exceptional case, a runtime exception will be thrown if this happens:

```
Integer i = null;
int j = i; // throws an exception!
```

This is where the coalescing operator comes in. This operator, denoted as `??`, provides a clean solution for this, by specifying a default value in case the input is `null`:

```
int j = i ?? 0; // will result in 0, instead of an exception
```

Like the `as` operator we have shown earlier, the coalescence operator is embedded as an expression within the host language (i.e., Java), and also embeds host language expressions within it. This is a common pattern in language embeddings and extensions.

5.2.3 Challenges

A somewhat naive implementation of the coalescing operator may look something like this:

```
assimilate-operator1: ⟦ e1 ?? e2 ⟧ → ⟦ e1 != null ? e1 : e2 ⟧
```

However, care should be taken to not make assumptions about the embedded expressions *e1* and *e2*. They may be of various different natures: Each expression may be a regular Java variable, a method call, embedded bytecode, or perhaps expressions in the form of another language extension. Foremostly, we must consider that they may exhibit **side-effects**. If the operator is applied to an expression with any form of side-effects, perhaps reading a string from a file or printing something to the screen, it must be evaluated only once. In the rewrite rule above, expression *e1* could be evaluated twice.

Using a temporary variable, we can ensure that the expression is evaluated only once. We can do this by assigning expression *e1* to a variable (expression *e2* on the other hand is only evaluated when used). By applying the pattern from Section 5.1, we can place a local variable declaration inside a bytecode expression:

```
assimilate-operator2:
  ⟦ e1 ?? e2 ⟧ →
  ⟦ [  stm:  Object $temp = e1;
       expr: e1 != null ? $temp : e2;
    ]
  ⟧
```

Unfortunately, this does not yet solve the issue of giving a unique name and type to the temporary variable. Clearly, we cannot safely assume that the operator will only be applied of expressions of type `Object`. Nor can we simply assume that the name *$temp* will never be used elsewhere in the host program. The latter is an issue we conveniently ignored in the previous section, but it is of vital importance in a production environment. Some code generators attempt to create a unique name for fresh variables using dollar signs in this fashion, which is a valid, but rarely used character for Java identifiers. Of course, no guarantee can be made that the program that hosts a language extension does not also make use of such names.

In mature program transformation systems, such as Stratego/XT [3,4,47], various solutions exist for the general issue of naming and typing generated local variables. Stratego can automatically generate a name that is globally unique in the environment (see **Figure 5.6**). By use of a type-checker, extended for the adapted language, it is possible to retrieve the type of expression *e1*. Unfortunately such solutions require knowledge of the context of the assimilation (expression types, locals in scope), which may not always be available. Furthermore, not all code generators make use of such systems, and a type-checker may not always be readily available for a language. Here, we investigate an alternative method that does not require such information and technology.

```
assimilate-operator3:
  ⟦ e1 ?? e2 ⟧ →
  ⟦ [ stm:  t x = e1;
      expr: e1 != null ? x : e2
    ]
  ⟧
  where x := <new>;       // built-in function to give a globally unique name
        t := <type-of> e1 // use a type-checker to determine the type of e1
```

Figure 5.6 Using a Stratego built-in function to name a fresh variable, and an external type-checker to type it.

```
assimilate-operator-using-implicit-type:
  ⟦ e1 ?? e2 ⟧ →
  ⟦ [ expr: e1;
      store x;
      expr: e1 != null ? x : e2
    ]
  ⟧
  where x := <new>
```

Figure 5.7 Using the overloaded `store` instruction to store the temporary variable.

5.2.4 Implementation

Whereas variables in Java have to be explicitly declared with a complete type and name, variables in bytecode are implicitly typed. Recall that bytecode instructions for handling local variables are only explicitly typed for the primitive types, but not for object types (Section 3.3.1). Furthermore, the `load` and `store` pseudo-instructions can be used for both primitive types and object types. We can use this to our advantage for implicitly typing a variable. Building upon the pattern from **Figure 5.6**, we can rewrite the variable declaration to bytecode (see **Figure 5.7**). This pattern eliminates the need to define or extend a type-checker for the language extension. In a complete compilation chain, this desugaring can be applied before type-checking. It results in an expression with an associated local variable that can be type-checked after it is generated, by the standard Java/bytecode type-checker (discussed in Section 7).

Introducing the var operator As the use of bytecode increases, so does the generally perceived complexity of the rewrite rule. Instead of directly using the `store` instruction, it is also possible to define a rule for introducing syntactic sugar for the implicitly typed variable declaration:

```
desugar-var: ⟦ var x = e; ⟧ → ⟦ [ expr: e; store x ] ⟧
```

The `var` operator is a useful tool for both generative as well as general-purpose programming. A similar feature is currently being proposed for both the C# and C++ languages, and is already supported by the Scala language. Such statically typed languages greatly benefit from this operator, as they often require the use of explicitly typed local variables, even if the type is already clear from the context (see **Figure 5.8**). In such cases the explicit type specification does not help maintainability, but instead clutters the source code.

```
var s = "String";      // assign a string
var ints = new int[5]; // assign an array

int i = s; // results in a compile-time error: "Incompatible types"
```

Figure 5.8 Example uses of the `var` operator.

```
assimilate-operator-using-the-stack:
  ⟦ e1 ?? e2 ⟧ →
  ⟦ [ expr: e1;                                    // push onto the stack
      stm:  if ([dup] == null) [ pop; expr: e2 ] // replace with e2 if necessary
    ]
  ⟧
```

Figure 5.9 Using the stack instead of a temporary variable. In this implementation, the expression is pushed on the stack, duplicated (`dup`) and compared to `null`, and replaced with *e2* as necessary.

By applying the `var` desugaring after the regular assimilation rules, we can use it in the rewrite rule for the coalescing operator. Using the `var` operator, our rewrite rule finally becomes:

```
assimilate-operator-using-var:
  ⟦ e1 ?? e2 ⟧ → ⟦ [ stm: var x = e; expr: x != null ? x : e2 ] ⟧
  where x := <new>
```

5.2.5 Alternative implementation: a context-free solution

A further step can be taken to eliminate the need to generate a name for the temporary variable, in our case generated by the use of the `<new>` function. This approach can be taken in an implementation that does not make use of a code generation platform that provides such a function, or if little is known about the environment in which the operator occurs. Instead of a local variable, we can also use an unnamed stack value (see **Figure 5.9**).

Admittingly, this solution is no longer as intuitive as the previous solution, and it is conceptually harder to understand due to the use of the stack. However, the advantage of this approach is also obvious: It no longer depends on some library or algorithm to provide a unique, new variable name. And perhaps more importantly, this is a truly **context-free solution**: This rewrite can be applied to any code fragment, without the need to know its environment. Any 'dumb' rewriting system or code munger, even if it is based on simply replacing strings, can safely perform this transformation.

For compound statements (statements that contain other statements), this may not be the preferred approach. Generally, care should be taken to not violate the stack-neutrality property of such statements (Section 3.5.2). For this, the stack height must be the same at the beginning of a statement as at the ending of a statement, and must remain the same height for any statement that is nested within it. If this is not the case, jumps into and out of the code are limited to locations with the same stack height. In short, it is not recommended to use the stack to store intermediate values of compound statements. Of course, depending on the source language, and how it applies control flow, stack-neutrality may not be an issue.

For expressions, as we have seen here, or for statements that do not contain other statements, the stack is a viable alternative to local variables. It eliminates the need for an explicitly named variable, and the use of the `store` and (implicit) `load` operation used earlier. As such, this approach results in a slightly smaller and – depending on the JVM implementation – often faster-running instruction sequence [35].

5.3 Finite state automata

5.3.1 Motivation

A **finite automaton** or finite state machine is a way to model behavior composed of a finite number of states and transitions between those states. One of the states is the starting state. Each state defines transitions to other states, based on the next input. A transition has two components: a condition and an action. If the condition is met, the automaton will perform the action and will enter the next state. This cycle repeats until an ending state is reached.

Java does not provide explicit support for finite automata. Instead, they are often built using general-purpose language constructs. For example, one may use the `if` statement or the `switch` statement for evaluating conditions and determining the next state. Using a loop construct, or by applying recursion, this process can be repeated until an ending state has been reached (see **Figure 5.10**).

5.3.2 Applications

Parsers determine the grammatical structure of a sequence of tokens with respect to a given grammar. They are applied in compilers as the first step of reading and analyzing source code. Sometimes parsers are hand-coded in a general-purpose programming language. This can be a tedious task, especially when keeping an eye on performance. Such parsers are also hard to understand and maintain. For these reasons, parsers are often created using external tools. This way the programmer can use a high-level syntax definition language to define the grammar. A general parser library or program can then be used to process this grammar and create a finite automaton-like data structure, in the form of tables or graphs. A static parsing algorithm can use this data structure to parse any given grammar.

Performance of static, table-based parsing algorithms is often far worse than that of hand-coded parsers, because of their interpretive overhead [36]. This means they are not always a viable alternative for a hand-coded parser. For this reason, parsers are often generated using a **parser generator**, which generates a parser dedicated for a single grammar. A parser generator processes a grammar definition and generates code to perform the actual parsing. This is often done on the basis of a finite state

```
void stateA() {
  ... // process and store current state

  switch (readInputChar()) {
    case 'a': stateA(); break;
    case 'b': stateB(); break;
  }
}

void stateB() {
  ...
  switch (readInputChar()) {
    case 'a': stateA(); break;
    case 'c': stateC(); break;
  }
}

...
```

Figure 5.10 A simple, hand-coded finite state machine. It uses a `switch` statement to determine the next state based on inputted characters, and applies recursion for transitioning to the next state.

automaton. It will create a static structure of reading the next token or character, and define how this influences the current state. For example, when reading a language that supports strings of characters between double quotes, the parser will enter a state indicating an opening quote is encountered. Once a closing quote is encountered, it will exit this state, and transition to another state.

5.3.3 Challenges

ANTLR [36] is a commonly used parser generator that can generate source code in various languages from a grammar definition. Originally ANTLR only generated C or C++ code, but presently it can generate C++, Java, C#, or Python code. Using code generation, it does not only have better performance, but also is more flexible than a static, table-based approach [36]. The code ANTLR generates code for automatons uses a `switch` statement to jump to the current state, rather than using recursion for this purpose. This eliminates the overhead normally associated with method calls. It uses a single block of code for each state, which may be entered from several other states, and has several exit points to other states. Rather than directly jumping to another state, it will use a local variable to specify the next state, and will re-enter the `switch` to jump to this state (see **Figure 5.11**).

The `switch` statement by itself is an indirect and highly dynamic way of jumping to the next target state, which is not strictly necessary for this application. Not considering the jumps made to re-enter the `switch` (these are of little impact and can mostly be optimized away), the `switch` statement itself is compiled to several jump instructions for the CPU. Due to this indirection, it may not perform as well as a direct jump using the `goto` statement that can be used in the C++ or C# back-ends of ANTLR. The switch-based approach also requires several more bytecode instructions, decreasing the chance that the method is inlined by the JIT compiler. The developers of ANTLR have been experimenting with directly generating a complete bytecode program to avoid this overhead ("there is a goto bytecode of course") [38,39]. This would involve a complete rewrite of existing parts of ANTLR however, and may not be as attractive as the current solution that uses a string templating engine to generate Java.

5.3.4 Implementation

Using the extended Java/bytecode language, most of the existing parser generation logic can be maintained in the well-understood Java language. Such an implementation is already provided by the current ANTLR implementation. With its separate back-end and templating engine, it provides a good foundation for generating an adapted output language. By using the added functionality of the bytecode language, the finite automaton's control flow can be expressed in a more natural and better performing fashion. The control flow statements offered by Java are simply not a very good match with the control flow desired for the finite automaton. Therefore, before going into a specific solution for this case, let us first discuss the general implications of the control flow constructs as they exist in both Java and bytecode.

Structured and unstructured control flow Java provides many abstractions over bytecode, but by design hides some underlying primitives, such as a *goto* statement (as it is considered "harmful" for most applications [19]). Instead, Java provides several **structured control flow** statements, such as `while`, `switch`, and a "for each" loop. These constructs enforce a treelike structure of blocks, which is arguably easier to understand than a linear list of statements and jumps [19].

Java does not allow jumping to an arbitrary labeled location, but does provide the `break` statement for skipping the rest of a loop, and the `continue` statement for forcing the next loop iteration (see Section 2.1.5). These constructs also accept a label to allow exiting or restarting an outer loop. The exact

```
void match() {
  final int INSIDE_QUOTE = 0;
  final int OUTSIDE_QUOTE = 1;
  p = 0; // input position 0
  int state = 0;

  while (true) {
     c = nextToken();

    switch (state) {
      case OUTSIDE_QUOTE:
        switch (c) {
          case '"':
            state = INSIDE_QUOTE;
            consume();
            break;
          case EOF:
            return;
          default: // only accept quotes or EOF in this state
            error(c);
        }
        break;
      case INSIDE_QUOTE:
        switch (c) {
          case '"':
            state = OUTSIDE_QUOTE;
            consume();
            break;
          case EOF:
            return;
          default: // accept any character between quotes
            consume();
            break;
        }
        break;
    }
  }
}
```

Figure 5.11 A basic finite state automaton, generated for a simple grammar that accepts any character within quoted strings (e.g., `"foo""bar"` would be a valid input). Note how the outer `switch` statement is used to indirectly jump to the next state: At the end of every state, the program loops back to this switch to locate and execute the next state.

semantics of these statements depends on the context they occur in, because of differences in the various loop constructs (that may or may not include a counting expression, and have a condition at the start or at the end). Suffice it to say, Java has its share of *goto* alternatives.

The goto statement Theoretically, with structured control flow statements as provided by Java, it is possible to write every program that could also be written with the help of a *goto* statement. Usually, such a program is more readable than a "spaghetti"-like web of goto statements. (Which is exactly why Java forbids the use of the *goto* statement.) Of course, this only applies to human programmers. For use by the JIT or interpreter, all control flow statements are translated to simple branching instructions, which include the dreaded *goto* (see Section 3.3.8). At this level, structured control flow would be a convoluted and indirect representation that unnecessarily abstracts from the system's operation. Likewise, for generated code this structure is not advantageous either, as it unnecessarily restricts the form of the generated program [48]. Generated code does not necessarily follow the same structure as the original source code; different languages have different (high level) control flow constructs. This can make code generation to a treelike structure of Java control flow statements a nuisance, compared to the flat stream of bytes or statements that can be produced using a *goto* statement.

```
void match() {
  p = 0; // input position 0

OutsideQuote: // beginning state
  switch (nextToken()) {
    case '"':
      consume();
      [ goto InsideQuote ];
    case EOF:
      return;
    default: // only accept quotes or EOF in this state
      error(c);
  }

InsideQuote:
  switch (nextToken()) {
    case '"':
      consume();
      [ goto OutsideQuote ];
    case EOF:
      return;
    default: // accept any character between quotes
      consume();
      [ goto InsideQuote ];
  }
}
```

Figure 5.12 A basic finite state automaton parser, based on **Figure 5.11**, using the `goto` instruction.

