OPTIMIZATIONS IN DISTRIBUTED RUN-TIME COMPILATION

BY

LARS REDER CLAUSEN

B.Sc., University of Århus, 1994
KAND, University of Århus, 1998

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

Replace this page with “red-bordered” form

Abstract

Distributed run-time code generation is a useful technique that can easily be implemented using the technique known as "compositional semantics." In this dissertation, we describe a new system, Jumbo, which adds run-time code generation features to Java. Its design allows code generators from disparate sources to be combined dynamically. Jumbo can be used to create highly adaptable code that can be distributed in binary form.

In order to handle the increasing complexity of software, programmers have long been creating reusable function libraries. These libraries can be distributed in binary form, allowing distribution that does not reveal implementation secrets. While libraries allow sharing or sale of code fragments, the granularity and inflexibility of pre-compiled functions force the libraries to be either overly general or prohibitively specific.

Run-time code generation systems present a light-weight alternative to function calls, where fragments of code, not necessarily in the form of whole functions, can be manipulated in order to create higher-quality specialized code. However, the currently existing run-time code generation systems either operate at the source level or take a non-compositional approach, and thus are not amenable to binary distribution.

Jumbo implements a novel form of run-time compilation, where code fragments in binary form from several sources can be combined at run time to form efficient specialized code. By basing Jumbo on compositional semantics, we avoid the monolithic nature of traditional code generation systems.

A distributed run-time code generation system allows some previously unknown functionality. For instance, in database systems, a client can create query functions that interact directly with the servers internal calls without either the query or the database implementation needing to be disclosed. Render farms can allow clients to specify textures with code that is merged with local, specialized functions, without disclosing their texture details. Anonymity, copyrights and trade secrets can be retained without sacrificing efficiency.

Jumbo is also useful for non-distributed systems that can take advantage of staged compilation for efficiency. We show several examples of how Jumbo can be used simply as an optimization tool, allowing

user-controlled and domain-specific optimizations. Jumbo can additionally be used in a manner similar to that of macros, providing a way to codify common programming practices.

We also consider the question of optimizing the code generators and the generated code, and present a rewriting system for Java abstract syntax trees designed to optimize the code generators. This general-purpose optmizer exploits the compositional nature of Jumbo to make aggressive optimizations on the code generators for the individual code fragments.

Acknowledgements

My advisor, Sam Kamin, provided not only advice, help, and direction, but also the financial support that allowed me to concentrate on this work.

I owe much of the work done on this thesis to my beloved wife and muse, Mickey, who helped me through times of waxing and waning enthusiasm, and who never doubted my ability to succeed.

Many thanks also to Ava Jarvis, who designed the parser and AST in the first place, and helped me with numerous technical problems as well as relaxing banter along the way.

Olivier Danvy was kind enough to take time out of his busy schedule to read my thesis and give highly useful comments. I owe him thanks for many insightful comments both now and earlier, and for helping me towards the study.

Green House and Phoenix House, the two cooperatives where I lived for most of my time while in the US, provided comfortable and welcoming frames for my stay. I shall remember the houses, the housemates, and the cats fondly.

The ACM chapter at the University of Illinois provided a much-needed place to relax and gripe about my problems. Thanks, guys!

I also owe thanks to the local Belegarth Medieval Combat Society group, Numenor, for being a good bunch of friends and a wonderful place to destress.

Finally, without the continued support and help of my parents this thesis would never have happened.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The complexity of software was labelled a “software crisis” as long ago as 1968[59], and it has only increased since then. Projects on the order of millions of lines of code are not uncommon. To handle this problem, efficient reuse of code has become an indispensable tool.

1.1 Libraries and Components

The primary way to reuse code has long been to encapsulate related functionality into a *library*. The division into separate pieces enforces a well-defined interface, making it feasible to reuse code across unrelated programs. This style of reuse has been highly successful, with a multitude of libraries being available for various platforms. Some significant libraries, like STL[62], GTK[83], and Lapack[2], are even available for multiple platforms, with an interface that hides platform differences. Most programming languages have some system for handling libraries.

The idea of libraries has been extended into *components*, self-contained entities that implement one or more externally defined interfaces. Components are often designed to be language neutral instead of being tied to a specific programming language. To interface with programs, they use *component systems* like CORBA[32], COM[70] or JINI[90] that allow programs to locate components that implement particular interfaces.

One problem with both libraries and components arises because they are designed to provide a flexible solution to a number of related problems. While this flexibility is generally a boon for the programmer, who is more likely to be able to use the library unchanged, it requires that the library solve a more general problem than most programs require. This generality leads to performance degradation, both because the

code is not optimized for the specific use, and because the library itself becomes larger, using more resources than necessary. The ideal library would not only be usable in a wide variety of situations, but it would be able to adapt itself to suit the situation at hand.

Some library systems allow tailoring the library to the specific use at compile time. In particular, the STL libraries for C++ make use of C++'s template mechanism to create specialized code. Not only does it let the programmer get a specific version of the library function, an STL implementation can choose to automatically inline small functions where appropriate.

1.2 Code Generation

In addition to the question of how libraries should adapt to their use, there are a number of cases where we would like a program to adapt to input that is not known until after the program has been deployed. In many cases, some run-time information is static or remains constant over a significant time, and this information can be used to make more efficient programs. The information can be simple data, such as the contents of a fixed matrix in matrix multiplication[49], data characteristics, like the size of samples in an FFT transform algorithm[27], or even structural information, like the composition of a set of network filters in a router[85, 49]. In the case of domain-specific languages, entire programs are static but created by the user.

All of these situations can benefit from a run-time adaptive system. Such systems are said to be "staged," as they split compilation into several stages: one stage at normal compile time, and one or more stages at run time. Such systems range from fully automatic run-time partial evaluation to semi-automatic staging with user annotation to systems that support direct creation and manipulation of code fragments at run time.

Partial evaluation[35] systems offer an automatic and quite powerful code adaptation system. Given partial (static) input to a program, a partial evaluator attempts to automatically figure out all parts of the program that only depend on the static input. It then evaluates those parts, leaving only the parts of the program that require knowing the remaining (dynamic) input. Partial evaluation has been known to give speed improvements of an order of magnitude, and is applicable to a wide range of problems. However, its automated nature can cause unrestricted code growth and limits the possibility of applying domain-specific knowledge in specialization.

To allow more flexible adaptation, several systems have been implemented that allow direct programmer

control of the code generation process. This allows better restrictions on code growth, as well as use of domain-specific optimizations to create highly optimized code. However, all of the current systems are monolithic in nature, in that they assume that all code is being created in the same place.

In this dissertation, we investigate the use of compositional semantics in the implementation of a run-time code generation system that allows distributed code creation. We describe a Java system with run-time code generation, dubbed Jumbo. Jumbo allows the programmer to manipulate pieces of code like program objects, passing them to functions, piecing them together, and eventually turning them into executable code. Code objects can be anything from single variables or expressions to full classes. The code objects are not source text, but binary compilations of the source, which both gives increased efficiency and hides the source from unwelcome perusal. The system is written entirely in Java, and so can easily be deployed on any platform with a compliant Java system. Jumbo currently implements most of Java 2 (version 1.3).

We have chosen to use Java because one of its main design goals is that executables should be independent of execution platform. This goal is achieved by the use of a platform-independent byte code that is executed in a virtual machine. Thus, any library written in Java is automatically usable on any platform, without requiring recompilation. This portability greatly facilitates reuse, as only one compilation is needed to create libraries that can be distributed to any conforming platform. Not only does platform independence remove the problems inherent in having to recompile libraries on a new platform, it also means that library designers do not have to divulge their source code in order to have the library be usable on new platforms. Thus, distributed code generation will be possible without having to deal with the vagaries of different computer architectures.

The dissertation is structured as follows: In chapter 2, we summarize previous research into staged compilation and Java optimization. In chapter 3, we give an overview of the current Jumbo system, and in chapter 4 we show some examples of how Jumbo can be used and how it performs. In chapter 5, we describe various ways performance could be improved, and in chapter 6 we present a rewrite system that performs a specialized partial evaluation on Jumbo code generators. In chapter 7, we evaluate the results and give suggestions for future research.

The reader is assumed to be familiar with compiler construction and programming language concepts, as well as have a basic understanding of code optimization principles. A working knowledge of Java and stack-based implementation is also assumed.

Chapter 2

Background

A program can be seen as a recipe for solving certain problems. The more problems a program can solve, the more widely applicable it is. However, a highly generalized program may not be easy to use or particularly fast. Programmers thus face a tradeoff between how many problems a program can solve, and how easily and efficiently it can do so. However, making it easy and efficient to solve particular problems can be a daunting task, especially if the problems are complex or heavily intertwined. Trying to write such specific programs for a number of problems quickly becomes overwhelming. To save effort, programmers strive to reuse code rather than having to invent the wheel over and over.

The most widely used technique for code reuse is that of system libraries. These self-contained collections of functions are easily distributed as binaries, and support for calling them is basic to most languages. However, they tend to be heavy-weight: Their smallest unit is the function, and to be of wide applicability, they must include support for a number of different cases. When trying to use them for small-scale, specialized purposes, efficiency suffers, or programming becomes cumbersome. Libraries that have a persistent state, known as components, face the same granularity problem.

Instead of having to choose between over-general libraries and implementing a specialized solution from scratch, it should be possible to have something akin to libraries, but with the ability to adapt itself to the specific task. Such adaptation typically takes the form of generating specialized code for that task. In this chapter, we describe various approaches to code generation and specialization. This chapter is not intended as a comprehensive description of all code-generating systems, but merely to show the various flavors of code generation and their pros and cons.

2.1 Source-based Specialization

The easiest and most commonly implemented kind of specialization works at the source code level. This allows a simple implementation that works with strings, is as portable as the base language itself, and, most importantly, does not have to count the specialization time against run-time performance. However, it cannot avail itself of data known only at run-time, such as profiling and run-time constants, and it requires access to the source code, which is not always an option. In this section we describe a selection of source-based specialization systems.

2.1.1 Source-generating Systems

The most commonly used code generators are those that generate source code to be compiled with a program. The generators can either take the form of stand-alone generators, such as Lex[53], Yacc[37], and some aspect-oriented systems[34], or it can be part of a programming language, like Lisp and Scheme macros[48], C++ templates[18, 77, 88], and AspectJ[43].

Lex and Yacc are prime examples of stand-alone generators. Almost all compilers, and a number of other systems (including Bash[26], NetHack[1][24] and LessTif[3]), use some implementation of these to perform the tedious but easily automated tasks of lexing (turning streams of characters into words) and parsing (turning a series of words into structured forms). They also show the limitations of this approach: Each has its own special language, normally quite different from the languages that it outputs in, and they require a separate program to turn them into usable source. These limitations act as barriers to entry: only tasks that are complex, automatable and easily isolated warrant the effort of using stand-alone generators. Indeed, several general-purpose libraries include light-weight parsers for simple parsing tasks.

More generally applicable are macro systems, which allow pieces of source code to be assembled as part of the compilation process. They originated with assembly languages, and were brought into symbolic programming languages as early as 1960. The first paper on LISP[58] includes a `subst` function that is essentially a macro expansion, and some form of macros have been part of LISP and Scheme since then. These macros are both powerful and well integrated into the language. Paul Graham describes how Lisp macros were used aggressively in the Viaweb on-line shopping system[30]. Due to the flexibility allowed

[1] Without which this dissertation would have been finished either a lot sooner or not at all.

by the macros, they could adapt faster than their competitors, and as a result Viaweb is still used as the back-end for Yahoo stores while most of their competitors have vanished.

The C preprocessor[42] is a primitive but widely used macro expander. Its major uses are defining constants, marking pieces of code for conditional compiling, and realizing local syntactic improvements. It is commonly regarded as unstructured, inelegant and likely to cause more bugs[23, 78]. It is, however, still in widespread use, especially for ensuring portability.

C++ retains the C preprocessor macros, but adds templates[77], a macro-like system for generic classes. Unlike C macros, they are a part of the C++ compiler, and thus aware of types. They are used to create new classes that implement type-specialized versions of more generic classes, e.g. `List<int>` refers to a constructed list class that only allows integers as elements. Since the template parameters may contain constants and even expressions that are evaluated at compile-time, C++ templates allow for a certain style of partial evaluation called "template meta-programming"[18, 89, 88]. Template meta-programming in C++ has the obvious advantage that it is automatically part of every standards-compliant C++ compiler. However, the syntax is unpleasant, and it can be hard to control the code explosion.

Aspect-oriented programming (also known as subject-oriented programming)[34, 61] is a programming paradigm where different aspects are isolated into separate pieces of source code, even when they, in usage, may be highly intertwined. This "separation of concerns" makes the code more maintainable and easier to debug. The first tools for aspect-oriented programming were entirely source-transforming tools[44]. They allowed different aspects of a program, such as logging or synchronization, to be described separately and then woven into a single program to be compiled. AspectJ[43] later evolved away from the source-transformation process into a language with aspects as proper members, and as such is not a code generating system any more.

2.1.2 Source-Based Partial Evaluation

Partial evaluation[39, 35] is an automatic specialization technique that creates specialized programs based on partial input. Given some of the input to a program, a partial evaluator will evaluate any expressions in the program that do not depend on the remaining input. The remaining part of the program is then "residualized" to form a new program that takes the remaining input. The original and the residual program should have the same semantics, but the residual program should ideally be faster. The last 15 years has seen a flurry of

activity in this area, in particular for functional languages.

One of the main advantages of partial evaluation is that very little programmer input is required. Typically, only the set of static variables needs to be declared, although adherence to particular programming styles can increase the efficiency of the partial evaluator. Given the required declarations, the partial evaluator can create highly specialized code. Speedups of an order of magnitude are not uncommon for certain domains.

However, the high degree of automation is also one of the greatest disadvantages of partial evaluation, in that only those transformations that can be done automatically are available. More general-purpose specializers allows the programmer to apply domain-specific knowledge in order to judge the applicability of transformations and to add transformations that would otherwise not be possible. Likewise, for statically typed languages there is no way to add new types more suitable to the domain at hand.

Another disadvantage of partial evaluation is that it can easily lead to code explosion. Overeager partial evaluators can lose more performance by overflowing instructions caches than they gain by using specialized code. Controlling the code generation typically requires extra annotation or programmer control, rescinding the benefit of automation.

Partial evaluation is an important specialization tool. Especially for functional languages, partial evaluation at compile time can yield significant benefits. Imperative languages have proven somewhat more difficult to effectively partially evaluate, though some success has been seen[55, 13]. Source-based partial evaluation is more common than run-time partial evaluation, due to the ease of implementation and fewer time restrictions. Run-time partial evaluation will be described in section 2.2.1.

2.1.3 AST-Based Code Manipulation

Some systems eschew the textual flavor of the source generating approaches, and allow the programmers access to the abstract syntax trees of their programs. These include Engler's MAGIK[21], the Intentional Programming approach of Simonyi[74], and Sirkin et al.'s GenVoca system[76, 75]. This approach avoids many pitfalls of source-level code manipulation, such as inadvertent variable capture and expression duplication, by using references in the abstract syntax tree instead of textual similarity to reference entities. Additionally, it allows better interfacing with both the compiler and the programming tools.

MAGIK[21] allows the programmer to define AST-level datatypes along with datatype-specific interactions with the compiler. Such interaction can include optimizing based on domain knowledge, security checks, and debugging statements. The compiler interaction is very explicit and allows significant control of operations at the cost of ease of use.

Jak is a meta-programming system for Java used in the Jakarta Tool Suite[4]. It allows the programmer to manipulate abstract syntax trees at run time. It has a powerful quotation syntax with several methods for manipulating the ASTs. However, it doesn't include any run-time compilation, only output to Java source.

Intentional Programming[74] does away with the traditional text-editing style of programming, and instead allows domain-specific editors to manipulate their parts of the code. It resembles MAGIK in that it allows extending the compiler with specialized constructs, but goes one step further by including the programming environment. Instead of representing source code as text, it uses a graph structure for which various editors, debuggers and compilers can be used. This representation allows domain-specific views and effects to be added by the user and intertwined with source code from other domains. For instance, matrix manipulations could be represented in a mathematical notation in the middle of normal source text.

GenVoca[75] is a full development system based on the idea of software system generators, i.e. higher-level definitions of the characteristics of the system, implemented automatically by a set of generators. Its aim is to automate the software engineering process to the point where a very high level specification can automatically be turned into highly efficient code.

Because of their use of S-expressions[58], Lisp-like systems do not have as sharp a separation between compile time and run time as most other languages. Run-time generated S-expressions can be interpreted and in some cases compiled at run time. The downside is that due to the limited type information, effective compilation and optimization of Lisp-like languages is difficult. We would like to combine the ease of run-time compilation found in Lisp-like languages with the static typing and compiled efficiency found in Java.

Source-level specializers, even those that access abstract syntax trees, do have several shortcomings: Because they work at the source level, they require access to the source code of the program, which may not always be possible. They also need more or less extensive compilation systems, which may involve system setup or long compilation times. These requirements render source-level specializers less useful for fast combination of disparate sources of code.

2.2 Run-Time Specialization

Run-time specialization allows the use of run-time invariants and profiling information in the code generation process. This extra information allows more numerous and accurate transformations, but is limited by the requirements of using a minimal amount of compile time. This trade-off between code quality and compilation speed exists in normal compilers, but is especially critical for systems that create code at run time.

2.2.1 Run-Time Partial Evaluation

Run-time partial evaluation systems differ from other run-time specializers in that they are all declarative. The programmer declares which functions or variables should be specialized, and the system then performs *binding-time analysis*, an analysis that determines which of the program's variables are bound at compile time and which at run-time. This analysis indicates which pieces of code can be specialized away, and which must be residualized, i.e. become the specialized program.

The Tempo system[13] performs run-time partial evaluation for C programs by manipulating blocks of machine code. While it can generate efficient code for some kinds of problems in a semi-automated fashion, it does not provide a general programming framework in the way macros and templates do. It uses the standard C compiler to generate code at compile-time, with labels interspersed to mark blocks of code. These blocks are then copied at run-time to create specialized code[63].

JSpec[72] is a run-time partial evaluation system for Java, built using the Harissa Java-to-C compiler and environment. It uses "specialization classes" to indicate which fields and variables are static and which dynamic. While the concept of specialization classes is interesting and merits more investigation, the loss of portability due to the dependency on the Harissa environment limits the direct usefulness of JSpec.

Fabius[50] is an ML-based run-time partial evaluator, highly optimized for speed of code generation. Using specialized emitter code, they can output MIPS machine code at a rate of down to six instructions executed per instruction generated. Rather than using an explicit specification, Fabius uses currying to indicate the areas that may be good targets for partial evaluation.

MetaML[82] is a variation of ML and provides a general syntax for staged binary compilation. It provides a simple staging syntax, and a `run` function that will evaluate and run staged code. It requires that a MetaML program with all staging syntax removed still be valid ML program, and automatically

captures variables between staging levels. The staging syntax serves to indicate to a partial evaluator which areas of the code should be delayed. Because of the regularity and functional nature of ML, this syntax is enough to perform a variety of code-generation functions.

Masahura and Yonezawa presents a system called BCS (for ByteCode Specialization) that performs run-time partial evaluation directly on Java bytecode[56]. Like Tempo, they use templates to create bytecode efficiently. They use a standard binding-time analysis system that allows the programmer to specify which method parameters are static or dynamic.

DyC[31] is an extension to C that supports some aspects of partial evaluation, combined with user annotation. The user must annotate the static variables, and those annotations are used together with binding-time analysis and aggressive optimizations to eliminate statically known statements. They stop short of doing the inlinings a partial evaluation system would do, focusing instead on code generation speed. They use template-based code generation, and obtain significant speedups for some programs, with break-even points as small as a single iteration in several cases.

Template-based code generation offers an easy way to generate code quickly. Most of the code generation can be done at compile time, so only copying of blocks of memory remains to be done at run time. However, creating templates requires that the types of variables are known at compile time, which may not be the case in a distributed setting. In particular, it rules out the creation of new types at run time,.

Like their source-based brethren, run-time partial evaluators work best on functional languages, and have to carefully control the amount of code generated. The problem of code explosion is multiplied for run-time partial evaluators, as code generation time is a run-time cost. Thus, the ability to control code generation in a flexible way becomes of utmost importance in run-time code generation.

2.2.2 Run-time Code Generation

In contrast to the declarative run-time specializers, some systems take a procedural approach. A *procedural specializer*, rather than using automated analyses, lets the programmer manipulate fragments of code at run-time in order to create a specialized program. This approach allows a number of specializations that declarative specializers may not be able to perform, at the cost of extra programming effort. In particular, procedural code generation allows domain-specific specialization, using knowledge of special invariants or probabilities of the target domain to perform optimizations that would not be obvious to a generic specializer.

Noticeably, a number of these procedural specializers operate on imperative rather than functional languages. While the lambda-calculus and derived functional languages have an extensive body of research on analysis and automatic specialization, less is known about how to do binding-time analysis and specialization for languages like C and Java. Procedural specializers allow some of the specializations that can be automatically performed on functional languages, without having to solve the problems of automatic analysis and specialization for imperative languages. Experimenting with procedural specializers can help form the body of experience needed to understand automatic specialization of imperative languages, but procedural specializers have uses beyond that which can be achieved by automatic specialization.

‘C[22] was the first run-time code-generation system for an imperative language. It adds syntax for specifying and manipulating code fragments in ANSI C. Like MetaML, it captures free variables, but it does not require the code to be valid if all annotations are removed. Because of the distinction between expressions and statements in C, such a requirement would be unworkable, as you would never be able to manipulate code representing a statement. ‘C allows run-time constants to be lifted into code fragments as constants. ‘C also statically type checks its code fragments. Type safety, however, implies that all types are known at compile time, so no new types can be declared at run time. This places some potential uses out of reach.

‘C has some limitations due to its implementation. It does not allow the creation of new types, only implementations of existing ones. Thus, only a subset of C is actually available for run-time code generation. It also does not quite treat its code fragments as first-class citizens: there is implicitly a “current function” being generated, to which all new code fragments belong. Thus, a code fragment only has meaning within its creation context, and cannot easily be distributed.

DynJava[65] is a type-safe run-time code generation system for Java inspired by Tempo. Like Tempo, it uses the native compiler (`javac`) to create code templates at compile time, and then copies these templates at run time to generate code. The type-safety of DynJava prohibits the creation of new types (classes) or methods. DynJava essentially only allows the dynamic generation of implementations of statically declared methods.

The DynJava paper shows one example with timings, that of a loop-unrolling version of a fast Fourier transform algorithm. Their best result is on a size 2048 FFT, where they get a speed-up of 22%, though it is unclear whether the time measured includes the time spent copying code templates. They have high startup

costs, with their first run of a session taking several seconds on a 500MHz Pentium III. This time probably includes loading the classes for the code generation system.

There are several systems that provide access to Java Bytecode at run-time, allowing the creation of new Java code. These systems include JOIE[12] and the Java Class API[19]. While these allow very flexible code generation, the bytecode syntax is more akin to assembly language than to Java, and thus bytecode-manipulating systems are generally too complicated for large-scale use.

The "linguistic reflection" system proposed by Kirby, Morrison and Stemple[45] allows run-time code generation for Java through creating source code and invoking the Javac compiler API. While this allows a somewhat higher-level approach than the Java Bytecode compilers, the syntax is still hardly amenable to large-scale usage.

2.2.3 Run-Time Compilers

Instead of choosing between the efficiency of statically compiled code and the portability and immediacy of interpreted code, some languages attempt to combine the two by using run-time compilation. These systems either distribute source code or some portable compiled code, which is then compiled at run-time.

Perl, Python, and OCaml are languages where the programs are distributed as source code, to be compiled at run-time into an internal bytecode representation which is then interpreted. The Perl Compiler [7] allows three ways to make a pre-compiled Perl program: Writing out the internal representation (which reduces load time), writing out C source (which allows the use of any compiler to optimize it), and compiling directly to machine code. These are all done as separate invocations of Perl, rather than during regular execution. Psyco [69] is an extension of Python that compiles functions as part of the execution environment, gaining a speed-up of about a factor of 2.

Self [36] is a class-less variant of SmallTalk [28] that allows programming during program execution. Like Perl and Python, it compiles source into an internal representation at run-time. In order to improve performance, Self collects profiling data, and compiles the functions that are used the most. This pioneered the Just-In-Time compilation scheme that became widespread with Java.