By using the *goto* instruction in embedded bytecode form (see Section 3.5.4), a considerably smaller parser implementation can be produced (see **Figure 5.12**). Besides the length, the nesting depth and general complexity of the generated program is reduced. (Some might argue against the readability of this program, based on the long-standing dogma that the *goto* should never be allowed. We remind the reader that Dijkstra [19] made a plea for structured programming in a time when it was still uncommon, but not so much to entirely *abolish* the goto statement.) Generally, however, the readability of *generated* code is of little concern. The real advantage of this approach comes in terms of simplicity and performance, without having to resort to pure bytecode generation. The indirection of the state variable and `switch` statement from **Figure 5.11** are no longer an issue: At the end of each state, either the automaton is exited, or a *direct* jump is made to the next state. The new program's control flow structure is simply a closer match to the finite automaton concept.

5.4 Iterators and yield continuations

5.4.1 Motivation

Java 5.0 introduced the **enhanced for loop**, a new language feature that allows programmers to iterate over collections. This "for each"-like construct allows programmers to iterate over elements of a collection using the following syntax:

```
for (<loop variable> : <collection>) {
  <body>
}
```

The enhanced for loop can be used with any (collection) class that implements the `Iterable` interface. This interface in turn requires the definition of a class that implements the `Iterator` interface, which

```
for (String s : splitSpaces("foo bar") {
  System.out.println(s);
}
```

```
Iterator it = splitSpaces("foo bar");
while (it.hasNext()) {
  String s = it.next();
  System.out.println(s);
}
```

Figure 5.13 The enhanced for loop, in its regular form (left) and in its desugared form (right).

defines methods to allow traversal over the specific data structure [2,33]. The enhanced for loop is simply a form of syntactic sugar that abstracts over this API (see **Figure 5.13**).

Currently, Java provides support for iterating over arrays and most collection classes in the Java standard library. The loop construct is a standard way of iterating over any of these types. Sometimes iterators are also used for non-collection types. For example, one may use an iterator for reading lines from a file, or to construct an object for each string read from a file, or as a way to express infinite lists. Classes can also define multiple iterators, for example to allow *in-order* and *post-order* traversal over a tree, or reverse iteration over a list. (Currently, Java only defines basic iterators for most standard collection types, but more advanced iterators are provided by third-party libraries.)

Before Java introduced the iterator-based loop, they were introduced in other languages, such as Sather [30] and Python. These languages make heavy use of iterators, and provide an additional language feature to *define* and generate iterators: **yield continuations** (sometimes simply called "generators"). With this feature, it is no longer necessary to define a separate iterator class with all methods required to implement the `Iterator` interface. Programmers can simply define the complete iterator in a single method. At any point of this method, the `yield` statement can be used to return an element of the iterator (see **Figure 5.14**). It will then "yield" the control back to the client of the iterator. If the client enters the next iteration, the iteration method will continue execution at the last point of exit, from where it may yield more values, or exit the iteration. Thus, semantically an iterator operates as a coroutine.

5.4.2 Challenges

Various implementation options of yield continuations exist. One of them is low-level virtual machine support for continuations. The JVM could store the current context (such as the stack and all local variables), jump back to the calling method and restore this information when entering the iteration. Implementing this is rather complicated however, especially in face of the strict security demands on the JVM.

An alternative solution is to treat the language extension as a form of syntactic sugar, and rewrite it to the normal Java language. Many Java features are essentially syntactic sugar; examples include inner classes, operators like += and many control flow statements. This is an approach we took in an earlier

```
public static Iterable<String> splitSpaces(String input) {
  String[] parts = input.split(" ");
  for (String part : parts) {
    yield return part;
  }
}
```

Figure 5.14 An iterator definition that divides a splits a given string using the space character, and returns all tokens. The syntax used here adds an additional `return` keyword to the `yield` statement, indicating that it returns a value. This also ensures that the language can be parsed without making `yield` a reserved keyword, maintaining backwards compatibility with identifiers named 'yield'.

project [62]. To accomplish this, the method needs to be rewritten to a form that allows it to be entered and continued at every point where a `yield` statement occurs. At such a point it may then return the next value of the iterator, and allow it to be continued later to retrieve the next value. One way to implement this is to use a `switch` or `if` statement to jump to the position of the code where it was last exited. For the example in **Figure 5.14** this would mean that the `for` loop in the example input would need to be decomposed into separate statements, in order to allow continuation at any point halfway the loop. It is possible to use the `switch` statement (or multiple `if` statements) to simulate the looping behavior of the `for` statement, while still allowing it to be entered at multiple points. This way the control flow semantics of both the yield continuation and that the existing control flow statements would be maintained. Essentially this transformation means a reimplemention of all normal control flow statements in Java, only to support the new `yield` control flow construct.

The Yielder library [58] takes another approach and uses a dummy `yield()` method invocation rather than an actual `yield` statement defined in the language. The resulting code can be compiled by a regular Java compiler. Using bytecode-level modifications, the dummy invocations are then replaced with an actual implementation. This is an intricate process that requires the use of a bytecode manipulation library to generate a new class definition, with a manipulated control flow for the iterator method in question. Since the Java compiler is oblivious about the special semantics of the dummy invocations, it is unable to do proper control flow analysis during the compilation, which may lead to unexpected results. Furthermore, some Java compilers generate code that breaks the stack-neutrality property we established for statements (see Section 3.5.1). They may allocate intermediate values on the stack, which changes the stack height at the start of different statements. Sometimes this is applied as an optimization, after the code generation process is completed [35]. As the stack height must be the same at two points in a method to allow jumps, this means that the bytecode control flow can no longer be directly manipulated without risking a stack mismatch.

5.4.3 Implementation

We implement the `yield` statement using by rewriting the method to a small Java/bytecode class that implements the iteration interface. Rather than completely rewriting the source program's control flow statements, we will make minimal changes to the program by maintaining its structure. We begin our transformation by rewriting every yield statement to an entry and exit point of the method:

```
assimilate-yield: ⟦ yield return e; ⟧ → ⟦ state = nextState; return e; label: ⟧
where  nextState := <next-state>;       // determine and increment the state count
       label     := <next-state-label> // new label for this state
```

The yield statement makes place for a regular `return` statement that returns the yielded value. To allow the method to be resumed at this point as the iteration continues, a label is generated in combination with a new *state* constant.

Using a small jump table at the beginning of the method, this *state* constant is applied to jump to the corresponding label. Of course, since the method is exited every time a value is returned, all local variables must be stored in persistent fields. In a further rewriting step, all local variables are replaced by fields. Finally, to implement the complete iterator interface, static boilerplate code is added, forming a complete class (see **Figure 5.15**).

The amount of boilerplate code required to implement the standard methods of the iterator interface illustrates the amount of code normally required to implement an iterator by hand. Our approach

```
class Split implements Iterator<String> {
 int state;
 Integer value;
 boolean valueReady;

 String[]          m_parts;
 String            m_part;
 Iterable<String> m_numbers = numbers;

 public String next() {
   prepareNext();
   if (!valueReady) throw new
       NoSuchElementException();

   valueReady = false;
   return value;
 }

 public boolean hasNext() {
   if (!valueReady) prepareNext();
   return valueReady;
 }
}
```

```
 private void prepareNext() {
  if (valueReady || state == 2) return;
  if (state == 1) [ goto afterYield ];

  m_parts = input.split(" ");
  for (int part : m_parts) {
     m_part = part; // make persistent
     state = 1; // remember state
     valueReady = true;
     value = m_part; // yield value
     return;
afterYield:
   }
   state = 2;
 }

 public void remove() { // N/A
   throw new
     UnsupportedOperationException();
 }
}
```

Figure 5.15 Implementation of the iterator class from **Figure 5.14** in plain Java (using an embedded goto instruction). Lines of code in direct correspondence with the original program are shaded.

implements the interface using a central `prepareNext()` method that prepares the next element of the iterator, rather than directly implementing the `next()` method. We use this to implement both the `next()` and `hasNext()` methods, where we cannot make assumptions on the order they are called.

It is not possible to statically determine if the iterator will return a value or not, given a state (i.e., point in the iterator method). This can only be determined at runtime: Instead of yielding a new value, the method may throw an exception, or perhaps the method is simply exited. We accommodate for this fact by allowing the `prepareNext()` method to indicate whether or not a value is returned, using the `valueReady` field. This method is invoked by the `hasNext()` method, which is used to determine whether the iteration will continue (see **Figure 5.13**).

The resulting program makes use of the standard control flow statements, in combined action with unstructured control flow to continue at multiple entry points, in the form of the `goto` instruction. The embedded `goto` instruction is essential in a straightforward transformation of the yield statement. It allows us to maintain most of the form of the original iterator method. In our original implementation [62], written before the present study started, this was a different story entirely. It required quite a radical rewriting that had a detrimental effect on the source program's performance. **Figure 5.16** shows a reduction of over a third in the size for the language extension implementation, where we can use

Program	Total LOC	LOC without boilerplate code*
Using switch and loop	260	225
Using jump instructions [pct. of original]	164 [63%]	130 [58%]

* = i.e., excluding the static generated code as illustrated by the non-shaded lines in **Figure 5.15**.

Figure 5.16 Comparison of the lines of code (LOC) used for the different implementations of the yield continuation transformations, excluding any comments. Both were written using the same coding conventions and libraries, although the newer version includes 8 lines of extra calls to allow for source code tracing (Section 4.2).

embedded bytecode. The main reason for this is that we now only rewrite the yield statements to jumps, as opposed to rewriting the standard control flow statements in the iterator method to match the new control flow. It should be noted that not all of Java's control flow statements (about 15 in total) were supported in the original rewriting, as it proved quite tedious to do so. This of course is supported naturally with the new version, using embedded bytecode. Another difference is that the new version benefits from the compiler's ability to generate debugging information and location information for compile-time errors (see Section 4.2).

5.4.4 Alternative techniques

In addition to the `goto` instruction we applied in embedded code, Java includes several conditional jump instructions. For example, the bytecode equivalent of the `if` statement is the `ifeq` instruction. Switch statements are compiled to either the `tableswitch` or `lookupswitch` instruction. These instructions, while not offering a concept unknown to the Java language, allow further digression from the standard, structured style of control flow. The C++ language formed the inspiration of a great part of the Java language, and besides the `goto` statement, also offers an *unstructured* `switch` statement. While it is an esoteric feature, it is claimed to be of benefit in specific situations [53,54], such as in highly optimized sections of code.

Dewhurst [53] uses the unstructured `switch` statement to implement a post-order tree traversal. Traversals of this kind are often implemented using recursion and nested function calls, or using a data structure to simulate this. Another approach sometimes taken is to add pointers with the 'next' node to the tree data structure, to allow quick traversal on the basis of these pointers. Sometimes the traversal is not encapsulated, but implemented in-place in the traversing method. These approaches have their associated costs in either runtime or complexity. Dewhurst's approach is more in the line of a coroutine: The iteration occurs in a method separate from the action of the traversal. The `next()` method (see **Figure 5.17**) determines the next node of the traversal, while the caller uses this data to perform the

```
bool Postorder::next() {
  switch (pc) { // jump to the previous position of the while loop
     case START:
        while (true)
           if (!lchild()) {
              pc = LEAF;
              return true; // process leaf, return here later
     case LEAF:
              while (true)
                 if (sibling())
                    break;
                 else
                    if (parent()) {
                       pc = INNER;
                       return true;
     case INNER:       ; // process inner node, return here later
                    } else {
                       pc = DONE;
     case DONE:        return false; // traversal finished
                    }
        }
  }
}
```

Figure 5.17 C++ definition of a method that prepares the next item for a post-order traversal of a tree (adapted from Dewhurst [53]). This algorithm should be read as an ordinary `while` loop that traverses the three (ignoring the `switch`). The unstructured `switch` is a separate construct that is superimposed over this traversal to achieve coroutine semantics.

```
boolean next() {
  [ getfield Postorder.pc : int;
    tableswitch
       0 to 2 start leaf inner // switch targets
       default: done
  ];

  start:
    while (true) {
      ...
  leaf:
      ...
    }
}
```

Figure 5.18 Simulating an unstructured `switch` using embedded bytecode.

action. Using the unstructured `switch`, the loop that traverses over the tree can be continued at the last point it exited.

The `next()` method, as defined in **Figure 5.17**, is certainly an efficient and well-encapsulated way to do a post-order traversal. It is also, to say the least, quite hard to read by most standards. This may introduce a maintenance problem. Dewhurst [54] argues that this implementation is the most clear and straightforward way of implementing the traversal, even if it is complicated by the introduction of the switch statement to achieve coroutine semantics. He further argues that the assumed maintenance problem for this function is small when put into perspective to other approaches; it only requires changes when the `next()` method of the post-order traversal needs to be modified. Users of the `Postorder` traversal never need to see the definition, but will benefit from the simple interface.

We will definitely do not argue that a statement similar to the unstructured switch has a place in a language like Java, but it clearly can have its purpose in specific situations. In the domain of generative programming, constructs such as the unstructured `switch` can be used without trading simplicity for performance or risking a maintenance nightmare. With the proper abstractions, no human would need to see the generated, unstructured code. (That is, no human except those who have to maintain an optimizing code generator. Fortunately for us, we are usually happy to invest in extra performance, and some even see the extra complexity as an added incentive.)

The same effect as with the C++ unstructured `switch` statement can be achieved with embedded bytecode, using the `tableswitch` instruction (or `lookupswitch`, its more flexible but worse-performing cousin). **Figure 5.18** sketches how the traversal can be implemented using this instruction, with separate labels for the different cases. With the separation of case values and labels, this syntax may be somewhat less convenient than the regular switch statement, but again, with the application of generated code in mind, this should not be of consequence to the end user.