The Java Virtual Machine was designed to allow both interpretation and run-time compilation ("just-in-time", or JIT compilation) of Java bytecode. Most modern JVMs include some form of compilation, and several JIT compilers perform optimization too. Optimizing JIT compilers include Sun's HotSpot[80],

LaTTe[92], IBM's Jalapeño[79], and OpenJIT[64]. While they face the same trade-offs between speed and code quality as run-time code generators, they are normally invisible to the programmer. However, the special characteristics of JIT compilation may become notable when attempting to optimize the program for speed, as can be seen in section section 4.2.

Quicksilver [73] takes the idea of JIT compilation further by storing persistent images of the result of the run-time compilation to use in future invocations. The use of persistence allows them to almost eliminate compilation time while retaining the speed benefits of an optimizing JIT compiler.

2.3 Java Optimization

In section 5.1, we consider how we may be able to optimize code fragments at an early stage. Some research has been done into optimizing Java Bytecode, including Cream[10], Soot[87] and JavaGO[46].

Cream uses side-effect analysis to drive well-known optimizations such as loop invariant removal and common subexpression elimination. The resulting speedup, even with whole-program analysis and liberal assumptions about native methods, is at best around 10%[2].

Soot makes extensive transformations using several internal representations (Baf, Jimple and Grimp). They show a speedup of 5–10% for several programs, and up to 60% on one example.

JavaGO performs whole-program analysis of the class structure and performs inlining of small, final methods. Their only example, matrix multiplication with setters and getters, is highly optimizeable, and yields a 100% speed increase for interpreted systems. There is no indication of how it performs with HotSpot or on more complicated programs.

Several "Java optimizer" systems focus on the size of the class files, removing unnecessary information and shortening method and field names to improve transmission and load speeds. The DashO system[67] additionally does some inlining of wrapper methods and marks non-overridden methods as final.

2.4 Conclusion

The last 15 years have seen a large amount of research into run-time code generation, aggressive optimizations and partial evaluation. Most of the research in partial evaluation has been done with the more

[2]Throughout the paper, we use "speed" to mean the inverse of "execution time". Thus, a 50% speedup is the same as a 33% reduction in execution time.

theoretically amenable functional languages, while Java in particular has fostered research into just-in-time compilation. There are some systems that allow programmers explicit control of the code generation, several of which are implemented for imperative languages.

Compile-time systems have been the object of most research in optimization and partial evaluation, as they do not need to consider the compilation cost as a part of their performance. However, run-time optimization and code generation allow the use of invariants and profiling data that may not be available at compile time, and so offer more opportunities for improving performance. The current high-level run-time code generators do not allow the creation of new types at run time. In essence, they allow value optimizations of various kinds, but not the creation of entirely new programs at run-time.

Jumbo, which will be presented in detail in the next chapter, is a novel run-time code generation system, featuring a simple syntax and straightforward implementation combined with the ability to be used in a distributed setting. It also includes the ability to create entirely new classes at run-time.

Chapter 3

An Overview of Jumbo

Jumbo is a distributed run-time compilation system for Java. It has been designed to allow code fragments to be defined in separate compilations, distributed as binary Java class files and eventually merged together to form efficient code. In this chapter, we first describe how to use the code manipulation aspects of Jumbo. We then describe the major components of the Jumbo system, after which we go into more detail on the combinatorial part that allows easy combination of code.

3.1 Using Jumbo

In this section, we describe the syntax and semantics of Jumbo's extensions to Java. The primary reason for constructing Jumbo is to have a system that allows easy run-time code generation. We have added a small syntactic extension that simplifies code generation, as well as a few functions that implement compilation and class loading.

To support programmer manipulation of code fragments, we have added quotation syntax reminiscent of that of Lisp[6]. The *quoted* expression `$<x+1>$` [1] is of type Code, and its value represents the expression `x+1`. The expression `$<x = x+1;>$` is also of type Code, but its value represents a statement rather than an expression. Code expressions can also represent run-time constants, switch branches, names, types, method definitions, field definitions, and class definitions.

In most cases, a quoted piece of code can be parsed on its own, but the user can also specify how to parse a quote. For instance, `$Type<MyName>$` indicates that `MyName` should be parsed as a type rather

[1] The dollar sign was chosen to avoid conflict with other syntax, and the less-than/greater-than signs make it clear where quoted code begins and ends, and to allow multiple levels of quotation.

than a variable. The `Type` indicator is necessary to disambiguate type names from other names, but the user can specify for any piece of code how it is to be parsed.

The other added syntax is *anti-quotation*, which allows the programmer to splice together code fragments. We use the antiquote (`‘`) operator to indicate anti-quotation. For instance, the code fragment `$<x*‘Expr($<y+1>$)>$` splices the inner expression `y+1` into the outer, yielding the same value as `$<x*(y+1)>$`. The antiquote is also said to introduce a *hole* into the quoted code, which can be filled with code from somewhere else. Any expression of type `Code` can be used to fill a hole. For instance, if the variable `c` is assigned the value `$<y+1>$`, then the expression `$<x*‘Expr(c)>$` has the same value as the self-contained expression above.

```
Code makeDot(int[] row, Code col) {
  Code sumcode = $<0>$;
  int len = row.length;
  for (int i = 0; i < len; i++) {
    if (row[i] != 0)
      sumcode = $<‘Expr(sumcode)+
        ‘Int(row[i])*‘Expr(col)[‘Int(i)]>$;
  }
  return sumcode;
}
makeDot(new int[]{10, 0, 3, 2}, $<W>$);
```

(a) Quoting code

```
0 +
10*W[0] +
3*W[2] +
2*W[3]
```

(b) Resulting code (decompiled)

Figure 3.1: An example of quotation: A dot-product expression generator

Figure 3.1 shows an example of how Jumbo can be used to generate specialized code. In this case, we generate a specialized dot-product expression for a particular vector. The `makeDot` method takes a static array of integers and a code fragment representing an array-valued expression. It iterates over the static array, using the values as multipliers in the dot product expression (starting with `$<0>$` in case the array is empty). It returns a piece of code specialized to calculate the dot product of the statically known array and the array returned by the given expression. The result is a Code value, as if the summation shown had been compiled to Code.

As a simple example of how optimizations can be added, we leave out any part of the computation where the value is known to be zero[2]. Other, more complex, optimizations are possible. For instance, if the

[2] Arithmetic simplification of semi-static expressions is not required of Java compilers, and has so far not been added to Jumbo.

Kind	Usage	Kind	Usage
Expr	Expressions	Int	Integer constant
Stmt	Statements	Char	Character constant
Name	Identifiers	Bool	Boolean constant
Type	Types	String	String constant
Case	List of case branches	Float	Float constant
Method	Method declaration	Long	Long constant
Field	Field declaration	Double	Double constant
Body	List of class members		

Table 3.1: Syntactic categories in Jumbo

static array was likely to contain repeated values, those could be collapsed into a single multiplication. Such domain-specific optimizations are generally beyond the scope of automatic specializers.

```
Code s = $<switch (x) {
             case 1: y = 1;
               ‘c
               z = 2;
          };>$;
```

Figure 3.2: A situation where an explicit syntactic category is necessary

The `Expr` tag after the antiquote indicates the syntactic category of the antiquoted expression: whether the surrounding code should consider the antiquote an expression, a statement, a method declaration etc. In a few cases, it is impossible to tell from context how to continue parsing after an antiquote, and having explicit categories also generally eases the job for the parser. For instance, figure 3.2 shows a piece of code that cannot be correctly parsed without knowing the syntactic category of `c`. If `c` is a statement, then it and `z = 2;` are statements within the first case, but if `c` is a case branch, it and the following statement form their own branch of the switch. Also, holes that are to be filled with literals rather than code fragments need to be marked with the category unless the type of the antiquoted expression can be determined at compile time. Currently, antiquotation without marking of syntactic categories can only be done for the Name category. As future work, careful tweaking of the parser may allow the syntactic categories to be omitted in most cases. Some cases, like the one described in figure 3.2, will always need to be explicitly marked.

Table 3.1 shows all the syntactic categories available. The categories in the right-hand columns “lift” values from the meta-level to the object-level. Thus, `$<‘Int(x)>$` uses the run-time contents of `x` to construct code representing an integer constant. The Name category takes a string, which may be used

for any identifier (variable, method name, class name, import name etc.). The Type category takes a type as created by `$Type<...>$` expressions, or as defined in the backend Jaemus, which is described below. The Case category takes a list of switch branches, i.e., `[$<case 1: case 2: break;>$, $<case 3: return;>$, ...]`. The remaining categories all take Code values. If no syntactic category is included, it defaults to Name. This default was chosen because abstracting variables is the most frequent use of antiquotation, and in many cases the object-level variable name is stored in a meta-level variable.

```
interface DotProduct {
  int dot(int[] V);
}
int[] V = new int[] {10, 0, 3, 2};
String classname = gensym("Dot");
Code c = $<class ‘classname implements DotProduct {
             public int dot(int[] W) {
               return ‘Expr(makeDot(V, $<W>$));
             }
         }>$;
DotProduct d = (DotProduct)c.create(classname);
d.dot(V);
```

Figure 3.3: Loading and using a generated class

Since Java only allows loading of full class files, all quoted code will eventually have to be put together as a class and compiled into a binary class file. The method `void c.generate()` causes the run-time compilation of the code objects. As an important side effect, all classes in the code object are written out as Java class files[3].

This leaves the problem of how to load and access the generated code. Since a code value may define more than one class, we need to specify which class to load. A call to `c.create(String name)` on a `Code` object will compile the code and output any class files, and then attempt to load the specified class using `Class.forName(String name)` and `Class.newInstance()`[4]. If successful, it returns an instance of the newly created class. If more instances are needed, a factory method defined in an interface or a call to `Class.newInstance()` can create them.

Once the class is loaded, we still need a way to access its methods. Since the Java verifier is required to verify the existence of all classes and methods used in a program, we cannot just use the names of the

[3]Java allows user-defined class loaders that can read the byte code out of a byte array. Experiments with storing the generated classes in byte arrays show no significant speed benefit.

[4]Java has no way to pass arguments to the constructors of dynamically loaded classes, so the class must either have a zero-argument constructor or provide a static factory method.

classes and methods generated, even if we know them. Instead, we have to make the externally accessible methods of a generated class implement an interface. Using an interface allows us to manipulate generated classes whose definition we do not know at compile time. Figure 3.3 shows an example of how to load and use generated code, using the `makeDot` function from figure 3.1. The class name is generated as a unique name so that multiple instances can coexist.

3.1.1 Variable Hygiene

When code from different modules or even different sources can be combined, it becomes difficult to ensure manually that distinct variables have distinct names. In particular, code generating methods that introduce temporary variables cannot be used recursively if they define variables with fixed names. In this section, we describe how Jumbo allows this problem to be handled

Jumbo has support for “hygienic variables”, i.e., variables with names that are guaranteed unique even when combined with other code. They are implemented in the the `Name` class, a subclass of `Code` which generates unique names during the compilation process.

```
Code saveVar(Type t, String name, Code body) {
  Name savespot = new Name("save");
  return $<final ‘Type(t) ‘savespot = ‘name;
           try { ‘Stmt(body) }
           finally { ‘name = ‘savespot; }
         >$;
}
```

Figure 3.4: Use of hygienic variables

The `Name` class is used as shown in figure 3.4. The `saveVar` method makes sure that the contents of a variable is stored during the evaluation of `body`, and is restored at the end of execution, even in the case of abnormal exit. This method is similar to `save-excursion` in Emacs Lisp. By using `Name`, we ensure that this variable can be used transparently, without worrying about the possible effects of nesting. Each usage has its own, unique name.

Variable hygiene is also useful for ensuring uniqueness of names in complicated systems, where it would otherwise be impractical for the programmer to keep track of names created in different methods or classes.

The hygienic variables are implemented in a slightly non-compositional fashion. By using overloading, variable definition for hygienic variables invokes a method on `Name` that generates a name for the variable.

This name is stored on the `Name` object for later retrieval. It is an error to try to define the same instance of a hygienic variable several times, or to use it without prior definition.

3.2 The Components of Jumbo

In this section, we describe the three parts comprising Jumbo. They form a layered system, where the lower layers do not use any knowledge of the upper layers, and could be used by other systems, if so desired.

The three parts are:

- The parser (written by Ava Jarvis). This part does lexing and parsing, using the grammar from the Java specification [29] with additions for quotation and antiquotation. The parser builds an AST, with quoted and antiquoted subtrees marked with specific nodes. The AST can either be output as source code text, or transformed into an intermediate representation (IR form).