5.4.5 Optimization

We can use the `tableswitch` instruction instead of the `goto` instruction to implement the jump table of the yield continuation, as demonstrated in **Figure 5.18**. While a good JIT optimizer could theoretically optimize the rare pattern of the `if` and `goto`-based jump table at load-time, in practice they often do not, and depend on the compiler to do this. Furthermore, using the `tableswitch` instruction results in a smaller number of total instructions in the method, which may make the iterator method a candidate for inlining by the JVM.

```
public static Iterable<TreeNode> traverse(TreeNode root) {
  for (TreeNode child : root.children()) {
    for (TreeNode node : traverse(child)) {
       yield return node;
    }
  }
}
```

Figure 5.19 Recursive iterators suffer from quadratic time behavior, as they copy the stream of values yielded through the recursive invocation.

Still, a high-level language construct such as the yield statement may hide its underlying performance costs to the end user. When you realize the classes, fields and methods generated, you may think twice about using it for performance-critical sections of code. A tree traversal based on *recursive* use of iterators may be impracticable for this reason, as a new iterator is created for each level of recursion (see **Figure 5.19**). A non-recursive solution should be written instead. Of course, it completely undesirable that programmers have to fall back to more complicated, harder to maintain, or badly encapsulated solutions, just out of performance considerations. Therefore, it may be rewarding to implement inter-procedural optimizations, inlining iterations where possible. Because of the presence of virtual methods (see Section 2.1.2), dynamically loaded classes (see Section 2.1.11), and private fields that cannot be accessed from a different class, this is not a generally feasible solution. Specific for recursive iterators, however, it is possible to rewrite the iterator to a non-recursive variant. Jacobs *et al.* [67] propose a method where they explicitly allocate the recursion stack, eliminating much of the overhead associated with the recursion.

Another performance improvement may come from another direction: the virtual machine. There are currently plans for using escape analysis to enable stack-based allocation of objects (i.e., using the stack for data that can be proven not to be used elsewhere). This means future JVM versions may alleviate most performance concerns regarding iterator object allocations.

5.5 Compiling for customized JVM implementations

5.5.1 Motivation

Java was first developed by Sun Microsystems, and as such **Sun Java**[8] is the reference implementation of the Java Platform. Implementing and maintaining a Java Virtual Machine is hard and time consuming, and it is difficult to reach the level of maturity equivalent to that of the implementations by IBM[9] or even Sun. Still, alternative implementations exist, including many open-source alternatives that are only partially compatible with Sun Java and the JVM specification. The proprietary nature of Sun Java prevented many Linux distributions from including it by default, which in turn has had a detrimental impact on Java adoption.

In November 2006, Sun announced that their JVM implementation, compiler and class library would be made available as open-source software under the **GNU General Public License** [12]. An open-source JVM can form a stable basis for implementing new features and new languages. Although such additions can stand in the way with interoperability, this is less of a problem than with a completely custom-built

[8] http://java.sun.com/.

[9] http://www.ibm.com/developerworks/java/jdk/.

virtual machine. Specifically for the purpose of experimental implementations, IBM also provides the open-source Jikes RVM (Research Virtual Machine). This JVM implementation is written in Java, and is self-hosted on Linux/IA-32 and PowerPC operating systems [25]. Much research is being conducted in the area of adapting Java-like virtual machines; there have been over 150 publications using Jikes RVM alone[10].

5.5.2 Applications

There have been several proposals for extending the JVM and similar virtual machines. We will shortly discuss some of these here, to illustrate the different possible directions that can be taken in enhancing the JVM.

Functional languages often rely on recursion rather than loops, and could benefit from a **tail call** instruction or optimization. Without such a construct, each recursive call to a method increases the call stack, even if this is not strictly necessary for the application. This leads to increased memory usage (and possibly stack overflows) and lower performance. Clements and Felleisen [21] show how this could be implemented in a virtual machine such as the JVM.

For **dynamic languages**, such as Python or Ruby, method calls are resolved at runtime. Since the Java bytecode instructions for calling a method require a statically determined method signature (see Section 2.1.8), a considerable work-around is required to emulate this behavior using JVM. This requires additional code generation, and the use of reflection. Because of this (and in part because of how Python deals with primitive types) Jython is about 1-10 times slower than equivalent Java code [9]. With virtual machine support for a **dynamic invocation instruction**, this could be significantly improved. A native implementation can use more efficient data structures and caching. It may also apply existing optimizations, such as speculative method inlining, which already exist for virtual method invocations. In fact, present drafts of the proposal[11] to add such an instruction, build on the existing, explicitly typed `invokevirtual` instruction. The proposed instruction includes a speculative signature specification (see **Figure 5.20**), which allows for speculative optimization or selection of a specific overload. Finally, in addition to performance benefits, a well-defined 'invokedynamic' instruction could help future interoperability between languages when one or more are dynamic.

Java has a limited number of **primitive types**. The performance problems associated with the use of complex numbers, for example, have been well documented. As Java does not have native support for such types, they are often represented as objects, which means they can be two orders of magnitude slower than equivalent Fortran code [64]. Even with the aid of escape analysis in current and upcoming JIT compilers, this overhead can never be completely avoided. One technique suggested for helping in the performance is the introduction of complex numbers as a natively supported Java virtual machine

```
invokevirtual java.lang.System.out.println(String : void)
invokedynamic java.lang.System.out.println(String : void)
```

Figure 5.20 The proposed `invokedynamic` instruction has the same form as the existing invokevirtual instruction. The difference is, that if the input parameters or method signature do not exactly match the instruction, the `invokedynamic` instruction allows runtime handlers to perform the necessary conversions.

[10] http://jikesrvm.org/Publications/.

[11] See JSR 292: *Supporting Dynamically Typed Languages on the Java™ Platform*. http://www.jcp.org/en/jsr/detail?id=292.

type [63]. This would require the introduction of new instructions to support the added primitive type (or support for custom primitive types). Alternatively, Java could be extended with support for **multiple return values** for methods, allowing the multiple components of complex or custom types to be returned as stack-allocated primitives. Of course, this would require the introduction of invocation instructions with support for tuples or named 'out' parameters.

Another feature that has received some attention in recent publications is the addition of first-class JVM support for **transactional memory** [13,14]. This could constitute a safe new synchronization alternative in a future JVM implementation. Like for dynamic invocation, current implementations rely on reflection and/or (runtime) code generation and have a significant performance overhead. Often these cannot prevent unsynchronized access to objects as the client can bypass the synchronisation indirection. With full JVM support for this feature, a standardized, high-performance, and safe interface could be provided instead.

5.5.3 Challenges

Introducing new virtual machine features in the form of instructions means that it is necessary to create a specialized compiler that can handle these new features. Building and maintaining a complete compiler can be a daunting undertaking. A less-time consuming way can be to adapt an existing compiler. It can provide a good foundation for the implementation, as well as an existing language to build on. **Open compilers**, such as OpenJava [52], Polyglot [34], JastAdd [66], Kawa [10], or the Eclipse Compiler for Java [56] can help in this task as they are specifically designed with customization in mind. However, not all 'open compilers' are created equally: Few offer the possibility to customize the bytecode back-end. OpenJava and PolyGlot, for example, offer an extensive interface for extending the language at the compiler front-end, but only allow it to generate regular Java source code. This intermediate source is then compiled by a separate back-end, in the form of a regular, external Java compiler.

Open compilers that implement a complete transformation from source code to bytecode, do this in **multiple stages**. Generally, these include reading and parsing the file, semantic analysis, and bytecode generation. Some compilers include one or more additional stages, perhaps desugaring the code before code generation, or using an intermediate form before emitting the bytecode instructions. Many compilers also include a separate stage for optimizing the generated (byte)code, relieving the code generation stage from having to emit well-tuned code. By separating the different stages, this model helps the understandability and maintainability of the compiler. It also offers more flexibility, enabling the development of a different back-end for a different platform.

In a multi-stage compiler, extensions in the form of syntactic sugar are easily added: Such extensions can be rewritten to the base language in one of the early stages of the compilation chain. Features that are more than mere sugar, and interact with the back-end, require modification in multiple stages of the compiler: from parsing, to analysis, right to the point where the resulting class is emitted. Third-party developers need to make a large investment to understand and be able to modify the internal workings of the different stages and compiler components, even if it only concerns a relatively small extension. Especially for compilers implemented in a general-purpose language, such as Java, rather than a specialized transformation language, this can be a tedious task [51].

5.5.4 Implementation

In order to generate specific new instructions or class file attributes, we can benefit from using Java/bytecode as an intermediate language. Rather than requiring full integration of new language

features by extending the various stages of an existing compiler, it allows such extended instructions to be directly emitted in the initial stages of the compilation. This way, the language extension can be encapsulated into a separate compilation stage, where emitting specific bytecode instructions is as trivial as desugaring code to the source language.

Of course, to support *new* bytecode instructions, we need to extend the Java/bytecode language and ensure that the compiler back-end can handle these. As our implementation is based on a high-level program transformation environment, rather than a general-purpose language, it is sufficiently flexible to allow quick adaptation of the instruction set. Extending the instruction set in the compiler requires changes in two places: In the syntax definition, and in the integrated bytecode verifier (see Section 4.1.2). The syntax definition enables instructions to be read by the parser, and is defined using the modular syntax definition formalism SDF2 [67]. The verifier definition ensures that the code is understood by the semantic analysis of the source code, and that it can be verified when the program is generated to pure bytecode (see Section 7.5). For most instructions, the required definitions are very small and require only a few lines of code (see **Figure 5.21**). For other, more complicated instructions, this may of course be longer. One example is the `invokedynamic` instruction, which takes a specified number of arguments from the stack, rather than a static number like the addition operator does. In the current compiler implementation, the regular invocation instruction also required the most implementation effort due to this property. The syntax definition for this instruction is as follows:

```
"invokedynamic" MethodRef → Instruction {cons("INVOKEDYNAMIC")}
```

In this definition, we can conveniently reuse the `MethodRef` argument syntax as used for the regular invocation instructions. Of course, the real effort in implementing this instruction lies in the verifier definition. For this, we need to define a rule that returns the resulting stack, given a tuple of the input stack and the invocation instruction (see **Figure 5.22**). Specific for `invokedynamic`, we need not be concerned with the exact types of the values on the stack, but we do need to confirm that the right *amount* of arguments is available. Using the list operations in the `where` clause of the rule, this is in fact handled implicitly – if the number of required arguments cannot be dropped from the stack, these operations fail. As a result, the rewrite rule will then fail to apply (for more about the notion of failure in rewrite rules, see Section 2.2.1). However, rather than handling failure implicitly, we instead want to report an error message in case this happens. We can do this by extending the rule to call the `bc-error` function in case it fails (see **Figure 5.23**). This function reports the error, and uses the implicitly provided location information (see Section 4.2) to report the offending source code and location that it originated from.

In the changes we have made to support the `invokedynamic` instruction, we have taken advantage of the fact that the integrated verifier is written in a specialized, high-level language. For actually emitting the instruction from a source language construct, any external solution may be used as desired: An external preprocessor may act as a front-end for the language extension. To complete the compilation of the class file, the compiler also needs to write the instructions to the binary class form. This requires additional changes to the back-end, which is currently less accommodating for this purpose. We currently use the Dryad toolset for this (see Section 7.1), which applies the BCEL library (implemented in Java) to emit instructions. To adapt this library to output the instructions is a somewhat more involved process, and

`"ladd" → Instruction {cons("LADD")}`	`LADD;` `op(\[Long,Long] → Long\ |2)`

Figure 5.21 Required syntax definition (left) and verifier definition (right) for the basic `ladd` instruction (addition of 2 longs on the stack).

```
bc-invoke-effect:
  (stack, ⟦ invokedynamic class.method (params : ret) ⟧ ) → stack2
  where args := <length> params;       // count the number of parameters
        stack1 := <drop(|args)> stack; // drop the arguments from the stack
        stack2 := [ Any | stack1 ]     // place return type 'Any' on stack
```

Figure 5.22 The `bc-invoke-effect` rule is evaluated to determine the stack effect of an invocation instruction. Here, it is defined to evaluate the effect of the `invokedynamic` instruction. Using Stratego list manipulation operators and the standard `drop` function[12], the stack is manipulated to remove all input arguments, and to return a value of type 'Any'.

```
      <+ bc-error(|"Oops! Insufficient arguments.",
                  ⟦ invokedynamic class.method (params : ret) ⟧)
```

Figure 5.23 Produce an error message in case there are insufficient arguments on the stack, by extending the `where` clause of the verifier rule in **Figure 5.22**. The choice operator (<+; see Section 2.2.1) ensures that the `bc-error` function[12] is called in case the rule fails.

is out of the context of this thesis. It could be considered future work to alleviate this limitation, as the only further information really required for the back-end is the binary byte representation of the instruction.

In conclusion, we have shown that the Java/bytecode language offers direct access to the bytecode back-end, while this is not the case with other approaches. We have also shown how the compiler may be extended to support new instructions. As such, using a preprocessor compilation model becomes a viable approach to rapidly implement a compiler for a specific instruction set or JVM implementation.

[12] A number of Stratego functions require the use of the pipe symbol (|) for their invocation [55], indicating the type of argument that is passed to it. Here, rather than using pseudocode, we will follow this directive to accurately illustrate the required modifications to the compiler.

Chapter 6

Application in Composition and Separate Compilation

The Java/bytecode language can be used to modify compiled class files and combine them with other classes, or smaller units of code. Inserting code into a method or class can be performed by simply placing a fragment of code – Java or bytecode – into the source or class. This opens the doors for implementing software composition systems that apply separate compilation, without the need for integrating them into an existing compiler. This makes the Java/bytecode compiler a vehicle for implementing various forms of composition. This section lists a number of these, and shows how the mixed language compiler can help implement them.

Central to the techniques described in this chapter is the notion of **separate compilation**. Units of code (usually classes) can be individually compiled and distributed in a modular, binary form. Separate compilation allows the reuse of such units in new applications; for example, the Java standard library is distributed in binary (class file) form. Compiling such modules, rather than distributing them in source form, has various advantages. For one, it eliminates the need reveal the source code. This may be of benefit for developers that put license restrictions on the source code, and may not want others to view it. Compiled code also ensures verification: As the code has already undergone complete compilation, any compile-time errors are prevented. Another advantage can be in terms of performance: Compiling an application that makes use of a separately compiled library, does not require further compilation of that library. Separate compilation in Java is normally restricted to units of code in the form of classes. As we will see in the remainder of this chapter, it is also possible to separately compile smaller units of code. Using the Java/bytecode language, a separately compiled method or even expression is feasible, and may be combined with a source code program.