  The IR form is similar to S-expressions [58], but exists only as objects, never as source text. It consists of a method name (the name of one of the combinators) and a list of arguments to the method. Using the Java Reflection API[81], the methods are called at compile time, creating the function objects that can perform compilation.

  Quoted code should not be created as objects yet, but as code that can create those objects at run time. To this purpose, quoted code in IR form is 'lifted' to create IR objects for method invocations that make the quoted code. For example, the code `$<x+1>$` would be lifted to the method invocations shown in figure 3.5. This lifting technique allows multiple levels of quotation, and makes the handling of quoted code almost trivial.

```
Constructor.binOp(6, Constructor.getQualifiedName("x"),
                  Constructor.integerConstant(1))
```

Figure 3.5: The lifted code for `$<x+1>$`

  In future versions, the AST and IR forms should be combined to allow lifting directly on the AST. Combining the two forms would also allow easier manipulation of the AST by the traversers in the rewriting system (see chapter 6).

- The combinators. The combinators are methods that create objects of type Code, as shown in figure 3.5. The Code objects are function objects that, when applied to an environment, create ClosedCode objects that consist of Jaemus code, a result type and an updated environment. The combinators handle type checking, allocation of virtual registers, method/field type lookup, and the conversion from AST level into instruction level. The combinators do not depend on the parser. This part of the compiler is almost entirely compositional.

- Jaemus, the Java byte code back end. Jaemus transforms bytecode instructions and class definitions into class files in the JVM format[54] and writes them either to disk or into a byte array. Jaemus handles the construction of the constant pool, the calculation of jump offsets, and the use of specialized bytecode instructions.

These three parts together form a standard Java compiler that generates code almost exactly like that of Jikes[9] or Javac.[5] Jumbo supports most of Java 1.2. Some features of inner classes, such as multiple levels of inner classes and certain ways of accessing outer variables, are not currently implemented. Additionally, Jumbo does not perform all the semantic tests required of a Java compiler. Given incorrect Java source code, it may produce Java class files that will not be accepted by the JVM class file verifier.

3.3 The Combinators

This section describes how the central part of Jumbo, the combinators, works. The combinators are higher-order functions that manipulate Code values in a compositional manner. [6] For instance, a function that implements addition could have the type Code $\times$ Code $\rightarrow$ Code. This function would take the left-hand and right-hand operators of the addition as arguments, and return an expression for the entire addition.

The Code type is central to our idea of a compositional semantics for code generation. The precise definition of the Code type depends entirely on the implementation. In our earlier implementations[40, 41], Code has been just strings of source code, or tuples of strings of assembly language and register allocation information. Given a reasonable definition of the Code type, we can define programs in terms of code combinators, functions without free variables that take Code values as arguments. We can change the definition

[5] Curiously, while the instructions are the same, code produced by Javac has faster method calls when run in interpreted mode on Sun's JVM. Since this effect does not appear when using the JIT compiler, it has little significance in practice.

[6] The standard notion of a combinator in functional languages is a function object with no free variables or side-effects. Such functions behave like (partial) mathematical functions, and are thus easier to reason about than functions with side effects.

```
Code whileLabel(final String label, final Code test, final Code body) {
  return new Code() {
    public ClosedCode eval(Environment env) {
      Instr startlabel = Instr.genNil("Start of while");
      Instr testlabel = Instr.genNil("Test of while");
      Instr endlabel = Instr.genNil("End of while");
      ClosedCode test1 = test.eval(env);
      Environment bodyenv = env.defineBreakLabel(label, endlabel);
      Environment bodyenv1 = bodyenv.defineContinueLabel(label, testlabel);
      ClosedCode body1 = body.eval(bodyenv1);
      return new ClosedCode(Instr.genGoto(testlabel)
                              .append(startlabel)
                              .append(body1.bytecode)
                              .addPop(body1.type)
                              .append(testlabel)
                              .append(test1.bytecode)
                              .addIf(EQ, startlabel)
                              .append(endlabel),
                            env, Type.void_type);
    }
  };
}
```

Figure 3.6: The combinator for the `while` statement

of Code to implement various ways to compile — source-based, bytecode-base, assembly-based or other systems.

In Jumbo, we see the Code type as a class that defines a method

```
ClosedCode eval(Environment env)
```

so Code = Environment → ClosedCode. The environment passed into the eval function contains definitions of variables, labels, classes, import statements and other information necessary for compilation. The resulting object contains a sequence of Java Bytecode instructions in the representation of the back end, an updated environment, the type of the result, and information about constant-valued variables. Thus, the type ClosedCode is actually Jaemus Code × Type × Environment × Constant Info.

Figure 3.6 shows a simple combinator, that of the labeled while statement. The outer function does nothing but return a Code value, with the salient part in the definition of the `eval` method. The three arguments to the combinator are required by Java to be declared final in order to be captured by the inner class.

The `eval` method first defines a trio of labels[7] that will be used to create the control flow in the while

[7]The Nil instruction is a Jaemus pseudo-instruction that does not appear in the class file.

loop. Secondly, it evaluates the Code object for the test. The resulting ClosedCode object is stored in `test1` for later use. Since no new definitions can be made inside the test, the environment returned in `test1` is subsequently ignored. The `bytecode` field of `test1` contains code that will evaluate to a boolean, to be used in the test.

Next, the environment is updated with information about break and continue targets. Since break and continue have different possible targets, the two are kept track of separately. The updated environment containing the break and continue labels is used for the evaluation of `body`. The environment used for `body` is discarded afterwards, as no variable declarations or labels are visible outside the while.

Finally, a new ClosedCode object is created, containing the instruction sequence of the while loop, the environment (unmodified, as the while statement defines a new scope from which no declarations can escape), and the result type, which is `void`. The bytecode sequence is created by appending together lists of Jaemus instructions. Additional instructions are added to jump over the body before the first test, to remove any value left on the stack by the body, and to perform the conditional jump based on the result of the test.

```
Code whileLabel(final String label, final Code test, final Code body) {
  return new Code() {
    public ClosedCode eval(Environment env) {
      Instr startlabel = Instr.genNil("Start of while");
      Instr testlabel = Instr.genNil("Test of while");
      Instr endlabel = Instr.genNil("End of while");
      ClosedCode test1 = test.evalWithLabels(env, startlabel,
                                             endlabel, endlabel);
      Environment bodyenv = env.defineBreakLabel(label, endlabel);
      Environment bodyenv1 = bodyenv.defineContinueLabel(label, testlabel);
      ClosedCode body1 = body.evalNoUse(bodyenv1);
      return new ClosedCode(Instr.genGoto(testlabel)
                              .append(startlabel)
                              .append(body1.bytecode)
                              .append(testlabel)
                              .append(test1.bytecode)
                              .append(endlabel),
                            env, Type.void_type);
    }
  };
}
```

Figure 3.7: The combinator for the `while` statement, using variants of the `eval` function.

Since we are using function objects (of class Code) rather than simply functions, we have the opportunity to define several functions on each function object. These functions can be used to perform local

optimizations. Figure 3.7 shows the while-combinator again, using optimizing function calls instead of just `eval`.

For the test part, we perform jump optimization [1] by passing three labels to `evalWithLabels`: The label to jump to on true values, the label to jump to on false values, and the next instruction after the test. Passing these labels allows the combinator for the test to use a compare-and-jump instruction instead of pushing a boolean value onto the stack which would immediately be consumed.

We also know that any value left on the stack by the last statement in the body is not going to be used. Since Java requires that an assignment expression evaluate to the assigned value, such extra values are quite common, and preserving them just to pop them immediately afterwards is suboptimal. The `evalNoUse` causes the expression to be evaluated for its side effects only, without leaving the result value on the stack.

These two optimizations are not required by the Java specifications, but they are the main differences between the naïve translation to bytecode and bytecode produced by `javac`.

This combinator is relatively simple. Other combinators have to handle things like constant propagation (as required in Java), field/variable distinctions and simple control flow analysis. The source file for the combinators currently has over 5000 lines.

Since Java, unlike C, allows forward references of fields, methods and classes, Jumbo uses multiple passes over the code to extract the required information. The `eval` method merely performs the last of four passes. Because of inner classes, each pass must go through all the code, so all combinators have a method definition for each pass. Each pass operates on an environment, though the first pass uses a slightly different environment.

An earlier version of Jumbo had one pass collect all classes, and the two next passes just operated on the collected classes. This setup did not give significant time savings, complicated the code, and did not allow for the correct handling of free variables and inner classes. The current version does a full traversal of the combinators for each pass.

The first pass extracts class names for all classes being defined. This pass allows us to generate the fully qualified class names and to add the classes to the class lookup table in Jaemus. Additionally, any free variables are collected and stored in the environment. The method used to make this pass is called `defineClasses`.

The second pass, performed using the `defineSupers` methods, sets up the superclass relationships

in the class table. Since a class can be defined before its superclass, these relationships cannot be found in the first pass, where the package name of the superclass may not yet be resolvable.

The third pass, performed using the `defineMembers` methods, defines the methods and fields. This pass also sets up the implicit constructors and super-constructor calls required by Java. While in most cases the super constructor call is a zero-argument call, if the super class is an inner class in the same class as the current class (i.e., a sibling class), the super constructor call must pass a reference to the outer class. Thus, super constructor calls must be made after the second pass. It might be possible to combine the second and third passes, since most super-constructor calls do not require knowing the name of the super class.

Capture of free variables in inner classes in Java is done by passing the free variables to the constructor of the inner class and storing them in private fields. The information on free variables collected in the first pass is used in this pass to add the fields and constructor arguments that handle the free variables. Additionally, any initializers of fields are collected here, so they can be added to the constructors in the next pass.

The final pass is the `eval` pass, in which actual Jaemus code is created, as described above. Additionally, during this pass every class definition calls `writeClassFile` to generate a class file from the entry in the Jaemus class lookup table.

3.4 Compositionality

The compositionality of the central part of Jumbo is an essential feature of the design of Jumbo. Suppose C : Program $\rightarrow$ Code is the basic semantic function of the language. Compositionality implies that one can compile programs with holes, because it means that the following function is well-defined: $C_H(P)$: Program-with-hole $\rightarrow$ Code $\rightarrow$ Code $=$ fn $P[] \Rightarrow$ fn $a \Rightarrow C(P[A])$, where A is any fragment such that $C(A) = a$.

The compositionality makes it trivial to combine separate pieces of code. Consider again the function of type Code $\times$ Code $\rightarrow$ Code that implements addition. Either of the operators could be a literal piece of code, or it could be some other expression of type Code, like a variable or a method call. The function itself cannot ask whether the operators are given as literal or as something else, only what their values are.

Another benefit of a compositional compiler over source- or AST-based system is that the program is defined in terms of code-generating code. This allows us to optimize and specialize the code generators as we would any other program, rather than having to create Jumbo-specific optimizers. Essentially, we

encode the meaning of the program by using combinators, whereas an AST does not have meaning until it is converted to some other form. An optimizer working on an AST must predict how the AST will be used before it can perform significant optimizations on the AST nodes.

Jumbo needs no support for combining code except in the parser and AST, where quotation, antiquotation and lifting are defined. Additionally, compositionality makes it easier to debug the compiler, as there are fewer side effects that can cause hard-to-trace errors. Finally, compositionality is the very thing that allows us to perform optimizations on the code generator. Since the meaning of an element is a function of the meanings of the sub-elements, we can safely perform code transformations that would otherwise be blocked by side effects.

While compositionality is important for the part that combines code, there are places where it would be more cumbersome than helpful. The parser and AST are not compositional. The parser uses JavaCUP and JFlex, separate programs not built for compositionality. The AST performs certain operations on its nodes that would be impractical if compositionality were enforced or that make for a cleaner interface with the combinators. These transformations include decoding array declarations like `int[] a[]` (a two-dimensional array), adding default super constructor calls to constructor bodies, and managing the types of array initializers like `new int[] {{},{}}`. These transformations impose some extra constraints on what can be used in a hole, but only for unusual cases.

Both the parser and the AST are used before the code distribution phase, and therefore do not need to be compositional to allow combination. The backend Jaemus is not compositional either. Again, no combination of code happens at this stage, so compositionality here is not necessary. Furthermore, the handling of constants (currently kept in a hash table) would be significantly slower in a compositional implementation.

The middle part of the compiler — the combinators — handles type checks, pseudo-register allocation, constant propagation and conversion from an AST-like form into the assembly-like form of Jaemus. This part is where code can be combined, and the part where compositionality is needed. The combinators are function objects passing information between them in an almost purely compositional environment. The result of a combinator additionally contains some Jaemus code, a result type and some constant propagation information.

We have chosen to make one part of the environment non-compositional: The class lookup table. Class

lookup is an expensive operation in Java, even if the class is already loaded into the virtual machine. Minimizing the number of class lookups reduced the code generation time by almost an order of magnitude for some examples. Since a class once loaded is the same for the remainder of the execution, and classes have unique fully qualified names, non-compositional class loading does not interfere with the compositionality of the rest of the compiler. To an external viewer, the class lookup table has all classes loaded from the start, but internally it operates in a lazy fashion.

Compositionality can to a certain degree be enforced by use of the `final` keyword in Java. A field marked `final` must be assigned in the constructor or it will have the default value for its type. It cannot be changed thereafter, thus the field behaves like an element in a tuple. Marking a field final has the advantage of static enforcement of the tupleness of the field. The disadvantage is that making updated copies requires explicit copying of each field of the object. For instance, when adding a variable to the environment, we want to make a new environment from the old one, with just the list of variables updated. The `clone()` method, which implements a fast, shallow copy, doesn't allow us to set final fields in the new object. Instead, each field must be extracted from the old tuple and assigned in the constructor, a time-consuming operation.

3.4.1 Code Generation Using Templates

For the final code generation, Jumbo creates a piece of byte code for each combinator at run time, piecing them together to form larger chunks and finally whole programs. Some code generating systems use a static compiler to create chunks of low-level code that can be copied together to generate code quickly. In this section, we consider whether using templates would be a viable option for a system like Jumbo.

Template-based code generation has been used successfully in a number of projects, including Tempo[63], BCS[56], DyC[31], and DynJava[65]. In template-based code generation, static compilation produces a number of fragments of low-level code, which are then copied together as part of the code generation process, patching loop indices, variables and other adjustable items appropriately. It has the advantage over regular code generation that fixed code chunks can be generated very quickly, but it requires correct handling of all parts that cannot be determined at static compile time.

Templates might be a way to speed up code generation for Jumbo, but only if there are enough parts of quoted code that are totally static. Elements that could break the staticness include: Undefined variables, lookup of fields and methods not yet defined at compile time, variable indices, control flow changes, and

```
$<int y = 3;
  ‘z = foo(x*3+‘z, x+y);
>$
```

Figure 3.8: Some of the obstacles to template-based code generation: Unknown types of holes, non-hygienic variables, and overloaded operators and methods.

missing type information in overloading resolution.

Figure 3.8 shows some of the ways these problems can appear. Here, `x` is a non-hygienic variable, so its type is unknown. `z` is a hole, whose type is also unknown. While `x*3` must be a numeric value, the lack of a type for `z` means that the addition can be either numeric addition or string concatenation. The method lookup for `foo` depends on the types of its arguments, but both arguments here have undecided types. If `foo` cannot be resolved, the type of `z` cannot be inferred from the return type either.

Control flow changes and variable indices must necessarily be handled by backpatching. Field/method lookup and overloading resolution all have three parts to them: Possible conversion of the operand types, selection of the correct operator, and determination of the result type. For most operators, including method references, the actual operator can be backpatched. Operand type conversion will be required when method or field lookup requires widening, e.g. from int to double, and that can only be determined when the method signature is known. For non-hygienic variables, the load and store instructions must also be adjusted at run time to match the type. The type of the return value is required to either pop the value off the stack or it will be needed in further evaluation.

Strings pose a special problem, as the plus operator is overloaded to mean string concatenation whenever one of the operands is a string. This problem could be handled as a special case, as numerical addition is more likely to be found in generated code. String concatenations also follow a strict pattern, so a generic template for string concatenation generation could be built in. The overloaded numeric types are less of a problem, as they only require a change in operator rather than an entirely different calculation.

Type inference may be an aid in determining possible types. If a non-hygienic variable has been used in a numeric context earlier in the same code fragment, it will certainly not later turn out to be a string, and the string case can be avoided. Other restrictions on the possible types may be inferred from assignment expressions and method argument passing. Since these type inferences can happen at compile time, we can afford to be as exact as possible. In the example in figure 3.8, we can guarantee that `x` is a numeric type, and so must `x+y` be. This limits the lookup possibilities for `foo`. If `foo` can now be resolved, the possible

types of `z` would be known, allowing us to resolve `x*3+'z` to either numeric or string addition.

Java bytecode contains a number of optimized instructions for loading the lowest-index variables and pushing frequently used constants. A backpatcher would not be able to use these optimized instructions to increase performance and reduce code size. However, they may make little difference once the bytecode has been compiled in the run-time system.

While some types may be determined at compile time, run-time generated types obviously cannot be determined in advance, and so will cause further calculations to operate with unknown types. For applications that use Jumbo's unique ability to create new types at run time, template-based code generation would probably be of little benefit.

Using the system call `System.arrayCopy()`, copying fixed chunks of code could conceivably be performed at speeds approaching one bytecode instruction per instruction generated. However, code generator startup costs would be high, as arrays in Java cannot be stored as static data, but must be created as part of the class initialization process at the earliest.

While template-based code generation has proven effective for other run-time code generation systems, the two unique features of Jumbo, compositionality and creation of new types at run time, make templates a less appealing option than would appear at first. Using templates would also require a radical redesign of Jumbo, as parts of code generation would be performed at compile time.

3.5 Conclusion

In this chapter, we have introduced Jumbo, a run-time code generation system for Java. Jumbo compiles a version of Java 1.2 extended with code objects, although some aspects of Java are not implemented. Code is represented as first-class objects, using a compositional semantics for Java, making it easy to perform code composition. Two new syntactic elements — quotation and anti-quotation — allow simple creation, manipulation and combination of code objects. The next chapter will show some extended examples of the use of Jumbo, along with measurements of their performance.

Chapter 4

Examples

To evaluate the usability and performance of Jumbo, we have implemented and run several examples of code-generating programs. In this chapter, we describe how some possible uses of code generation can be implemented using Jumbo, and, where appropriate, we compare the performance of the code-generating version with equivalent non-generative implementations.

A run-time code generation system like Jumbo has many potential applications. The most obvious are the examples that use run-time values to optimize a subsequent computations (so-called "value-based optimization"). Other applications are not so much for speed as for programmer convenience or for hiding data. In this chapter, we describe the following applications:

Dot Product: This simple example, also used in other chapters, shows how code generation can be used for data specialization.

Convolution matrices: This example is taken from the DyC paper[31], and shows a case where a significant speedup can be had for realistic problems using data specialization.

Product Line Architectures: This example shows a simple implementation of a technique for systematic selection of features in program design.

Anonymous database queries: In this example, not only do we get a speedup, but we also show how the binary nature of code fragments can be used to protect privacy.

Numerical algorithms: This example shows both how Jumbo can be used for efficient implementation of domain-specific languages, and how loop fusion can be easily implemented with Jumbo.

The final section mentions some other possible applications, but we believe the possibilities are literally endless.

4.1 Measuring Efficiency of Code Generation

The two main indicators of performance are the crossover point, where the increased speed of the generated code pays for the cost of generating it, and the asymptotic speed difference. However, the exact cost of generating code depends on what we count.

Any use of Jumbo entails the following steps, though possibly interleaved rather than being strictly in this order:

1. Load Jumbo compiler API.
2. Execute code-combining parts of the program producing `Code` value.
3. Compile `Code` value.
4. Load and verify generated classes.
5. Execute generated classes.

When doing a single run of the examples, the cost of loading the compiler classes usually dominates the cost of compilation by a factor of 10. The classes comprising the compiler currently total 860 kilobytes of class files, though this number could be reduced by running them through class file compressors. Loading the classes takes much longer than actually compiling all but the largest programs. Fortunately, it only has to be done once during a Java session. In a situation where one program runs continually to generate several code pieces, this startup time may not be relevant to the overall performance of the system.

The cost of loading the Jumbo compiler API could also be ameliorated by making Jumbo a part of the run-time system rather than stand-alone classes, in which case the verification of the Jumbo classes could be avoided. Additionally, the classes could be stored in a format more suitable for the particular run-time system, and might be subject to off-line optimizations. This idea could be carried further, to the point where the Jumbo system is completely integrated with the JVM. We consider this idea further in section 7.3.

After generating and compiling a new class, it must be loaded into the JVM. The built-in code verifier in the JVM is responsible for at least half the time spent loading a class. If we have a trusted system,

we can turn off the verifier completely. However, when combining components from potentially diverse sources we cannot normally assume such trust. Some applications, such as domain-specific languages, may be considered trustworthy enough to disable the verifier, which would reduce the cost of compilation significantly.

We ran all tests on an otherwise idle AMD Athlon XP1700+ machine with 1 GB of memory running Debian GNU/Linux on kernel 2.4.18. The programs were compiled using Blackdown Java version 1.4.1. All times are wall-clock time in microseconds, measured with a native interface to the `gettimeofday()` function[1].

4.2 HotSpot

Measuring the performance of our examples was complicated by the effects of the HotSpot compiler[80, 60] in the Java Virtual Machine. This section explains how the run-time optimizations of HotSpot interact with measuring execution speed. Unfortunately, no in-depth description of the workings of HotSpot was available, so we can only guess at the reasons for the effects we measure.

HotSpot is the just-in-time compiler central to Sun's Java Virtual Machine. HotSpot not only generates native code, but also performs profiling and applies optimizations in the most heavily used areas of the program. By only optimizing the "hot spots" of the program, HotSpot gets more speed increase for the time spent optimizing than if it had optimized the whole program. The hope is that this additional speed increase pays for the cost of the instrumentation necessary to figure out the hot spots.

The JVM has three different settings for optimization: Interpreted, Client and Server. In interpreted mode, no translation into native code is done, nor any optimizations. In client mode (the default), two levels of optimization are applied, but only methods below a certain size are optimized. In server mode (intended to be used for long-running programs on servers), aggressive optimizations are applied in a more fine-grained manner, making the initial performance difficult to predict, but achieving significantly better performance later in the execution. In server mode, the size of a method still places some limitations on how many optimizations are applied to the method.

Earlier versions of HotSpot could only replace a method with an optimized version between calls to

[1] Java by default does not support measuring CPU time (or even microsecond time), and since the JVM runs the optimizer and garbage collector in separate threads, CPU time would not accurately reflect the actual execution time.

the method. In such a situation, aggressive inlining would lead to decreased performance, as the inlined code was less likely to be optimized. In the recent versions, HotSpot can optimize the currently executing method after a set amount of time has been spent executing it and replace the method during execution. This "on-stack replacement" is only done when there is a backwards branch in the code, so unrolled code and other loop-less methods are less likely to be optimized.

Despite the changes in version 1.4, inlining done by HotSpot still appears to be more efficient than hand-inlining, to the point where "outlining", the opposite of inlining, has been observed to increase performance. There also seems to be a threshold for how early in a session optimizations will be applied. Thus, methods executed early in a session have different optimization characteristics than if they were executed later. The Sun JVM offers no control over the parameters of HotSpot.

While HotSpot is beneficial to performance in most cases, it also makes it difficult to measure the effect of other optimizations. When run-time optimizations are applied in an unpredictable manner, the effect of earlier optimizations can be obscured. In some cases, obvious optimizations at compile-time (such as method inlining) can decrease performance, not only because the method size may grow beyond the limits of optimizations, but also because the run-time optimizer may decide to do optimizations earlier when encountering a number of small methods instead of one medium-sized one.

Figure 4.1 shows the execution time of the simple generated function for evaluating dot products first shown in section 3.1. To better show the changes over a large expanse of time and code size, all the axes are logarithmic. Given a static vector V, we generate a method to multiply V by another vector. As $|V|$ grows, so does the size of the method generated. In the graph, the x-axis indicates the size of the method, the y-axis indicates the number of times the method is called, and the z-axis (vertical) shows the time spent on execution. The graph clearly shows the two features mentioned above. On the left side of the graph, the two small jumps show where the two phases of the optimizer appear. Most likely, the first phase is simple compilation to native code, while the second phase is actual optimization. In each case, the optimizer at first adds extra execution time (it spends time optimizing code that will not be executed for long), but it flattens out to a lower level than before. On the right side is a flat plateau, indicating that the method is so large that HotSpot refuses to optimize it.