6.1 Partial and open classes

6.1.1 Motivation

Some programming languages (such as Ruby and C# [42]) allow classes to be split across multiple source files. Each *partial class* can define a non-overlapping set of methods and fields that will be joined into a single class (see **Figure 6.1**). This can be used for **separation of concerns**: splitting large classes into smaller units that can be edited separately, such as a GUI part and an event-handling part, or a public and a private interface. An advantage of partial classes over using inheritance for such applications is that methods in partial classes can also access **private methods** (see Section 2.1.2) defined in other parts of the class.

Partial classes are combined *before* compilation, but it is also possible to apply this technique to add members to classes that were already compiled. This variant is known as **open classes**, which is also considered one of the fundamental elements of aspect-oriented programming [31]. Open classes are still combined at compile-time (some implementations exist that do this at load-time), but this makes it

```
// In Calculator.java
partial class Calculator {
  public void add() { // the add button is pressed on the calculator
    setDisplay(...);
  }
}

// In Calculator_Gui.java
partial class Calculator {
  private void setDisplay(int number) {
    ... // modify the calculator's displayed value
  }
}

// In Calculator_Test.java
partial class Calculator {
  public void testDisplay() {
    ... // test if the setDisplay() method works correctly
  }
}
```

Figure 6.1 A *partial class* definition of a graphical calculator application. Note how the `partial` keyword is applied to all participating class definitions, to explicitly denote that these classes are to be joined. (Aiding in the readability of the application and preventing undesired merging of classes.)

possible to apply the technique to adapt already-compiled classes that may be included in libraries or applications.

6.1.2 Applications

Partial classes are sometimes used as a way to **modify generated code.** DSL code generators, such as visual GUI designers, often create very basic classes designed to be customized. By using a separate partial class the generated code can be modified indirectly. This allows the code generator to safely overwrite the generated code if the originating DSL source code is modified. This greatly simplifies the generator's job compared to parsing and editing the generated class, attempting to identify the old generated code and any user code. Since the generated code is in a separate file, it also becomes less tempting for programmers to reformat or "tweak" the generated code, possibly losing those changes when the generator runs again. Alternatively, it is possible to use *open* classes to allow modification of generated code. All generated classes can then be compiled, verified, and distributed in the language-independent class file form.

Unit tests are small test classes that confirm every part or member of an application's classes work as specified. Writing unit tests can directly help find existing bugs, and maintaining a collection of unit tests can help prevent regressive bugs from being introduced (i.e., it ensures that the developer is alerted if something stops working that worked and was tested before). One complication with separate testing classes is that they can only access public members of a class that is being tested. In many cases, it can be beneficial to also test any non-public members. Sometimes the use of non-public class members is even advocated against, simply because they are hard to test through conventional unit testing techniques. One solution for this problem is to create unit tests in a partial class, which *does* allow access to non-public class members (see **Figure 6.1**).

6.1.3 Challenges

Both partial and open classes are a relatively simple concept: Take all the class members defined for a class in multiple files, and merge them together to form a new, single-file class. This is exactly the approach commonly taken for partial classes: They are combined by a preprocessor, before the actual compilation starts. Open classes however, while conceptually similar, are merged during the compilation process. Rather than first merging and then compiling the resulting class, members added to a class are directly compiled into the target class. To reach this level of integration, this requires adaptation across all stages of the compiler: It must support such classes in the initial semantic analysis (i.e., looking up invoked class methods and type-checking), and all stages that make use of the information gathered in that stage.

An alternative approach to implement open classes by conventional means, is to extract signature information from all parts of a class, compiled or not. This can then be used to create dummy class members, to allow separate parts of a class to be compiled by a regular Java compiler. This allows the compiler to properly resolve any uses of members of the other parts of the class. Further manipulation of the resulting class files, given the proper registration of all dummy members, could then result in a complete, merged class. Of course, this is still more complicated and error-prone than the preprocessor approach that can be taken for partial classes.

6.1.4 Implementation

The Java class structure – methods, fields, types, flags, etc. – can generally can be mapped one-to-one to the bytecode class file structure. In the mixed Java/bytecode language we use this property to allow classes to include members defined in either base language (see Section 3.5). This of course helps in the definition of open classes. By thinking of the input classes and class files for this transformation as instances of this single, combined grammar, there is no difference in the merge process required for partial classes. It is sufficient to take all the class members, compiled or not, and combine them into a single class (see **Figure 6.2**). After this preprocessing phase completes, the resulting class can then be separately compiled by the Java/bytecode compiler. By applying the source tracing technique (see Section 4.2), we still support compile-time errors and debugging information normally not available with a preprocessor approach.

Unlike other class members, field initializers and constructors (see Section 2.1.2) cannot be mapped one-to-one to the compiled class file structure, as they are joined together at compile-time (see **Figure 6.3**). However, we implement support for these constructs using the same "compose and conquer" approach as we do for open classes: Any Java-defined field initializers and instance initializers (a rarely used Java construct, similar to constructors [2]) are simply separately compiled and inserted into the constructor.

```
class Calculator {
 // From Calculator.java
  public void add() { ... }

  // From Calculator_Gui.class
  private setDisplay(int number : void) [ ... ]

  // From Calculator_Test.java
  public void testDisplay() { ... }
}
```

Figure 6.2 Merged class from two Java files and one class file. The differences in syntax give away that `setDisplay()` is in fact a compiled method.

```
class Foo {
  public Foo() { // constructor
      i = 3;
  }

  int i = 1; // field initialization

  { // instance initializer (rarely used Java construct)
    if (i == 1) i = 2;
  }
}
```

Figure 6.3 A constructor and a field and instance initializer. They compile to a single constructor method, which will include all initializers in order of appearance, and then the original constructor (i.e., it assigns 1, 2, and 3 to `i`).

The constructor, if required, then undergoes further compilation. This means that given initializer definitions in multiple parts of a class, they can be combined with any existing constructor that exists in compiled parts of the class. In other words, they can be mapped to the merged class without any complications.

In conclusion, we can support partial and open classes in exactly the same way through preprocessing and separate compilation. The result is a class that can be compiled and verified using the mixed language compiler. Any duplicate or illegally referenced members can be detected by the compiler's verifier, after composition. Given this single compilation model, one might do well to ascertain which applications of partial classes could also be implemented using open classes. For example, the generated GUI code in **Figure 6.1**, which is commonly combined using partial classes, could also benefit from a open classes approach. It could be separately compiled to a class file, which could result in greater reliability (as the GUI part is already compiled and verified) and a higher compilation performance.

6.2 Traits

6.2.1 Motivation

Like many other object-oriented programming languages, Java employs inheritance to create reusable code. Java does not support **multiple implementation inheritance**, where classes can inherit from multiple base classes. This decision was because of the increased complexity and ambiguity, as well as versioning problems it can cause (often summarized as "the diamond problem") [6]. Instead, Java provides interfaces that cannot provide an implementation, but can only declare available method signatures (see Section 2.1.1). This approach eliminates most of the problems that exist with multiple

```
public class Shape with TDrawing {
  ...

  Vertices getVertices() { ... }
}

public trait TDrawing {
  void draw() { ... }

  Vertices getVertices(); // required method
}
```

Figure 6.4 An example of a class (*Shape*) importing a trait (*TDrawing*). The trait provides a `draw()` method and in turn requires a `getVertices()` definition. The `with` operator allows multiple traits to be combined in this fashion. Other operators manipulate traits, such as the `minus` operator that subtracts the methods defined in a trait.

```
public trait Arithmetic with Math minus ExtendedMath {
  // import all methods from the 'Math' trait,
  // but exclude members defined in 'ExtendedMath'
}
```

Figure 6.5 Using operators to manipulate traits.

inheritance, but also somewhat restricts the way code can be reused. Other approaches to facilitate **code reuse** have been developed as alternatives, but have not (yet) made it into the Java language.

Traits are primitive units of reusable code that define a set of methods for use in regular classes (see **Figure 6.4**). Like partial or open classes, they do not apply inheritance for composition, but are simply integrated into normal classes. Traits however do not declare which classes they are used in. Instead, classes can specify what traits they require (i.e., they use a "pull model"). This makes traits reusable for any number of classes, and in principle allows the inclusion of traits in libraries. Using special operators, traits can also be composed into new traits. For example, in addition to the `with` operator to add methods (**Figure 6.4**), the `minus` operator subtracts methods defined in one trait from another trait (see **Figure 6.5**). Some trait implementations also include a `rename` operator to rename methods in a trait, which can help in case multiple traits define the same method signature. This allows traits to explicitly handle the diamond inheritance problem.

6.2.2 Existing implementations

Schärli *et al.* [6] originally designed and implemented traits for a SmallTalk dialect. Since then, the concept has been ported to many languages, including Java, C# and Scala [24]. Since SmallTalk is a dynamic language, the original implementation used **dynamic typing** of traits. A trait could call any method in the client class, and in case of a nonexistent method, this resulted in a *message-not-understood* runtime error. More recently traits were studied in statically typed languages, where such errors can be detected and prevented at compile-time [23,24,26,37]. Several approaches to **static typing** have been suggested, such as using type-checking client classes after inclusion only [24,26], explicitly specifying method requirements to also type-check traits [23], or even inferring these requirements [37].

Unlike inheritance and mixins, traits are **stateless** (i.e., they cannot define fields), and do not define type information such as an interface on client classes (since they are composed using operators other than inheritance). There are numerous (prototypical) implementations and interpretations of "traits" for Java, and some go beyond this definition. Smith and Drossopoulou [23] for example, allow traits to be substituted at runtime. However, in the present study, we will follow the basic definition, as given in Schärli *et al.* [6].

6.2.3 Challenges

Being able to type-check traits as separate entities makes it possible to apply **separate compilation**. Although we have no intention for the JVM to load these classes – traits are merely **compile-time entities** – it can be useful to compile traits to actual class files. Libraries of regular Java classes are usually distributed in compiled form, supported by both compile-time verification and dynamic linking in the JVM. Compiling traits to class files allows trait definitions to be included with related library distributions. Although not dynamically linked, traits distributed this way can be used by the compiler for reuse and inclusion in third-party classes. Compiled traits actually allow this to be done in a language-independent fashion (assuming the compiler for a given language supports importing trait definitions).

Since the regular Java language does not support traits, there is also no support for traits in the class file format. One way to represent traits as class files is as a form of **abstract classes:** All *required* methods can be represented as abstract methods. This representation can remain fully compatible with the standard class file format, and allows verification of the trait class with any regular class checker. To indicate that such a class file is a trait, and not a mere abstract class, the class file format allows it to be marked using a custom attribute or **annotation** to indicate that the file in fact a trait. Additionally, since trait classes only have a compile-time purpose, it may be desirable to prevent other classes from using or extending the trait by marking it `private` and `final`.

6.2.4 Implementation

As illustrated in the previous sections, our compiler and language can help with the composition of binary and source classes. For trait classes the principle is the same, which means there is little to no extra implementation effort compared to combining traits at the source-level. All members of a class and any imported traits can simply be combined into one big class file. The compilation process can then continue to compile all source code to bytecode, and can finally confirm if the newly formed class is legal (and has no missing or duplicate methods).

We built a Traits compiler for Java using the Java/bytecode compiler as a foundation. Totaling 121 lines of code[13], it is much smaller than a typical traits compiler, or most language extension preprocessors for that matter. This is because it does not include any logic for compiling Java classes, it is purely restricted to the composition of traits and classes. This is a three-stage process:

- The **analysis** stage, where all traits are read from their respective sources or class files (either format is supported, the traits compiler only cares about the class and method signatures);
- the **evaluation** stage, where all trait imports (`with` and `minus`) are evaluated;
- and finally the **rewriting** stage that uses the gathered information to add or subtract methods from classes and processed traits.

After composition, we rely on the Java/bytecode compiler for the actual compilation of the composed classes; any compilation errors related to the general language are reported at that point. Any trait-specific errors are detected in the trait compiler (i.e., missing trait declarations, missing 'required methods', or circular trait imports). The remainder of the compilation process, including error reporting (see Section 4.2), is handled by the general Java/bytecode compiler.

We implemented the traits compiler in the Stratego language [55]. Since we used some advanced language constructs that are well outside of the context of this thesis, we present the traits compiler here in pseudocode form (see **Figure 6.6**).

6.2.5 Future work

A possible future extension of the compilation of traits would be to merge the trait it at the time the class is loaded, rather than at compile-time. Regular Java classes are dynamically linked after compile-time (see Section 2.1.11). Dynamic linking, in general, has obvious advantages: It allows linking applications with different or updated versions of a library. However, dynamic linking is far from a silver bullet for

[13]The complete program also includes a small syntax extension definition for the traits language. Excluding this, I/O code, imports, blank lines, and comments, it totals 57 lines of code.

```
// MAIN

assimilate-java-traits = // invoke all stages
  for all traits:
    try classfile-to-trait; // (if applicable, convert compiled trait)
    register-trait
  for all inputs:
    try complete-trait or complete-class

// ANALYSIS

// If a trait was already compiled, restore its signature
classfile-to-trait:
  ⟦ abstract class x { ~*methods } ⟧ →
  ⟦          trait y { ~*methods } ⟧
  where y := restore uncompiled name of trait x

register-trait:
  ⟦ trait x ~*imports { ~*methods } ⟧ →
  store all information from the inputted signature

GetTraitMethods:
  // (uses the register-trait information; in regular Stratego, this would be
  //  a runtime-defined or "dynamic" rule)
  x → all-methods
  where if already being evaluated:
          all-methods = error("circular reference", x)
        otherwise:
          // Recursively include methods of imported traits
          all-methods = <evaluate-trait-exprs> imports

// EVALUATION OF TRAIT IMPORTS

evaluate-trait-exprs(classname, methods) =
  in case of a "with" import:
    GetTraitMethods;
    specialize: replace all class name references with current class name;
    union of all current methods and imported methods;
    evaluate remaining trait imports

  in case of a "minus" import:
    get-specialized-trait-methods(classname);
    substract all imported methods from the current methods;
    evaluate remaining trait imports

// REWRITE TRAITS AND CLASSES

complete-trait:
  ⟦          trait x ~*traits { ~*_ } ⟧ ->
  ⟦ abstract class y          { ~*methods } ⟧
  where y       := give reserved name
        methods := GetTraitMethods;
        remove all implemented "required" methods

complete-class:
  ⟦ class x ~*traits { ~*methods     } ⟧ ->
  ⟦ class x          { ~*all-methods } ⟧
  where all-methods := <evaluate-trait-exprs> traits;
        for all "required" methods that are not implemented:
          error("Missing required method", method)
```

Figure 6.6 A pseudocode approximation of the *traits* compiler. Note how it uses class and method signatures for composition, but includes no other logic about the structure or semantics of the language. Not included here are API invocations to process the command-line arguments, load all inputs, and disassemble method signatures.

dependency management, but it can be helpful if the modified library provides sufficient compatibility with the expected version. When applied to the concept of traits, its advantages may not be so obvious. Traits do not define a type or provide an interface in the way normal classes do: They only affect an individual class. If the class is updated, it will be recompiled and linked again with the trait. In case the trait is updated, only the members that already existed at the time the class was compiled may be accessed. Finally, 'dynamic linking' for traits may be costly in terms of performance, as rewriting a class to include traits is more computationally expensive than regular dynamic linking. Still, given the proper runtime class loader modifications, it may be interesting to investigate the possibilities of dynamically linked traits.