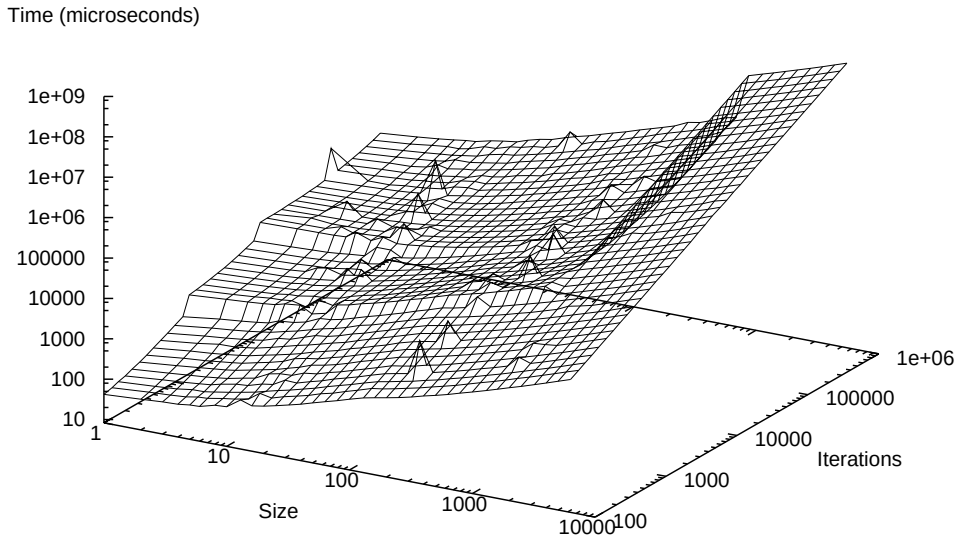

Figure 4.1: Execution time of dot-product

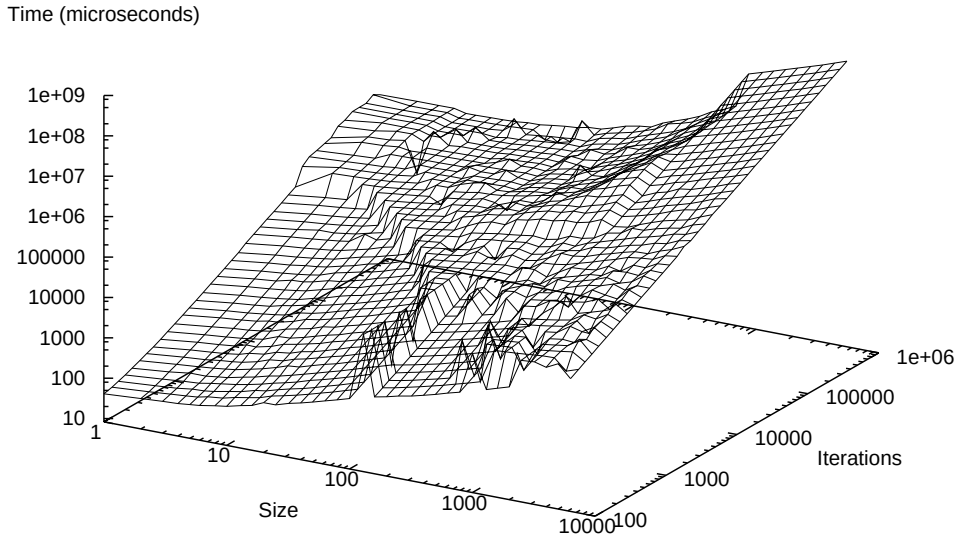

Figure 4.2: Execution time of dot-product using `-server` option

Figure 4.2 shows the execution time for the same function, but with the `-server` option for the JVM enabled, again with logarithmic axes. The situation is even more complicated in this case. The two jumps of optimization are no longer straight, but instead curve in a hyperbolic fashion. This curvature seems to indicate that execution time, rather than the number of invocations, determines the onset of optimizations. At a certain level, HotSpot apparently attempts optimization at once, then stops the attempts for slightly larger methods, leading to the sharp ridge at size 100. There seem to be some attempts at optimizing even very large functions, though it has little actual effect when we get beyond a thousand iterations.

The timings in this chapter are all done with the `-client` option. It is the default setting, and thus more commonly used. Repeated tests have shown that the `-server` option also causes significant variations in the running time for short runs. Differences of an order of magnitude are frequent, and in some cases two orders of magnitude difference were found.

When measuring the execution time of native code, it is standard practice to discard the first few runs to stabilize the cache. With HotSpot, discarding the first runs may be what pushes us into the optimizing area. If we want to have an unbiased measurement of the speed, we should either make sure that each variant runs on its own, with little preceding execution, or discard enough runs that we are sure the same amount of optimization is applied to all parts. For the latter, it may be problematic to get an even amount of optimization, as methods may be separately optimized based on how much they are used. A system with many methods will not exhibit the clear separation of optimized and unoptimized areas, as HotSpot may optimize just one small method at a time.

Which of the two above measuring methods gives the more accurate estimate of the effect of code generation depends entirely on the eventual application. For a program that tends to run frequently but briefly, the first method will depict the actual behavior better. For a program that runs for a long time (such as a server), the second method is preferable.

To give an idea of how the system would behave in either situation, the examples below show both cross-over point (the number of iterations necessary to break even) and asymptotic speed increase. The measurements are done with a minimum of prior execution, apart from loading the compiler classes. The asymptotic behavior will show how the system performs when run for an extended period of time.

```
public static Code makeSumCode(int[] row, String colname) {
  Code sumcode = $<0>$;
  for (int i = 0; i < row.length; i++) {
    if (row[i] != 0)
      sumcode = $<‘Expr(sumcode)+‘Int(row[i])*‘colname[‘Int(i)]>$;
  }
  return sumcode;
}
public static Dot codegenDot(int[] V1) {
  String codegenName = "VectorDot_"+V1.length;
  Code c = $<public class ‘codegenName implements Dot {
    public int dot(int[] V2) {
      int product = ‘Expr(makeSumCode(V1, "V2"));
      return product;
    }
  }>$;
  return (Dot)c.create(codegenName);
}
```

Figure 4.3: A code generator for dot products

4.3 Code-Generation Examples

The examples below show show not only that generated code is more efficient, but also that our syntax for code quotation is easy to work with. In most cases, the syntax in the quoted code is very similar to the syntax of the equivalent unstaged program.

4.3.1 Dot Product

Calculating the dot product is a central part of matrix operations. Matrix multiplication in particular involves many dot products. To multiply two $n \times n$ matrices, n^2 dot products are required.[2] One way to improve the speed of matrix multiplication is to specialize the dot product function. This specialization has the most effect for sparse matrices, where a specialized version can skip any zero element, though matrices with other characteristics amenable to optimization could also see an effect.

Figure 4.3 shows how to generate specialized dot product calculators. We create a new class that contains the specialized method. The body of the method is created by building a single expression with all the non-zero calculations of the dot product. Using this expression not only removes the overhead of the loop and

[2]Divide-and-conquer methods have improved on this, to the point where less than $n^{2.376}$ multiplications are required[14], but the overhead involved gets progressively more prohibitive.

Vector size	Contents	Cross-over point	Relative speed
10	90% zeroes	30000	6.40
100	90% zeroes	5500	15.76
1000	90% zeroes	2300	12.45
10	random	50000	2.00
100	random	100000	1.26
1000	random	none	.28

Table 4.1: Cross-over points and asymptotic relative speed for DotProduct

extra assignments, but also the array lookups for the static array. The `Dot` interface contains one method, `int dot(int[] v)`, that multiplies the vector `v` with a fixed vector.

Figure 4.4 compares running times of the normal and the code-generating dot product functions, for sparse vectors (90% zeroes) with respectively 10, 100 and 1000 elements. The time spent loading the compiler classes is not included. The break in the curve up to 1000 microseconds for the normal code is where the JIT compiler starts compiling the code. The cross-over point is visible in all three cases.

Table 4.1 shows the cross-over points and the asymptotic relative speed for both sparse and dense vectors. The speed loss for 1000-element dense vectors is due to the generated code becoming too large to be optimized.

At a vector size of about 2000, the cross-over point for sparse vectors comes after as many iterations as there are entries in the vector. At this point, it might be possible to increase the speed of matrix multiplications by using code generation. The straight-forward method of multiplying matrices dots each row in one matrix with every column in the other matrix. For square matrices, this method means as many iterations as there are elements in each row, and thus matrices of size 2000 by 2000 or larger should benefit from using code generation. Experiments with this style of matrix multiplication has failed to reach the cross-over point, as the sheer number of classes loaded used too much memory. However, non-square matrices could reach the cross-over point earlier.

While the specialized dot-product code does not yield a faster matrix multiplication, it does give significant speedups especially for sparse vectors. For vectors with 90% zeroes, the crossover point is low enough to be reached in realistic applications, while for dense vectors, the speed increase is modest and only reached after tens of thousands of iterations with small vectors. For vectors of size 1000 or more, the code-generating version is significantly slower than the straight-forward version. This slowdown points out

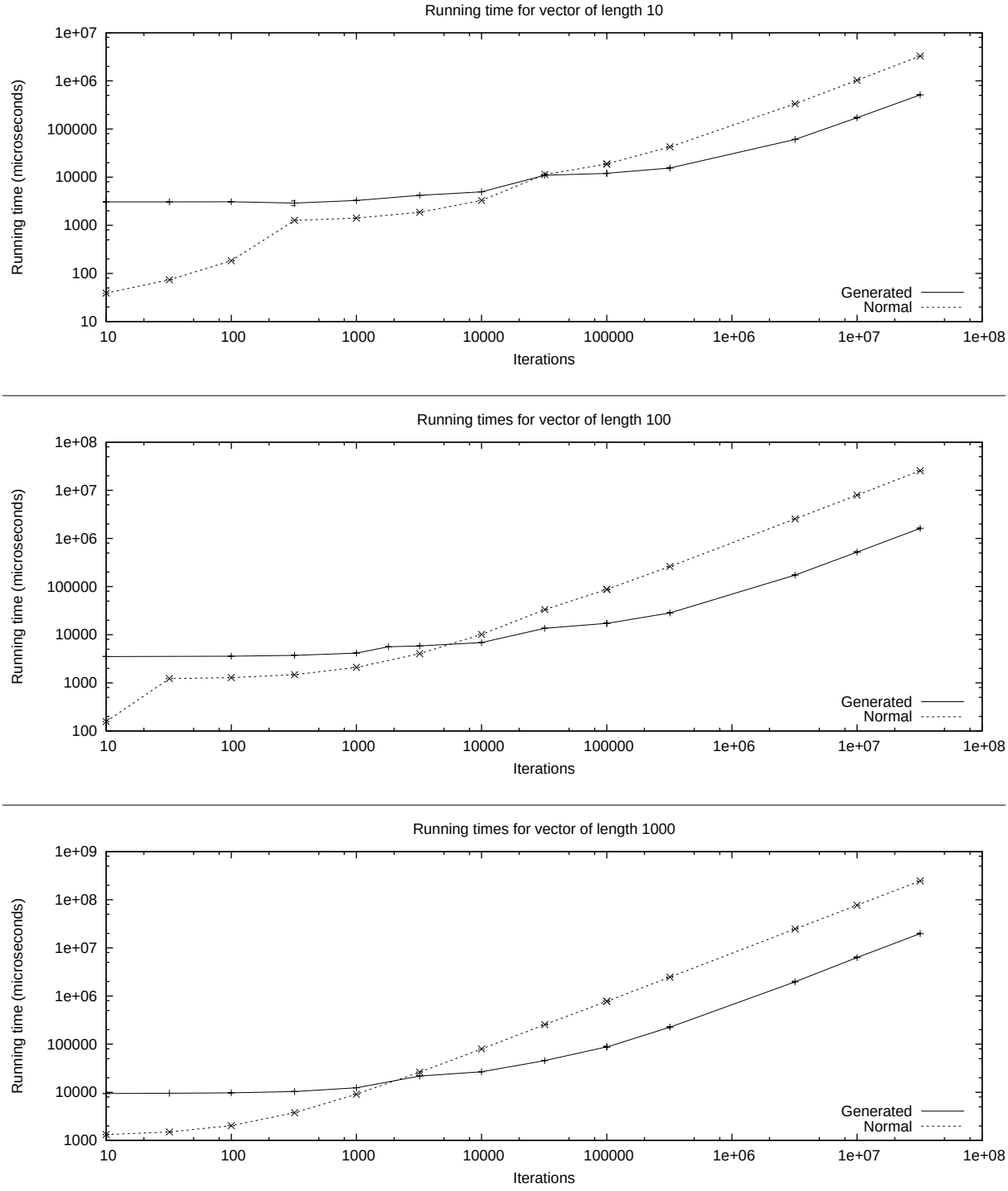

Figure 4.4: Running times for normal and code-generating versions of dot product (90% zeroes, log scale)

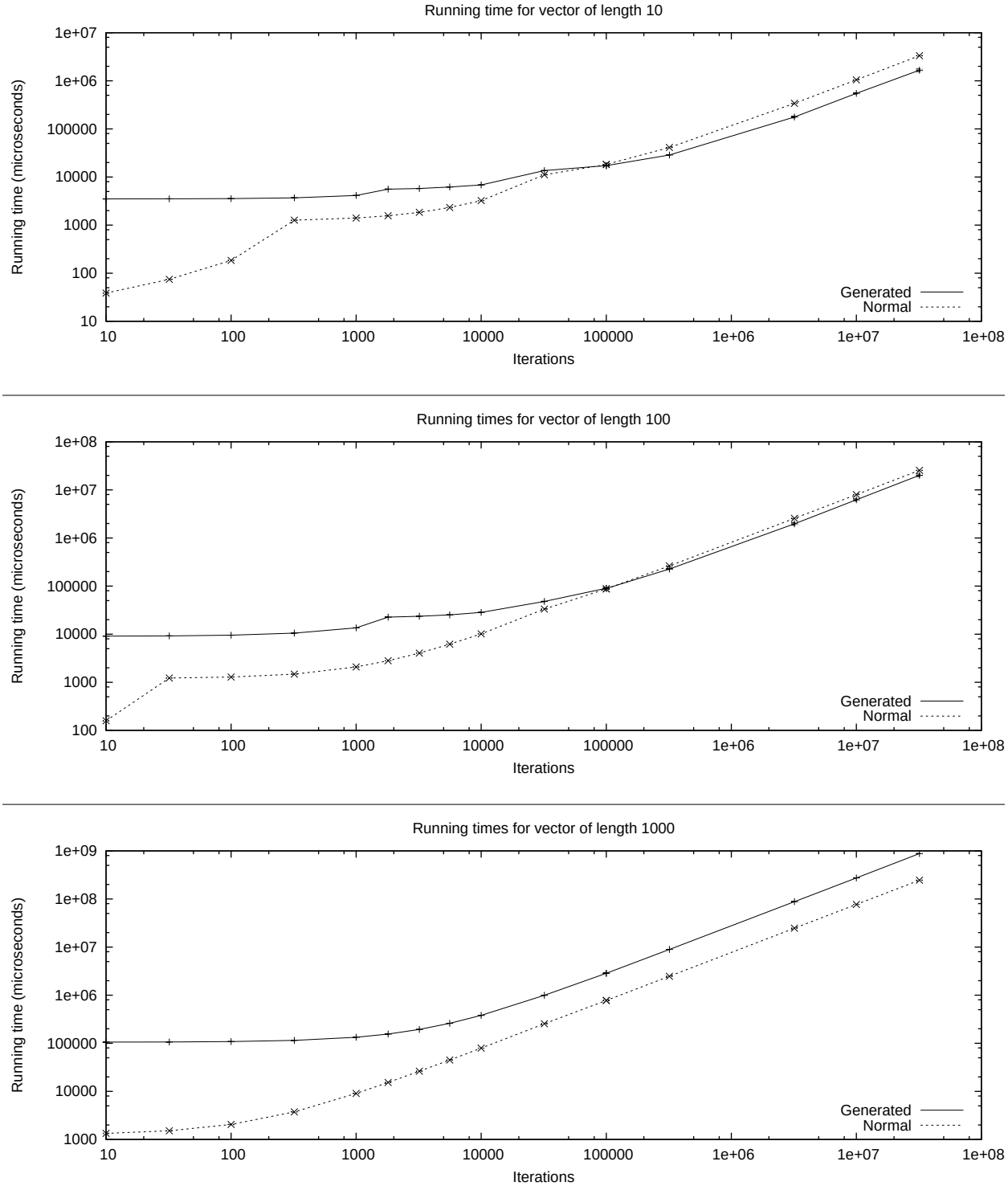

Figure 4.5: Running times for normal and code-generating versions of dot product (log scale)

that code generation should be used carefully — it is not a panacea that automatically speeds up code, but used under the correct circumstances, it can yield significant improvements.

4.3.2 Convolution Matrices

Convolution matrices[25] are a generic way to manipulate images. Given a matrix of weights, each pixel value is determined by weighting the surrounding pixels according to the matrix. This method can implement a number of different effects, including blurring, edge detection, sharpening, embossing and much more.

In the DyC paper[31], Grant et al. give an example of how matrix convolution can be specialized. They get an asymptotic speedup of a factor of three on an 11x11 matrix with 83% zeros, with the cross-over point reached after only one iteration.

Figure 4.6 shows a straightforward implementation of the convolution matrix filter. It applies a square convolution matrix to an array of `Pixel` values.[3] For simplicity, we omit the handling of the edges of the image. The outermost two loops loop over the image, while the two inner loops loop over the convolution matrix. For each element of the matrix, we add the scaled values of the pixel to the sum, which afterwards is assigned to the output pixels.

The matrix is static for the entire image, and in real-world applications most likely would be used for a number of pictures. Thus `matrix`, `cols`, `i`, `j`, `base`, and `scale` contain static values for a given matrix. We could also consider `width` and `height` to be static for a given image size, but it would make little difference in our system, and would require new code generation to handle different sized pictures.

The code-generating version using Jumbo is shown in figure 4.7. We unroll the matrix loop completely to remove the loop overhead, and to be able to extract the values from the matrix array. At the same time, we take advantage of the values in the matrix being static to avoid doing any calculation for the zero entries, and avoid a multiplication for the one entries. Notice that while we could have given an antiquoted value for `cols/2`, it is being handled by Jumbo itself due to the required constant propagation of Java.

The code generating version is remarkably similar to the original version. The innermost loops were moved to a separate function because expressions cannot contain loops. Other than that, it is a direct transla-

[3] It turned out to be faster to use pixel objects than a flat array of integers, as long as the allocated objects are reused. The `getPixelArray` and `forgetPixelArray` functions handle recycling of pixel arrays.

```
class Pixel { int red, green, blue; }
class ConvolutionFilter extends SizePreservingFilter {
  double[][] matrix;
  public ConvolutionFilter(double[][] m) { matrix = m; }
  public Pixel[] filter(Pixel[] src, int width, int height) {
    final int cols = matrix.length;
    Pixel[] output = Convolution.getPixelArray(src.length);
    for (int x = cols/2; x < width-cols/2; x++) {
      for (int y = cols/2; y < height-cols/2; y++) {
        int sum_r = 0, sum_g = 0, sum_b = 0;
        final int base = ((y-cols/2)*width)+x-cols/2;
        for (int i = 0; i < cols; i++) {
          for (int j = 0; j < cols; j++) {
            final double scale = matrix[i][j];
            if (scale != 0.0) {
              final int offset = base+i*width+j;
              sum_r += (int)src[offset].red*scale;
              sum_g += (int)src[offset].green*scale;
              sum_b += (int)src[offset].blue*scale;
            }
          }
        }
        output[y*width+x].red = sum_r;
        output[y*width+x].green = sum_g;
        output[y*width+x].blue = sum_b;
      }
    }
    Convolution.forgetPixelArray(src);
    return output;
  }
}
```

Figure 4.6: Original matrix convolution implementation

tion of the original version with but a few annotations of syntactical categories and some care put into which things to quote and which not to quote.

Table 4.2 shows the speed increase and crossover points of the code generating version versus two normal versions, one straight-forward and one that avoids multiplying by zero[4]. Five different matrices were used: “solid” is a 9x9 matrix with 20% zeroes, “blur” is a 3x3 matrix with 4 of 9 entries being zero, “relief” is a 5x5 matrix with 40% zeroes, “streak” is a 5x5 matrix with 80% zeroes, and “dyc” is a matrix similar to that used in the DyC paper[31], an 11x11 matrix with 83% zeroes. The values used are the averages of five separate runs. Image loading and format conversions are not included in the results shown, but the time to generate and load the new class is.

[4]While the simple optimization of not multiplying by zero may seem obvious, the Gimp[57], for instance, does not employ it.

```
Code makeConvolveIter(double[][] m) {
  final int cols = m.length;
  Code convolve = $<;>$;
  for (int i = 0; i < cols; i++) {
    for (int j = 0; j < cols; j++) {
      final double scale = m[i][j];
      if (scale != 0.0)
        convolve = $<‘Stmt(convolve) {
           final int offset = base+‘Int(i)*width+‘Int(j);
          sum_r += (int)src[offset].red*‘Double(scale);
          sum_g += (int)src[offset].green*‘Double(scale);
          sum_b += (int)src[offset].blue*‘Double(scale);
        } >$;
    }
  }
  System.out.println(convolve);
  return convolve;
}

public SizePreservingFilter convolutionGen(double[][] m) {
  Code c = $<
    public class ConvolutionGen extends SizePreservingFilter {
      public ConvolutionGen() {}
      public Pixel[] filter(Pixel[] src, int width, int height) {
        final int cols = ‘Int(m.length);
        Pixel[] output = Convolution.getPixelArray(src.length);
        for (int x = cols/2; x < width-cols/2; x++) {
          for (int y = cols/2; y < height-cols/2; y++) {
            int sum_r = 0, sum_g = 0, sum_b = 0;
            final int base = ((y-cols/2)*width)+x-cols/2;
            ‘Stmt(makeConvolveIter(m));
            output[y*width+x].red = sum_r;
            output[y*width+x].green = sum_g;
            output[y*width+x].blue = sum_b;
          }
        }
        Convolution.forgetPixelArray(src);
        return output;
      }
    } >$;
  return (SizePreservingFilter)c.create("ConvolutionGen");
}
```

Figure 4.7: Code-generating matrix convolution implementation

```
{
  final int offset = ((base+(1*width))+2);
  sum_r += (((int)src[offset].red)*0.1);
  sum_g += (((int)src[offset].green)*0.1);
  sum_b += (((int)src[offset].blue)*0.1);
}
{
  final int offset = ((base+(2*width))+1);
  sum_r += (((int)src[offset].red)*0.1);
  sum_g += (((int)src[offset].green)*0.1);
  sum_b += (((int)src[offset].blue)*0.1);
}
...
```

Figure 4.8: Unrolled matrix loops

Image size	Matrix	Relative speed	Crossover point
100x100	solid	1.68	21
100x100	blur	1.97	25
100x100	relief	2.03	11
100x100	streak	4.83	7
100x100	dyc	6.97	2
320x217	solid	1.68	4
320x217	blur	1.87	3
320x217	relief	1.98	3
320x217	streak	4.69	1
320x217	dyc	6.86	1
640x435	solid	1.64	1
640x435	blur	1.69	1
640x435	relief	1.81	1
640x435	streak	3.05	1
640x435	dyc	5.88	1

Picture size	Matrix	Relative speed	Crossover point
100x100	solid	1.41	35
100x100	blur	1.39	63
100x100	relief	1.47	24
100x100	streak	1.84	30
100x100	dyc	2.17	10
320x217	solid	1.41	7
320x217	blur	1.35	6
320x217	relief	1.46	5
320x217	streak	1.90	3
320x217	dyc	2.17	2
640x435	solid	1.41	2
640x435	blur	1.38	1
640x435	relief	1.42	1
640x435	streak	1.63	1
640x435	dyc	2.05	1

Table 4.2: Speedup and crossover points of code-generating convolution compared with naïve and zero-avoiding implementations

All three programs reached a stable speed after only three iterations, so we stopped iterating at ten iterations. Crossover points beyond that are extrapolated.