In conclusion, we have seen traits, a powerful mechanism for safely enabling code reuse. The "pull model" of traits makes it an excellent candidate for separate compilation. As such, it enables the distribution of traits in compiled form, within libraries. We are unaware of existing implementations that provide this, as they combine traits at the source level. By taking advantage of the mixed language, it requires little further effort to use the class file format, and thereby make use of separately compiled traits.

6.3 Integration of crosscutting concerns

6.3.1 Motivation

Crosscutting concerns are aspects of a program that cannot be cleanly decomposed from the rest of an application and its design, and are often scattered throughout it and tangled with other concerns. Regular, object-oriented programming techniques can help in the reuse of code by providing classes that can be extended and invoked where needed. For crosscutting concerns however, this at best results in invocations of the class scattered and tangled throughout a program. A cleaner approach can be to capture such concerns, and use automatic rewriting of the program to include them.

6.3.2 Applications

The textbook example of a crosscutting concern is a logging facility. Logging may be used to register all database transactions for example. This means that anywhere where a database operation is performed, the code must be adapted to invoke the logging module. One might argue that an object-oriented solution exists for this specific problem, perhaps in the form of a decorator class (i.e., an indirection between de application and the database interface). But what if the logging extends to other areas? To aid debugging for example, it may be desirable to log all method invocations in the entire application. This requires integration in all or most methods of a program; in such cases a more general solution is desired.

Consider a more concrete example, in the form of a simple banking application where funds are transferred from one account to the next (**Figure 6.7**). In a real-world banking application, it is not

```
void transfer(Account checking, Account savings, int amount) {
  if (from.getBalance() < amount) throw new InsufficientFundsException();

  checking.withdraw(amount);
  savings.deposit(amount);
}
```

Figure 6.7 A simple method for transferring funds from one bank account to the next.

```
void transfer(Account checking, Account savings, int amount) {
  if (!getUser().allowWithdraw(checking)) throw new IllegalAccessException();
  if (from.getBalance() < amount) throw new InsufficientFundsException();

  Transaction transaction = TransactionManager.start();
  try {
    checking.withdraw(amount);
    savings.deposit(amount);
    logOperation("transfer", checking, savings, ammount);
    transaction.commit();
  } catch (RuntimeException exc) {
    transaction.rollback();        // undo transaction if something goes wrong
    throw exc;
  }
}
```

Figure 6.8 A more complete funds transfer method, now including various (simplified versions of) crosscutting concerns.

sufficient to simply withdraw the funds from one account and then deposit them in the next. For instance, such an operation requires security checks to verify that the user is authorized to perform the transaction. It also should perform the operation inside a database transaction, preventing data loss or conflicts with other transactions. For diagnostics, it should also include logging. In short, in such a method the basic functionality (sometimes called business logic) would become tangled with other aspects. This results in the method losing its elegance (see **Figure 6.8**).

Defining crosscutting concerns as a separate concept, added to an application at compile-time, can aid in the basic understandability of the application. (Although not everyone will be comfortable with implicitly adding security concerns to an application; such separation should be used responsibly.) It can also be used as a way to decide at compile-time or deployment-time whether or not to include a certain feature. Some performance profilers and tracing tools, for example, require the insertion of deeply integrated, performance-intensive code. By programmatically adding such code, it can be inserted as desired, or left out in production builds.

6.3.3 A systematic approach: Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) attempts to aid programmers in **separation of concerns.** The paradigm is not intended as a full-out replacement for existing separation of concerns mechanisms, such as inheritance and aggregation in object-oriented programming. Instead the focus is to provide a complementary mechanism to support systematic composition of concerns that cut across the existing base separation and would otherwise be scattered across various modules [40]. While such concerns have been dealt with for a long time, Kiczales *et al.* [31] introduced the first set of techniques to modularize and manipulate these in a *systematic* fashion [40]. Kiczales *et al.* also developed the most popular AOP language, AspectJ [32], an extension of Java. Numerous other implementations have been introduced since then, including many for Java.

AOP allows the programmer to express crosscutting concerns explicitly in modules called **aspects.** Aspects can contain **inter-type declarations** and **advice.** Inter-type declarations are a form of open classes (see Section 6.1) and allow modifications of existing types. *Advice* allows behavioral changes of existing classes, by joining code into specified points of a program. The programmer can define **pointcuts** to indicate what methods or field accesses ('**join points**') should be affected by these changes. Together they provide a systematic approach for integrating code into a range of methods (see **Figure 6.9**).

```
aspect Logging {
  pointcut transfer(amount):
    call(void Bank.transfer(Account, Account, int amount))

  after: transfer(int amount) {
    System.out.println("Transferred: " + amount);
  }
}
```

Figure 6.9 Pseudocode for a logging aspect. It defines a pointcut for calls to the `Bank.transfer()` method. Based on this pointcut, advice is given to print a logging message, using the `after` clause to denote that this must happen *after* the transfer.

Aspect weaving is the process of integrating aspects with regular classes. While originally this was done by combining aspects and classes at the source level, AspectJ was later adapted to do this at the class file (bytecode) level. This meant that the aspect weaver no longer requires access to the source code of targeted classes; they may be separately compiled and distributed in binary form.

6.3.4 Challenges

Developing AspectJ as a bytecode-level aspect weaver required significant engineering effort. To allow bytecode-level operations, it was integrated in an open compiler (the Eclipse Compiler for Java [56]). While this is a mature, open compiler that provided a solid basis for the Java language, integration of aspects proved to be a daunting task, truly crosscutting into various levels and places of the compiler. This led to a problem when the Java 5 language was first introduced, which included various new language features, such as generics. These features were not supported in the forked aspect compiler. Eventually, with the help of the Eclipse developers, the AspectJ implementation integrated these features, and matured in terms of performance and correctness.

6.3.5 Implementation

In Section 6.1 we demonstrated how the Java/bytecode language can be used to insert complete class members into existing classes through the *open classes* concept, which is similar to inter-type declarations as they exist in AOP. In this section we focus on adapting method *bodies*. There are various approaches to 'weaving' code into methods, some implemented in stand-alone applications such as the aforementioned profiling and tracing tools. Here, however, we will focus here on the more systematic approach as provided by *aspects*.

Weaving aspects into source code Consider again the aspect defined in **Figure 6.9**, which adds advice to an invocation of the `transfer()` method. Weaving this aspect into a regular Java source class requires finding where the `transfer()` method with the exact given type signature is called. This cannot be directly extracted from the Java syntax (see **Figure 6.10**), and requires a semantic analysis of the application. For this, a tool such as Polyglot [34], Dryad [7], or the Eclipse front-end [51] may be used, allowing implementation of the aspect weaver separate from the compiler.

```
void caller() {
  ...
  bObject.transfer(checking, savings, inputAmount());
}
```

Figure 6.10 A fragment of a program that calls `Bank.transfer()`. To determine if the method called is in fact the method we are looking for, a type checker must identify the type of `bObject` and the method arguments.

```
void caller() {
  ...
  bObject.transfer(checking, savings, [ stm:  int amount = inputAmount();
                                        expr: amount ]);

  // Inserted aspect
  System.out.println("Transferred: " + amount);
}
```

Figure 6.11 Weaving in the logging aspect. The `amount` argument, used in the aspect, is assigned to a variable in-place.

After all join points have been located, it is a simple matter of inserting the aspect code according to the given strategy. For the logging advice, it will be woven in *after* the method invocation. Since this particular aspect makes use of the `amount` argument, a new temporary variable must be allocated to store this argument. For this we make use of the bytecode expression pattern (see Section 5.1), allowing us to directly replace the argument in question with an expression that stores the value in a local variable. Alternatively, one of the more advanced patterns for handling temporary variables could also be applied here (see Section 5.2). **Figure 6.11** shows the method with the advice woven in. The classic source-level weaving version of AspectJ was implemented according to a similar principle (although it must place local variable declarations in a separate statement).

Weaving aspects into compiled code We can also weave aspects into compiled classes, which is a very similar process to weaving to a source class. Using the property of stack-neutrality of statements (see Section 3.5.1), any statement or block can be safely inserted into bytecode. This means an aspect's advice can be directly inserted into a bytecode method body, without requiring beforehand compilation. The process of locating join points, however, is different for compiled code: The code has already undergone semantic analysis at compilation time and is explicitly typed, allowing it to be used *as is* to find the matching join points (see **Figure 6.12**). At any join points found, the advice can be inserted in its Java form. Our logging advice however makes use of the `amount` argument of the method call, which means we first have to assign a local variable to the value of this argument. The bytecode method invocation loads all its arguments (and the invoked object) of the stack. Therefore we must capture any arguments we need *before* the invocation. We use a series of `store` and `load` instructions to consume and restore the stack arguments using local variables (see **Figure 6.13**). The argument variables can then be directly used from the aspect inserted in Java form.

Aspects in compiled form Until now we have discussed weaving aspects in source form. Another feasible approach is to use a compiled form of aspects, allowing aspects to be separately compiled to class files. This allows aspects to be compiled and verified before they are woven into the code, which could result in increased reliability and ('weave-time') performance.

The binary representation of aspects, even more so than for traits (Section 6.2), requires the inclusion of meta data in such a class file. While regular bytecode can be used to represent the inserted code of an aspect, this metadata is used to describe the pointcuts and advice strategy information. In source code this information is included in the form of the aspect language (see **Figure 6.9**). In a binary class this

```
caller(void) [
  ...load Bank instance and Amount arguments on the stack...
  ldc 100;
  invokevirtual Bank.transfer(Account, Account, int : void);
]
```

Figure 6.12 Bytecode representation of a method call to `Bank.transfer()`. The invocation is explicitly typed, allowing easy detection of the join point.

```
caller(void) [
  ...load Bank instance and Amount arguments on the stack...
  ldc 100;

  // Store any used arguments in locals, then push them back on the stack
  store amount;
  load amount;

  invokevirtual Bank.transfer(Account, Account, int : void);

  stm: { // inserted aspect
    System.out.println("Transferred: " + amount)
  }
]
```

Figure 6.13 The bytecode method call, with source code advice woven in.

metadata must be in another form. Existing AOP implementations that support binary traits often use annotations for this, a standard Java mechanism for placing meta data in source code or class files [2]. Such approaches essentially form a new aspect language on the basis of annotations. This can certainly be attractive since no extra syntax is needed. It also allows the same annotations to be used in source and in compiled form. However, the binary class files produced through this method contain extensive metadata, and can still only be used by using the appropriate aspect weaver (which is often applied in the runtime for annotation-based implementations). The class files are otherwise meaningless for the JVM, which defines no semantics for the annotations. Based on this observation, we may also pursue another approach, where we do not pose the constraint of compliance to the class file format. Using the standard AOP language and inline bytecode, we can maintain the standard source-level aspect (metadata) language (see **Figure 6.14**). We compile only the inserted fragments of code (relying on support for compilation of code fragments; see Section 7.2.1). This results in a **pre-compiled** aspect, which does not requires further compilation of the inserted aspect code. Rather than loading such a file through a class loader, it can then be loaded using a parser for the mixed aspect language[14]. To apply the advice to target classes is at that point the same as applying regular source advice. **Figure 6.15** shows an example of bytecode-based advice inserted in a Java source code method.

Summarizing, there are different approaches one can take to insert crosscutting concerns into existing code. Our compiler can help integrate crosscutting concerns, by providing a representation that can be altered in a uniform way, regardless if it represents source code or is already compiled to bytecode. It

```
aspect Logging {
  pointcut transfer(amount):
    call(void Bank.transfer(Account, Account, int amount))

  after: transfer(amount) {
    [ ldc "Transferred: ";
      load amount;
      ...
    ]
  }
}
```

Figure 6.14 A pre-compiled aspect, using inline bytecode.

[14]Out of space and performance considerations, it is also possible to store the aspect in abstract syntax tree form using the standard binary ATerm [55] format. The resulting overhead from loading the file is then comparable to loading a binary Java class.

```
void caller() {
  ...
  bObject.transfer(c1, c2, [ stm: int amount = transferred; expr: amount ]);

  // Inserted aspect
  [ ldc "Transferred: "; ... ]
}
```

Figure 6.15 Weaving the pre-compiled logging aspect into Java source code.

provides facilities for debugging information, which supports error messages, and allows any inserted code to be debugged using the originating source file. Finally, the verifier can be used as a safety net to help detect any illegal modifications. These features together allow for a very slim implementation of aspect weavers that do not need to be integrated into a compiler.

Finally, the aspect-oriented features we have discussed here are some of the most used, but also most basic features of aspect weavers. Different aspect weavers have other join point models, and may allow for more elaborate specification of point cuts, including various conditions other than just a method signature. Existing implementations have shown that these can be woven into bytecode through various analysis techniques. Our contribution is to simplify the actual integration of advice, and to simplify the representation of compiled aspects.

Chapter 7

Design and Implementation

7.1 Tools

For the implementation of the compiler, we made use of the **Stratego/XT** program transformation system, a generic infrastructure for program transformation [3,4,55]. It provides a modular parsing and transformation solution that has proven invaluable in the implementation of the compiler, as well as for some of the case studies. Stratego/XT includes the modular syntax definition formalism **SDF2** [67] that allows one to extend an existing grammar. We made use of the **Java-front package**[15] that provides a syntax definition for the complete Java language, which we extended for our application (as seen in **Figure 3.2** and **3.3** on page 30). We also made use of the derived **Dryad toolset**[16], a collection of tools for developing Java transformation systems, providing semantic analysis of Java code and interfaces to Java class files. Dryad closely follows the Java Language Specification to provide type checking as well as qualification and disambiguation of identifiers. As such it forms the basis of our Java/bytecode compiler. Because of this substantial basis for dealing with the Java language, we name our implementation **the Dryad Compiler**.

Like Dryad, our implementation uses the Stratego programming language, a program transformation language based on rewriting using rewrite rules (see Section 2.2). It allows the specification of strategies (functions) for controlling how these rules are applied. Stratego has an extensive library of strategies for term traversals and transformations, and uses the **ATerm format** [55] to allow processing of abstract syntax trees as produced by the SDF parser. Using Dryad, this format for terms can also be used to read from and write to binary Java class files. We designed our bytecode syntax definition to closely follow the tree grammar of the ATerm representation as defined by Dryad (albeit with the introduction of certain abstractions, as seen in Section 3.2). This means that in our implementation we are only concerned with rewriting terms, from one grammar to the next. With the Stratego feature of using inline concrete syntax in the source code [4,55], this results in a highly readable and maintainable program.

In the remainder of this chapter we will give a high-level overview of our compiler implementation and its design. We include some small code samples that introduce some of the more advanced features of Stratego to illustrate selected aspects of this implementation. For background information on the Stratego language, we refer the reader to the Stratego documentation [55].

[15] http://www.program-transformation.org/Stratego/JavaFront/.

[16] http://www.program-transformation.org/Stratego/TheDryad/.

7.2 Compiler architecture

The compilation process is divided into three stages. The inputted Java/bytecode is first processed by the **front-end** stage, which performs parsing and semantic analysis. The **code generation** stage then rewrites the Java constructs to bytecode instructions (using the reduced instruction set described in Section 3.2). Finally the **assembly** stage specializes all pseudo-instructions, and transforms the generated code to a regular Java class file. **Figure 7.1** shows a diagram of these stages, and includes some of the more important components that will be introduced in the remainder of this section.

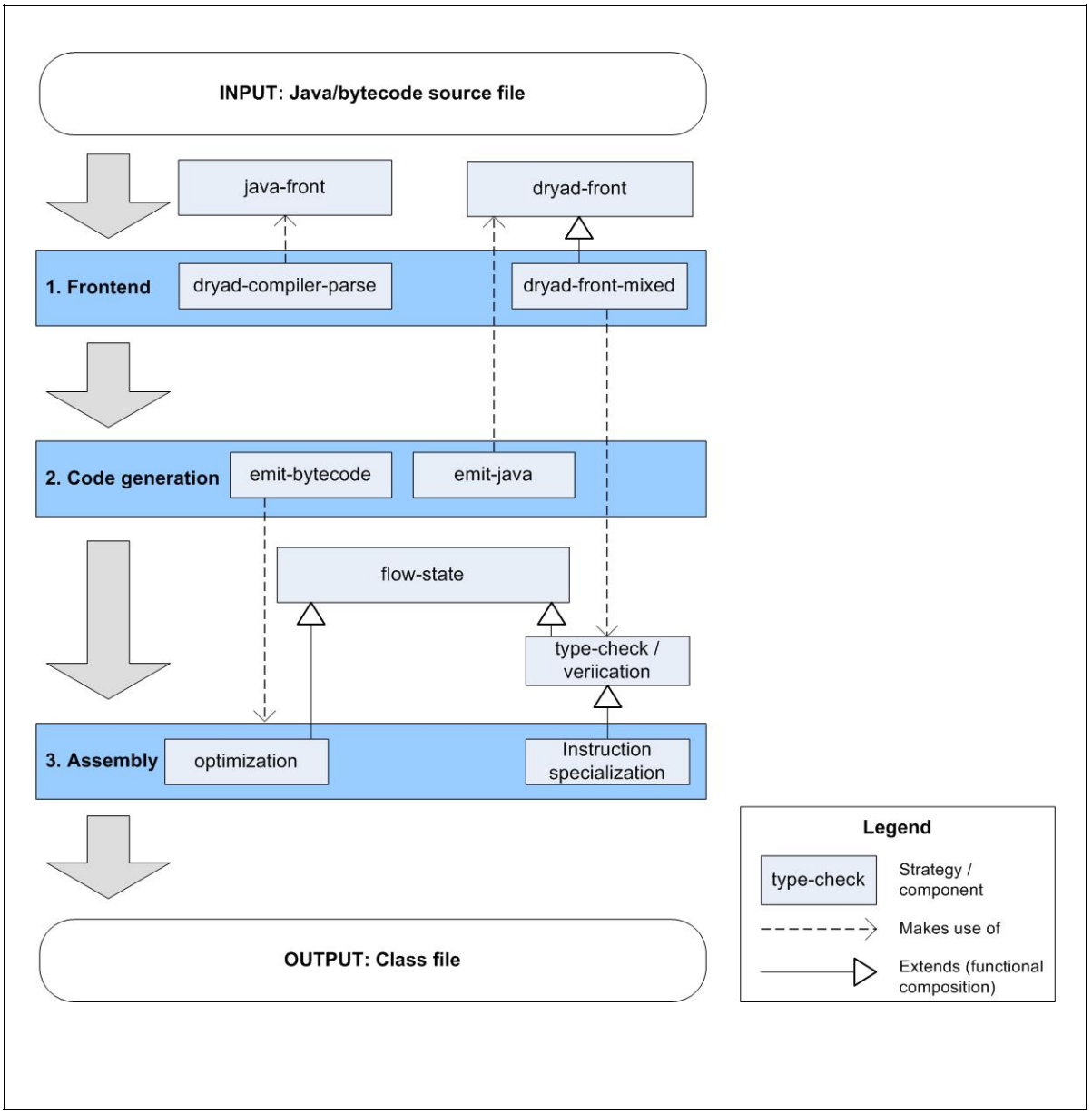

Figure 7.1 A diagram of the compiler architecture. For each of the three stages, key strategies and their relations are shown.

7.2.1 The front-end

At the beginning of the compilation chain, the input is parsed using the Java/bytecode parse table. This input can be any number of source files, or optionally a code snippet such as a method, a Java expression, or list of bytecode instructions. Parsing a snippet allows quick compilation of a fragment of code, which can be useful for debugging or unit testing. We also applied this technique to pre-compile fragments of code used in *aspects* (see Section 6.3).

The **partial parsing** implementation depends on the generalized-LR parser used to parse the SDF2 grammar [67], which allows parsing of ambiguous inputs. Java code fragments can sometimes be ambiguous; for example, an input '`int i;`' could be parsed as either a field declaration, or as a local variable declaration. The parser deals with such ambiguities by producing a parse forest – a set of all possible parse trees for the input. Rather than requiring the user to specify the correct sort (i.e., interpretation) of an input, we disambiguate automatically based on a set of rules specific for the Java/bytecode language. For our application – compiling small fragments of code – we have selected rules that simply work by means of a preference order on the set of possible interpretations. The strongest preference is for regular classes (which could alternatively be interpreted as local classes in method bodies), followed by methods, statements, and finally expressions. For our example input ('`int i;`'), this means it is interpreted as a declaration statement.

The parsing and disambiguation process occurs in the `dryad-compiler-parse` strategy (see **Figure 7.1**). It can be reused as a basis for parsing Java language extensions, by providing a custom parse table. After the input is parsed, the resulting abstract syntax tree is processed by Dryad. Dryad loads all imported classes and performs type-checking. It also uses this information to disambiguate Java identifiers. For example, a statement like `System.out.println()` is highly ambiguous: `System` could be a package, a class, or even a local variable; likewise `out` could be an inner class or a field. Naming conventions may help the human reader, but cannot be depended upon in a compiler. Dryad determines the exact meaning of identifiers based on their context, taking into account local declarations and imports. Dryad also qualifies all unqualified identifiers (e.g., resulting in `java.lang.System.out.println()`), and makes all this information available through annotations on the abstract syntax tree and through an API interface.

The `dryad-front-mixed` strategy extends the regular Dryad front-end, by providing an interface over the bytecode class structure and by extending the Java type checker with the bytecode type checker/verifier that is also used in the assembly stage (see Section 7.2.3).This ensures that all type errors are caught during the front-end stage, and provides essential typing information for the code generation stage. This information is for example used by Dryad for method overload resolution.

7.2.2 Code generation

In the code generation stage, the source abstract syntax tree is rewritten to class file form. At the heart of this translation lies a series of rewrite rules. These rules rewrite Java constructs to bytecode instructions, thereby essentially specializing the language to its bytecode component. As these rules can make use of the reduced instruction set, any specialization of the emitted instructions can be delayed until the assembly stage, which greatly simplifies the used rewrite rules (see Section 3.4) This stage also processes any existing bytecode instructions, to ensure proper sharing and scoping of local variables. **Figure 7.2** shows an example of a basic rewrite rule for the Java `if` statement.

```
emit-if:
  ⟦ if (e) stm1 else stm2 ⟧ →
  ⟦ <emit-java> e;
    ifeq else;
      <emit-java> stm1;
      goto end;
    else:
      <emit-java> stm2;
    end:
  ⟧
  where (else, end) := <new-labels>
```

Figure 7.2 Rewriting the Java `if` statement to a series of bytecode instructions. The conditional expression *e* and the inner statements are recursively rewritten through application of the `<emit-java>` strategy. The `where` clause in this rule is used to define new `else` and `end` labels, used in the right-hand side of the rule.

For some more advanced Java statements and expressions, additional context information is required for compilation. This is provided through annotations set by the front-end, which provide type information and allow for reflection using the Dryad API.

7.2.3 Assembly

The final compilation step is the assembly stage. While we perform complete type-checking and verification in the front-end stage, the assembly stage performs a final verification of the generated bytecode. This allows the detection of errors in the code generation process, but also picks up Java errors that are not currently reported by Dryad. (Presently, Dryad only performs basic type-checking of expressions, and does not confirm that, for example, a conditional expression as seen in **Figure 7.2** is actually a boolean.)

During the assembly stage, the type-checker also provides type information that is used for instruction specialization: The rewriting of instructions from the (overloaded) reduced instruction set to regular bytecode instructions. This process is described in Section 7.5.

After instruction specialization, the bytecode is normalized further, removing the source location information and exception handler blocks (see Section 3.2). Every instruction is traced back to the original line of code it originated from. This information is stored in debug symbol tables (see Section 2.1.9). Similarly, all the instruction offsets of the exception handler blocks are placed in exception handler tables.

7.3 Source code tracing

During every stage of the compiler, location information about the original source code is maintained. This information is either provided by the parser, or extracted by the front-end using source tracing information that may be embedded in the syntax (see Section 4.2). Source tracing information maintains the original source construct that was used to generate the target construct, its location information, and the resulting generated code:

```
trace (<originating code> @ <filename>:<line>:<column>) [
  <generated code>
]
```

```
void foo() {
  String s = 1 as String;
}
```

```
Foo.Java:3:3: Input can never be of type:
              java.lang.String
  String s = 1 as String;
  generated: checkcast java.lang.String
```

Figure 7.3 On the left, an expression using the **as** operator (see Section 5.1) that causes an error message (right), which refers to the original code and the generated `checkcast` instruction.

The *originating code*, a fragment of source code from the original file, is optional in this clause. We include it, however, to display compile-time errors not only with the originating location, but also with the fragment of code that it originated from (see **Figure 7.3**).

7.3.1 Rewriting with source code tracing

The parser places default location information as annotations on the abstract syntax tree, which can be used if tracing information is not available. These annotations can also be used to implicitly generate source tracing information when a source construct is rewritten:

```
desugar-if:
  ⟦ if (e) stm ⟧ →
  ⟦ trace (stm_traced) [
      if (e) stm else ;
    ]
  ⟧
  where stm_traced := <id> // assign input of this rule (including annotations)
```

In this rewrite rule, the one-armed `if` statement is desugared to a regular, two-armed if (as seen in Section 2.2). The rule also introduces a `trace` clause that includes the original `if` statement, and its associated position information in implicit annotation form. As we apply this pattern of adding tracing information for practically rewrite rule in the compiler, we defined a strategy that will introduce the `trace` clause automatically:

```
desugar-if': ⟦ if (e) stm ⟧ → ⟦ if (e) stm else ; ⟧

desugar-if = trace(desugar-if') // apply tracing (to one or more rules)
```

The *trace* strategy operates on the basis of the context of the language construct that is rewritten, and will only add a `trace` clause if there is no tracing information yet available. Note that we make use of the ATerm format [55], which is based on the principle maximal sharing of terms, which means that the inclusion of the original construct has the same memory footprint as a pointer.

Although we built our implementation on this basis, in Stratego, it is equally possible to generate this information in an implementation based on a different platform. The resulting, generated code contains the information in the syntax, and can be produced independently of the parser and library strategies we provide for it. This way, a language extension implementation can still make use of the source tracing facilities, while maintaining minimal coupling with the back-end compiler implementation.

7.4 Evaluation and unit testing

For testing specific features of a language or language extension, unit tests can be used to verify that the correct code is generated. Writing such tests can directly help find bugs during the development of the

```
operator-tests =
  test-eval(|[ "a"  as String  ]|, "a");
  test-eval(|[ "a"  as Integer ]|, "null");
  test-eval(|[ "a"  ?? "b"     ]|, "a");
  test-eval(|[ "a"  ?? null    ]|, "a");
  test-eval(|[ null ?? "b"     ]|, "b")
```

Figure 7.4 Unit test definitions for the `as` operator (Section 5.1) and the `??` operator (Section 5.2). For each line, the expression on the left is evaluated using the Java Virtual Machine, while the expression on the right is the expected result.

compiler. Maintaining a collection of unit tests can help prevent regressive bugs from being introduced. However, specific for code generation, specifying the complete outputted code for a specific input can be quite tedious. Generated code is often much larger than the inputted code (see for example **Figure 5.4** on page 48; the bytecode generated for the simple `as` operator). This may tempt some programmers to blindly copy-paste the result from the generator itself, defeating some of the purpose of the tests. Additionally, it is not trivial to verify generated names for fresh variables and labels, even though this may influence the semantics of the generated program.

An alternative way for unit testing is to test the functionality rather than the code of a generated program. We support this approach by executing the generated program in the JVM, and comparing the result of the program to a given expected result. The partial program parsing and compilation technique (see Section 7.2.1) makes it possible to compile and directly evaluate small program fragments. We have successfully applied this technique for unit tests of the Dryad Compiler itself, as well as for the Java language extensions of some of the case studies (see **Figure 7.4**). Every input fragment in such a unit test generates Java/bytecode for the specific application. The resulting program or fragment is then further compiled by Dryad Compiler to a stand-alone class file. This program is then loaded and executed in an instance of the JVM, and returns a value in the form of a string. This string is then compared to the expected unit test value. If this test fails, or if any of the compilation steps reports errors or warnings, the unit test will fail.

7.5 Verifier and inferencer

7.5.1 Bytecode analysis and verification

The JVM specification includes an abstract specification of how a bytecode verifier (see Section 2.1.10) should operate [1]. It allows different approaches of type checking bytecode, and [15] contains an overview of the area. The most straightforward (but also most computationally intensive[17]) implementation and follows directly from the specification. The JVM specification describes the process as a **fix-point iteration**, where a method's instructions are iterated over a number of times until all possible execution paths have been analyzed. Because of restrictions on the form of a method its instructions, this can be a straightforward process. The **first restriction** is that for all instructions their effect on the stack can be statically determined. For a method invocation instruction, this means that it must specify the arguments it takes from the stack and what the return type is. This also means that the verification can be done in an **intraproducural** setting [18], i.e., it only requires analysis of a single method at a time. The **second restriction** is that the stack must be the same for all possible execution paths leading up to an instruction (which follow the branching instructions). This restricts the way

[17] In Java SE 6 class files were extended with a new attribute to declare types of local variable types. This can significantly increase the JIT verifier's runtime performance as it only needs to check types, not infer them. However, this only applies to classes generated by the new compiler, and it is not used in this project.

```
// given an instruction, and its input stack, determine the resulting stack
bc-stack-effect =
  ... // match the instruction, to determine the result:

  IFEQ;        op(\Boolean  → []\  |1) // takes 1 Boolean, returns nothing
+ ARRAYLENGTH; op(\Array(t) → Int\ |1) // takes 1 arraytype, returns int
+ RETURN;      op(\[]       → []\  |0) // takes 0 operands, returns nothing
```

Figure 7.5 Instructions and their effect on the stack are defined using the `op()` function. Each instruction is specified on a line of code with respectively the instruction, the inputs expected on the stack (e.g., a boolean), the resulting output (e.g., an empty list; []), and finally the number of inputs expected. This information is used to determine the stack effect of an instruction, and for displaying errors for invalid inputs.

programs that can be expressed in bytecode, but is essential for allowing accurate and fast verification by the JIT compiler. Using fix-point iteration all paths can be followed and iterated over, until finally all stack states can be determined to be complete and correct. This process can be aborted with an error message if two paths leading up to an instruction have inconsistent stack states, or if the stack types are not as expected.

While it is not intended to have such as broad coverage as the BCEL verifier, our current implementation provides adequate type and stack consistency checking. It is implemented using a basic fixpoint iteration strategy, expressed as a **monotone framework** [18]. This representation, due to Kam and Ullman [17], allows a generic formulation of such analyses. **Figure 7.1** shows the place of this (the `flow-state` strategy) and derived components in the compilation architecture.

One of the instances of this framework is the **type checker**, where the framework is applied to determine the incoming stack for each instruction. It is defined using two operators, and an argument that determines the initial lattice (i.e., stack):

- The transfer function f_i determines the resulting stack of an instruction (see **Figure 7.5**). In its basic form, this function gets a tuple of a specific instruction in the method and its input stack, and must return the resulting stack. As an implicit argument (using dynamic rules [55]), source tracing information is provided, which is used in case an error is encountered.
- The join operation $\sqcap$ merges the stack at branch targets (i.e., labels and exception handlers). While different control flow edges leading up to a single point in a program must always have the same stack height, the types on the stack may be unified to form a new stack. For example, a `String` and an `Integer` on the stack unify to the `Object` type. In addition to the two input stacks, this function is also provided with implicit source tracing information.
- As an additional parameter, the framework takes the initial stack (ι) of the method or code fragment (which is normally an empty list).

Interesting about these analyses is that they are performed on the generated bytecode, which can come from an arbitrary source language. Bytecode instructions as represented here include source tracing information and inline exception handlers, forming a tree that can be implicitly traversed using the framework. The flow-state algorithm maintains all context information so any errors reported can directly be provided with source tracing information. **Figure 7.6** shows the concrete implementation of the unassigned variable analysis using these techniques.

We have used this technique for other analyses, such as an unassigned variables analysis and an analysis to confirm `final` variables are only written to once. The analyses that we specified are all based on a forward data flow analysis and use as the bottom element ($\perp$) the term `Unreachable`.

```
/** Verify definite assignment of local variables. */
bc-verify-local-assignment =
  bc-flow-state-stack(
    // f_i: handle load/store instructions (see below), or do nothing (Fst)
    bc-local-assignment-state <+ Fst

  , // ⊓: intersect states (e.g., remove if assigned only in one arm of "if")
    isect

  | // ι: initial state; "this" and all method parameters are assigned
    ["this" | <bagof-MethodParamNames>]
  )

/** Add any assigned variable to the state set. */
bc-local-assignment-state:
  (state, STORE(x)) -> [x | state] where not(<Unreachable> state)

/** Give an error if an unassigned variable is loaded. */
bc-local-assignment-state:
  (state, LOAD(x)) -> stored
  where not(<Unreachable> state);
        not(<elem> (x, state));
        bc-error(|"A value must always be assigned to", x, LOAD(x))
```

Figure 7.6 Verification of definite assignment: Java requires each local to have a value assigned to it before use. This bytecode-level analysis confirms this property and generates errors with source tracing information for unassigned local variables.

7.5.2 Optimizations based on bytecode analysis

To allow small transformations of code based on the collected information, we extended the monotone framework with an additional, optional operator g_i. This operator may return a new instruction or sequence of instructions, given an instruction of the program and the state at that point (e.g., the operand stack). Based on this, we also formulated a dead code elimination optimization, which replaces all instructions that have an `Unreachable` state with an empty sequence of instructions. We also implemented peep-hole optimizations that simplify common bytecode patterns. As the pattern matching is implemented as part of the framework, it is fully aware of the control flow in a method. This ensures that the optimization algorithm does not wrongfully replace a fragment of code that has multiple inbound control flow edges.

7.5.3 Type inference for bytecode instructions

Most bytecode instructions normally have a specialized version for various types (see Section 2.1.8). For example, to load a local variable on the stack one of five different instructions must be used (`aload`, `iload`, `fload`, `dload`, or `lload`, depending on the type of the variable). Before running the code, the Java verifier determines the types of all local variables and stack operands using data flow analysis. If an instruction of the wrong type is used, it will throw an exception. For the purposes of both analysis and bytecode engineering, we felt it would be more convenient to generalize this to a single overloaded `load` instruction (see Section 3.4). The type information already available to the integrated verifier, and it is possible to apply this information select a properly typed instruction based on a given pseudo instruction.

In addition to *load* and *store* instructions, specialized instructions are also used for arithmetic operators, comparison operators, primitive type conversions, stack operations, and array access (see the JVM Specification [1], Section 3.11.1). For each of these, we have defined equivalent pseudo-instructions that can be used instead (see Section 3.3). Any regular bytecode program can be expressed using this set, as

it provides direct equivalents for all the regular bytecode instructions. For the purpose of modifying or analyzing existing classes, we created a tool that rewrites the class to use the reduced instruction set. This **disassembler** was fairly straight-forward to implement, as we made sure that a direct mapping exists between the regular instructions and their overloaded counterparts.

The idea of overloaded instructions is not unique, and is for example also applied by the .NET framework, which defines an instruction set where most instructions are overloaded. For Java, the decision to use type-specific instructions was made to allow easy implementation of interpreters (whereas in .NET code is always JIT compiled). Nevertheless, since proper, modern JVM implementations include a verifier that can statically determine the types on the stack, this typing information on instructions is essentially redundant.

The Dryad Compiler implements the instruction type inferencer as a direct extension of the built-in type checker (see **Figure 7.1**). The type checker determines the current types on the stack at any position in the bytecode. The inferencer extends it by defining the g_i transformation operator. This allows it to replace all overloaded pseudo-instructions with type-specific, regular instructions, based on the information collected by the type-checker. Essentially, it maps each overloaded instruction and type combination to a regular, typed bytecode instruction. Some instructions, such as the equality operators (see Section 3.3.8), actually rewrite to a sequence of regular instructions. This is because such instructions know no direct equivalent in the regular set. For example, the `lt` operator, when applied to integer inputs, rewrites to:

```
if_icmpge else
  ldc true
  goto end
else:
  ldc false
end:
```

This fragment of code makes use of one of the regular *if* comparison instructions (see Section 3.3.8), and loads a boolean on the stack to indicate the result of the comparison. Sometimes the code generated in this fashion is not the most efficient, but by using optimizations in a later compilation stage it is possible to remedy that.

Finally, it is possible that a pseudo-instruction is applied to an input type for which no specialized instruction exists (e.g., `add` is given an object and an `int`, or a `float` and an `int`). In such cases we report an error, rather than attempt to convert the input types. Using the source tracing information, such errors are reported not only with the offending instruction, but also with a reference to the originating source code location.

Chapter 8

Related Work

8.1 Compilers and domain-specific languages

Open compilers, such as **Polyglot** [34] and **OpenJava** [52], provide an extensible front-end to a Java compiler. They can be used for building extensions of the Java language, by transformation of the abstract syntax tree (AST) before being handing it to a standard Java compiler. We support similar **preprocessor** implementations, but without the restriction of only generating Java code. Instead, we also allow direct use of the underlying bytecode primitives normally only supported by extending the back-end of a compiler. For example, the implementation of finite state automata (used in parsers), or yield continuations benefit from this (see Section 5). Another open compiler is the very mature **Eclipse Compiler for Java** (ECJ) [56], implemented in Java. It has unparalleled support for incremental compilation, and is integrated in the Eclipse development environment. The **JastAdd** extensible Java compiler [66] is built upon a system of attribute grammars for semantic analysis and inter-type declarations to allow modularization of otherwise tangled concerns. Brothner [10] introduced **Kawa,** an extensible compiler for the Scheme language. It is based on the Kawa language framework, a library that provides a basis for developing (Lisp-related) dynamic languages. It forms an abstraction layer over the low-level **BCEL** bytecode engineering library. These compilers allow for the same preprocessing approach, but also provide an extendible back-end to take full advantage of the JVM. However, to take advantage of this in languages and extensions requires intricate integration in all stages of the compiler. Especially with the large Eclipse compiler, this can be a very involved process. Our approach, as it allows bytecode in the input, provides a more loosely coupled approach that allows taking advantage of the back-end in the initial stages of the compilation process.

Many open compilers, are written in a general-purpose language (the aforementioned are implemented in Java, with the exception of JastAdd). Rewriting programs in such languages, rather than a specialized transformation language, can be a tedious task. Kalleberg and Visser [51] developed a system for interfacing with the ECJ front-end, allowing program transformation with the **Stratego/XT** transformation system [3,4] using the ECJ AST. While our implementation has not reached the level of maturity of ECJ, it is natively written in Stratego/XT which allows complete integration with any language extensions written in it. Using a language-independant exchange format[18], it also allows interoperability with other platforms. Bravenboer *et al.* introduced **MetaBorg** [4,47], a general approach to use the Stratego/XT system to integrate and assimilate embedded DSLs into a host language. It spans from syntax definition extension to rewriting programs, and may be used in conjunction with our approach.

[18]Stratego uses ATerms [55], which support text or XML representation of the AST, but needless to say we also support concrete Java/bytecode syntax.

Macro systems and some pre-processors are generally only concerned with *syntactic* extensions to a language. Recent systems for Java include JSE [43], EPP [44], and JPP [45]. Unlike open compilers they cannot make use of semantic or typing information. It is not clear how these systems scale for large extensions to the base language.

A general problem with generating to source code is the limited ability to include **debugging information** [65]. This information, used by debuggers and to display stack traces (for runtime errors), is normally generated for the generated source code rather than the originating code. Van Deursen *et al.* [50] developed **origin tracking**, a technique for maintaining source location information in rewriting systems. It maintains a pointer to the original code for every construct where a rewrite rule is applied. This information can be passed to a compiler by use of a construct like the C# `#line` directive [42], which allows specifying the original source code line number. We unify these techniques and implement syntax to specify origin information, as well as a library for rewriting with this information. We apply this for composition and separate compilation, both at the Java and at the bytecode level.

Java 5 introduced **annotations** [2], in-code metadata that may be placed at select places (classes, packages, member signatures, and local variables). They can be used for **attribute-oriented programming** [57], a form of declarative programming where annotations are used to rewrite classes. Annotations can form an alternative for true embedded DSLs, at the cost of some syntactic salt and flexibility. While annotated Java still complies to the Java syntax, it can only be correctly compiled by the use of an external annotation processing tool[19]. An example of this is the **AspectWerkz** aspect weaver [60], which uses annotations to represent aspect descriptions. Our compiler can be used to develop such tools, providing facilities for modifying compiled classes and providing debugging information. We however encourage the development of specialized languages, and as such designed an aspect language for compiled aspects on the basis of a regular aspect language with inline bytecode.

Another approach to add language extensions without adapting the syntax or compiler is to use **dummy invocations**. This technique is applied in the **f2j compiler** that translates Fortran to Java source code [48], while translating the goto statement to placeholder statements (e.g., `Dummy.go_to(10)`). More recently the Yielder library also used a similar approach to implement the 'yield' control flow statement [58]. The resulting code can be compiled with a regular Java compiler, but then requires bytecode-level modifications to replace the dummy invocations. Since the Java compiler is oblivious about the special semantics of the invocations, and is unable to properly integrate this in the control flow analysis, this approach has a limited applicability. For example, it may cause the compiler to fail because of seemingly unreachable code, and it may perform optimizations that break the intended rewriting (see Section 5.4.2). A later revision of f2j directly generates bytecode instead, eliminating such problems [49]. In our approach we do this by allowing the required bytecode primitives in the source code, which spares the code generator implementors from a complete bytecode-based implementation.

8.2 Software composition

To aid code reuse, various extensions to Java have been proposed and implemented. These require modification of Java classes at the source or bytecode level. Kiczales *et al.* [32] introduced **AspectJ**, a popular implementation of **aspect-oriented programming** [31,40] for Java, aiding in separation of concerns by allowing the insertion of cross-cutting concerns into classes. Initially implemented to integrate such aspects at the source level, AspectJ was later adapted to do this at the class file (bytecode)

[19]Sun's annotation processing tool (apt) is commonly used for this, which is restricted to only generating new code rather than modifying code, and to operation at the class member level rather than the method body level.

level. This meant it no longer requires access to the source code of classes and aspects, but it required the AspectJ developers significantly more implementation effort to integrate this into the existing Eclipse compiler. Using a combined target language of Java and bytecode, we allow the aspects can be directly composed into target classes regardless of their base language, Java or bytecode. This allows for very slim aspect weavers that may implemented separately from the compiler (see Section 6.3).

Traits, a concept introduced by Schärli *et al.* [6], apply composition to integrate and combine modules of reusable code into classes. Existing implementations do this at the source level [6,24,26,37]. This preprocessor approach simplifies the composer as it need not be concerned with the remainder of the compilation. We maintain this advantage, but with our extended target language also accept compiled classes and traits as inputs (see Section 6.2).

8.3 Bytecode and assembly

The **Soot** framework [35] allows analysis and optimization of Java bytecode. It provides different levels of language abstractions over representing bytecode, ultimately completely abstracting over the operand stack, forming a Java-like language. This provides opportunities for optimizations at different levels. We do not abstract over the language per se, but offer pseudo-instructions that can simplify analysis and emitting code. Lance *et al.* [22] perform analysis over bytecode as an alternative to Java source analysis, taking benefit of the simpler structure of the language. Using **Jaristotle** to analyze a low-level bytecode representation, they use debugging information to maintain a link to the source code. We implement our own set of analyses and optimizations, as well as a verifier on the basis of monotone frameworks [18] (see Section 7.5). Using source tracing (debugging) information, we maintain a similar link to the original source code, allowing certain error conditions checked and reported after the code is emitted.

Some programming languages, such as C++ and Ada allow **inline assembly** code [41,59]. This is often used for optimizations or obfuscation, which is also possible with the Java/bytecode language. Assembly code, however, is a representation of actual CPU instructions, and is therefore much more low-level and less safe than bytecode. This means it can also access to processor-specific instructions or system calls, not supported in bytecode. Inline assembly code is also not very portable, as it depends on the compiler and the CPU architecture it compiles for [59]. Bytecode on the other hand is platform-independent as it targets the Java Virtual Machine. The dependency on the CPU and compiler, together with the lack of structure of assembled code, means that it is very hard to manipulate natively compiled programs, even after disassembly. We apply such manipulations of bytecode for the purposes of separate compilation and modifying compiled classes; with assembly code it is not feasible to do this. For inter-operability with surrounding code, inline assembly only supports local variables through the use of the stack or registers, while we allow the use of named local variables.

Like Java, the C# language [42] is compiled to a bytecode language, the Common Intermediate Language. It also uses a verifier to guarantee the safety of the bytecode. Using attributes on classes or methods, it is however possible to disable the type-safety features of this verifier (as long as the security policy allows it). C# applies this fort **"unsafe code"**, a series of pointer-based language constructs that allow direct access to memory (of the virtual machine). Unsafe code is often used for interoperability with native libraries written in C or C++, or sometimes for minor optimizations. Unlike unsafe code, embedded bytecode adheres to generally the same safety restrictions as the host language (i.e., Java). For interoperability, the **Java Native Interface** (JNI) [46] however provides similar features (and unsafety) as unsafe code, albeit with a higher performance overhead since it cannot provide the same level of integration with native (pointer-based) data structures.

Chapter 9

Conclusion

Java provides a safe, mature, and ever-evolving platform for developing applications, from small applets to large enterprise applications. In addition to the Java language, the platform is frequently used to host other general-purpose languages [9,48,49,11,27,28]. To increase programmer productivity in specific areas, domain-specific languages have been and continue to be developed. Java and bytecode form simple, portable compilation targets implementing languages in both categories. Combining the two languages has a truly synergetic effect: The raw power of bytecode is combined with the convenience and familiarity of Java. The combined language allows rapid development of language extensions that operate in a separate compilation stage, which is further supported through source code tracing to provide compile-time errors and debugging information.

The Dryad Compiler also opens the doors for new, more fine-grained forms of separate compilation. Embedded languages and software composition systems, such as aspect weavers [31,32,40], or compilers for traits [6], can use the language as a foundation for combining classes and methods up to instruction- and expression-level precision. Using fine-grained separate compilation, reusable units of code can be pre-compiled with the same precision. With the integrated debugging information facilities, these composed fragments of code can be traced back to their original source files. This in turn allows accurate source location information for error messages as well as debug-stepping, supporting software composition using a preprocessor approach.

9.1 Future work

Our current experience with the Java/bytecode language is based on the use of code generation and software composition applications of limited size. In the future, it would be interesting to see how this system scales up, and how the separate and multiply staged compilation performs for larger applications. Our current implementation does not yet support the complete Java language, as we simply implemented the language features as they were demanded. While this is not a limitation for implementing new languages or systems that do not depend on the complete Java language, for larger applications and for more general use it may be essential to offer support for the complete Java language.

As our current compiler implementation compiles to native code rather than code for the JVM, integration options with Java applications are limited. It would be interesting to compile a future version as a native Java program. This would offer new opportunities, such as run-time or load-time code generation and manipulation. Such techniques are already actively applied in the field of aspect-oriented programming [60], and provide increased flexibility, albeit at the cost of performance. Using the pre-compilation techniques such as we already applied to aspects (see Section 6.3.5), such overhead can be minimized, but can never be completely avoided.

References

[1] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[3] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer *et al.,* editors, *Domain-Specific Program Generation, volume 3016 of Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[4] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *The 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[5] P. Hudak, Modular Domain Specific Languages and Tools. *Fifth International Conference on Software Reuse,* 1998.

[6] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[7] Stratego/XT Extensions. http://www.strategoxt.org/AdditionalPackageDownload/.

[8] F. Yellin, Low level security in Java. *Fourth International World Wide Web Conference*, (Boston, MA), World Wide Web Consortium, Dec. 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.

[9] S. Pedroni and N. Rappin. *Jython Essentials*. O'Reilly and Associates, 2002.

[10] P. Bothner. Kawa - Compiling Dynamic Languages to the Java VM. In *Usenix conference in New Orleans*, June 1998.

[11] M. Odersky and PWadler. Pizza into java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146-159. ACM Press, 1997.

[12] Sun MicroSystems. Free and Open Source Java Project Overview. http://www.sun.com/software/opensource/java/project_overview.jsp (retrieved on 2007-2-1).

[13] A.-R. Adl-Tabatabai, B.T. Lewis, V. Menon, B.R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. *PLDI 2006: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*.

[14] B.D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C.C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional execution of Java programs. *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*.

[15] X. Leroy. Java bytecode verification: An overview. *Computer Aided Verification, 13th International Conference*, volume 2001 of *LNCS*, pages 265–285, Paris, France, July 2001. Springer-Verlag.

[16] E. Haase. JustIce: An implementation of a free class file verifier for Java, Technical Report, Institut für Informatik, Freie Universität Berlin, 2001, http://bcel.sourceforge.net/justice/.

[17] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica 7*, 305-317, 1977.

[18] F. Nielson, H.R. Nielson, C. Hankin. *Principles of program analysis*. Springer-Verlag, 1999.

[19] E.W. Dijkstra. Goto considered harmful. Communications of the ACM, 11(3):147–8, 1968.

[20] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[21] J. Clements and M. Felleisen. A Tail-Recursive Machine with Stack Inspection. *Transactions on Programming Languages and Systems,* 2004, http://www.ccs.neu.edu/scheme/pubs/cf-toplas04.pdf.

[22] D. Lance, R.H. Untch, N.J. Wahl: Bytecode-based Java program analysis. *ACM Southeast Regional Conference*, 1999

[23] C. Smith and S. Drossopoulou. Chai: Traits for Java-like Languages. *The Ninetieth European Conference on Object-Oriented Programming (ECOOP'05)*, 2005.
[24] J. Reppy and A. Turon. A foundation for trait-based metaprogramming. *International Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.
[25] B. Alpern , S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar, The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, v.44 n.2, p.399-417, January 2005.
[26] L. Liquori, A Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM Transaction on Programming Languages and Systems*, 24 pages, ACM Press, to appear, 2007
[27] J. C. Hardwick and J. Sipelstein. Java as an Intermediate Language. Technical Report CMU-CS-96-161, Carnegie Mellon University, August 1996.
[28] J. Gough and D. Corney. Evaluating the Java Virtual Machine as a Target for Languages other than Java. *Joint Modula Languages Conference*, 2000.
[29] A. Krall and J. Vitek. On Extending Java. *Joint Modular Languages Conference*, 1997.
[30] S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in Sather. *ACM Trans. Program. Lang. Syst.* 18, 1 (Jan. 1996), 1-15.
[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwing. Aspect-Oriented Programming. *ECOOP '97*, Springer Verlag, p. 220-242, 1997.
[32] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. *The 15th European Conference on Object-Oriented Programming,* p. 327-353, June 18-22, 2001.
[33] Sun Microsystems, (2006), Java Platform, Standard Edition 6, API Specification. http://java.sun.com/javase/6/docs/api/.
[34] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. International Conference on Compiler Construction (CC), 2003.
[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. *Proceedings of the 1999 Conference of the Centre For Advanced Studies on Collaborative Research,* Mississauga, Ontario, Canada, 1999.
[36] T.J. Parr and R.W. Quong. ANTLR: A predicated-LL(k) parser generator. S*oftware – Practice and Experience*, 1995.
[37] P.J. Quitslund. Java traits – improving opportunities for reuse. Technical Report CSE-04-005, OGI School of Science & Engineering, Beaverton, Oregon, USA, Sept. 2004.
[38] T. Parr. ANTLR Code Generation weblog. http://www.antlr.org/blog/antlr3/codegen.tml
[39] T. Parr. Language Translation Using ANTLR and StringTemplate. http://www.codegeneration.net/tiki-read_article.php?articleId=77.
[40] A. Rashid and L. Blair. Editorial: Aspect-oriented Programming and Separation of Crosscutting Concerns. *The Computer Journal,* 46(5):527–528*,* 2003.
[41] B. Rao. Inline assembly for x86 in Linux. http://www-128.ibm.com/developerworks/library/l-ia.html.
[42] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language, 2nd Edition*. Addison-Wesley, 2006.
[43] D. K. Frayne and Keith Playford. The Java syntactic extender (JSE). *The 2001 Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '01)*, pages 31–42, Tampa, FL, USA, 2001
[44] Y. Ichisugi and Y. Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. *ISCOPE '97*, LNCS 1343, pages 153–160. Springer, 1997.
[45] J.R. Kiniry and E. Cheong. JPP: A Java pre-processor. Technical Report CS-TR-98-15, California Institute of Technology, Pasadena, CA, September 1998.
[46] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*, Addison Wesley Longman, Inc., 1999.
[47] M. Bravenboer, R. de Groot, and E. Visser. MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. *Summer School on Generative and Transformational Techniques in Software Engineering* (GTTSE 2005), Braga, Portugal, July, 2005.
[48] D. Doolin, J. Dongarra, and K. Seymour. JLAPACK | Compiling LAPACK Fortran to Java. *Scientific Programming,* 1999.
[49] K. Seymour and J. Dongarra. Automatic Translation of Fortran to JVM Bytecode. *Concurrency: Practice*

and Experience, 2003.
[50] A. van Deursen, P. Klint, and F. Tip. Origin tracking. Journal of Symbolic Computation,1993.
[51] K.T. Kalleberg and E. Visser, Fusing a Transformation Language with an Open Compiler, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07),* 2007.
[52] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A class-based macro system for java. *The 1st OOPSLA Workshop on Reflection and Software Engineering*, 2000.
[53] S.C. Dewhurst, S.C. *C++ Gotchas*, Addison-Wesley 2003.
[54] S.C. Dewhurst. A Matter of Judgment. *Dr. Dobb's Journal*, October 2003. http://www.ddj.com/cpp/184403876
[55] M. Bravenboer, K. Trygve Kalleberg, R. Vermaas, and E. Visser. Stratego/XT Tutorial, Examples, and Reference Manual (latest), 2007. http://www.strategoxt.org/StrategoDocumentation/.
[56] The Eclipse Foundation. JDT Core Component. http://www.eclipse.org/jdt/core/.
[57] XDoclet team: XDoclet: Attribute-Oriented Programming. http://xdoclet.sourceforge.net/.
[58] A.B. Dov. How to write Iterators really REALLY fast. http://chaoticjava.com/posts/how-to-write-iterators-really-really-fast/.
[59] G. Singh. Inline Assembly in GCC Vs VC++. http://www.codeproject.com/cpp/gccasm.asp.
[60] J. Bonér. What are the key issues for commercial AOP use - how does AspectWerkz address them? *3rd international Conference on Aspect-Oriented Software Development*. ACM Press, 2004.
[61] R.B. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, L. Smith, and L. Walton. A software engineering experiment in software component generation. *18th International Conference in Software Engineering*, 1996.
[62] java-csharp: C#-inspired language extensions for Java. https://svn.cs.uu.nl:12443/repos/StrategoXT/java-csharp/tags/2006/.
[63] P. Wu, S. Midkiff, J. Moreira, and M. Gupta. Efficient support for complex numbers in java. *The ACM 1999 conference on Java Grande.* ACM Press, 1999.
[64] B. Blount and S. Chatterjee. An Evaluation of Java for Numerical Computing. *ISCOPE*, 1998.
[65] D.G. Waddington and B. Yao. High fidelity C++ code transformation. *The 5th workshop on Language Descriptions, Tools and Applications*. Electronic Notes in Theoretical Computer Science, Elsevier, 2005.
[66] T. Ekman and G Hedin. The JastAdd Extensible Java Compiler. *The 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, Montreal, Canada, October 2007. To appear.
[67] M.G.J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, Compiler Construction (CC'02), volume 2304 of Lecture Notes in Computer Science, pages 143–158, Grenoble, France. Springer-Verlag, April 2002.
[68] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: proof rules and implementation. In Proceedings of the Seventh Workshop on Formal Techniques for Java-like Programs, 2005.
[69] D. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering - Special Issue on Formal Methods 16*, 9 (Sept. 1990), 1024-1043.