The speedup is modest for the dense matrices and somewhat better for the sparse matrices. Against the naïve version that does not avoid multiplying with 0, we see a five-fold speed increase for the largest pictures. The zero-avoiding version fares better, only being about half as fast as Jumbo on the sparse matrices. The time spent generating code, however, is recouped within a single iterations for pictures of just one-quarter of a megapixel. Given that most digital cameras today create pictures of several megapixels, picture processing is a field with excellent opportunities for benefiting from code generation.

4.3.3 Product-Line Architectures

Batory advocates the idea of "product-line architectures" as a way to handle the increasing complexity of software[5]. In product-line architectures, the building blocks for programs are the desired features. These building blocks can be combined to automatically form variants of a complex system.

Batory demonstrates how software libraries suffer from a combinatorial explosion because of the many possible combinations of largely orthogonal features. A library implementing a deque has a number of different features to choose from: Are the operations synchronized, does the deque implement the generic "container" interface, do the deque elements form a doubly-linked list, are the deque elements reused in a free list, etc. A library attempting to implement all combinations of features would grow to unmanageable size, yet any combination of features may be of use to some programmer.

Batory's solution to this problem is to allow specific combinations to be generated according to simple specifications. The combinations are made possible by separating each feature into a distinct layer of subclasses implementing that specific feature. Using a system that understands composition of these layers, he achieves code size and performance as good as or better than available hand-coded libraries.

Figure 4.9 illustrates the concept of using layers of subclasses to select features. If, for instance, only features I and III were included, the implementation would include classes A-I, B-I, and A-III. Class C would not exist at all, as the only feature requiring it was not included. Externally, class A would be presented by A-III and class B by B-I, the lowest subclasses in the selected layers.

As a concrete example, consider figure 4.10, which describes the PLA layers of a deque with several possible features. At the top is an abstract `Deque` class without implementation. The first level is optional

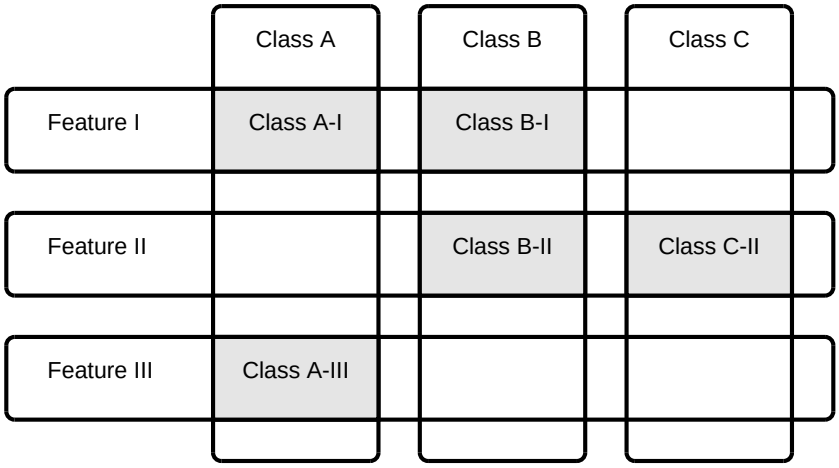

Figure 4.9: Feature selection in a product-line architecture system. Rows represent features, columns represent classes, shaded intersections represent subclasses implementing specific features.

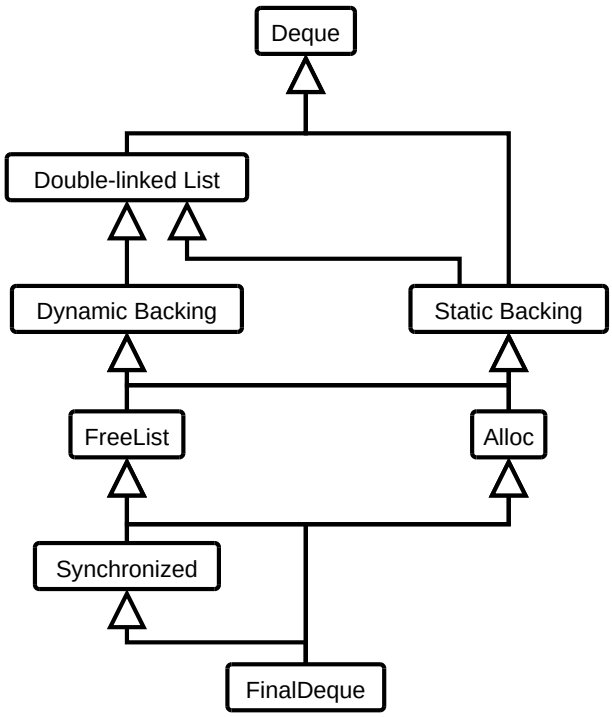

Figure 4.10: The inheritance hierarchy for a PLA implementation of a deque.

and allows the deque to be regarded as a double- or single-linked list. The second level allows selecting whether the elements should be stored dynamically using a linked list, or statically in an array. Selecting both linked lists and array backing allows constant-time access to elements and easy traversal given any element. These two levels together allow five different variants, as dynamic backing requires a linked list.

The third level gives the option of using a free-list to store discarded elements, or simply allocating new elements all the time. The optional fourth level makes all operations on the deque synchronized, essential in multi-threaded applications, but inefficient otherwise. Eventually, we get to the final `Deque` implementation. As the inheritance arrows indicate, each level is formed by inheriting from the previous selected level.

The number of combinations of features that can be selected with this model is five for the first two levels and two each for the remaining two levels, a total of 20 different deques. Maintaining even this modest number of separate deque implementations would be cumbersome and error-prone, whereas maintaining the individual parts of this system reduces redundancy and allows the programmer to concentrate on getting the elements right. Batory's example uses 8 layers of features, giving rise to 144 possible different deques even without repeating any features.

Jumbo allows an easy implementation of what Batory implemented as a stand-alone system. By abstracting on superclass names, we can automatically pick the desired set of features and generate code for them. Figure 4.11 shows a simple way to select among possible features. For each selected feature, new subclasses implementing that feature are added, and the name of those classes are remembered to be used as superclasses for the next level. Note that not all classes involved need to be subclasses for each feature.

Some features have restrictions on their usage. In this example "dynamic" requires "dll" or "sll", one of either "dynamic" or "static" list allocation must be selected, and one of either "freelist" or "alloc" element allocation must be selected. Other restrictions could be an ordering of features or mutually exclusive features. The feature selection in this example is primitive, but could easily be improved to automatically check complex feature restrictions similar to the systems described by Batory[5].

The implementations of the features themselves are simple. The implementations of three of the features are shown in figure 4.12. The "dll" feature adds `next` and `prev` references to the elements, a `head` field to the deque, and methods to add and remove elements from the deque (only the front adders and removers are shown). The `head` field is required so the links can be updated when elements are added and removed.

```
Code makeDeque(List features) {
  Iterator featureiter = features.iterator();
  String nextfeature = (String)featureiter.next();
  Code code = $<class DequeElem implements PLADequeElem { Object data; }>$;
  String parentdeque = null, parentelem = "DequeElem";
  if (nextfeature.equals("dll")) {
    code = makeFeatureDLL(code, parentdeque, parentelem);
    parentdeque = "DLLDeque"; parentelem = "DLLElem";
    nextfeature = (String)featureiter.next();
  } else if (nextfeature.equals("sll")) {
    code = makeFeatureSLL(code, parentdeque, parentelem);
    parentdeque = "SLLDeque"; parentelem = "SLLElem";
    nextfeature = (String)featureiter.next();
  }

  if (nextfeature.equals("static")) {
    code = makeFeatureStatic(code, parentdeque, parentelem);
    parentdeque = "StaticDeque";
  } else if (nextfeature.equals("dynamic")) {
    if (!(features.contains("dll") || features.contains("sll")))
      throw new Error("Feature ‘dynamic’ requires ‘dll’ or ‘sll’");
    code = makeFeatureDynamic(code, parentdeque, parentelem);
    parentdeque = "DynamicDeque";
  } else throw new Error("Must specify either ‘dynamic’ or ‘static’");
  nextfeature = (String)featureiter.next();

  if (nextfeature.equals("freelist")) {
    code = makeFeatureFreelist(code, parentdeque, parentelem);
    parentdeque = "FreeListDeque"; parentelem = "FreeListElem";
  } else if (nextfeature.equals("alloc")) {
    code = makeFeatureAlloc(code, parentdeque, parentelem);
    parentdeque = "AllocDeque";
  } else
    throw new Error("Must specify either ‘freelist’ or ‘alloc’");

  if (featureiter.hasNext()) {
    nextfeature = (String)featureiter.next();
    if (nextfeature.equals("synchronized")) {
      code = makeFeatureSynchronized(code, parentdeque, parentelem);
    }
  }
  code = $<‘Stmt(code)
    public class FinalDeque extends ‘parentdeque implements PLADeque {}
    public class FinalElem extends ‘parentelem { }>$;
  return code;
}
```

Figure 4.11: The feature-selection part of a PLA-like deque implementation in Jumbo

```
Code makeFeatureDLL(Code code, String parentdeque, String parentelem) {
  return $<‘Stmt(code)
      public class DLLDeque extends ‘parentdeque {
        DLLElem head;
        void addFront(DequeElem newelem) {
          ((DLLElem)newelem).next = head;  head.prev = (DLLElem)newelem;
          head = (DLLElem)newelem;
        }
        DequeElem getFront() {
          DequeElem frontelem = head;  head = head.next;
          head.prev = null;  frontelem.next = null;
        }
      }
      public class DLLElem extends ‘parentelem {
        DLLElem prev, next;
      }>$;
}
Code makeFeatureStatic(Code code, String parentdeque, String parentelem) {
  return $<‘Stmt(code)
      public class StaticDeque {
        DequeElem[] elements = new DequeElem[2];
        int frontindex = 0, backindex = 0;
        void addFront(DequeElem newelem) {
          if (frontindex == 0) {
            DequeElem[] newelements = new DequeElem[elements.lenght*2];
            System.arrayCopy(elements, 0,
                             newelements, elements.length, backindex);
            backindex += elements.length;  frontindex = elements.length;
          }
          elements[--frontindex] = newelem;
          super.addFront(newelem);
        }
        DequeElem getFront() {
          super.getFront(frontelem);
          return elements[frontindex++];
        }
        public int size() { return backindex-frontindex; }
      }>$;
}
Code makeFeatureDynamic(Code code, String parentdeque, String parentelem) {
  return $<‘Stmt(code)
      public class DynamicDeque {
        public int size() {
          int size = 0;
          for (‘parentelem tmp = head; tmp != null; tmp = tmp.next)
            size++;
          return size;
        }
      }>$;
}
```

Figure 4.12: Selected feature implementations for a PLA-like deque implementation in Jumbo

At the next level, we choose how to store the elements in the deque, either in an array or using a linked list. The "static" feature manages an array of elements, but makes sure to call the methods in the super class that might manage the linked lists. The "dynamic" feature instead can rely on the linked list implementations to store the head reference, and so only needs to implement the `size()` method. Further features are implemented in similar ways.

Java's type system has some influence on how the PLA's can be designed. In this case, while the external interface specifies that `Object`s are stored in the deque, internally they are wrapped in a `DequeElem` object. Since Java does not allow overloading on return types, the allocation feature must wrap each `Object` in a `DequeElem` object and transition between the internal and external method names. Additionally, the feature implementations must use the correct types. For instance, the "dynamic" feature uses the extensions that "dll" or "sll" have added, and so must refer to `'parentelem` rather than `DequeElem` to ensure the existence of the `head` and `next` fields. The "static" feature can choose which it refers to, as it doesn't depend on the additions.

We must also be careful as to which classes implement the interfaces used to make the code externally visible. The `PLADeque` interface, specifying external method names, can only be implemented from the allocation layer onwards, while the `PLAElem` interface must be implemented at the beginning, so the classes can use the type. In a more complex system, such restrictions could prove problematic.

This example shows how Jumbo's code generation features can be put to other uses than just run-time specialization. Using Jumbo, we can implement product-line architectures simply and efficiently, allowing the programmer to concentrate on correct implementation of the individual features. If fact, we could go one step further, and combine the different versions of the same method into one, avoiding the overhead of super method calls. As long as the combined method does not become too large to optimize, this should give some additional savings.

4.3.4 Anonymous Database Queries

Whether for legal reasons, to prevent embarrassment, or to protect commercial secrets, many database users may want to keep their queries hidden from database owners. At the same time, downloading the entire database may be impractical or even illegal. Some way to allow a database search to take place without showing the details of the search criteria is desirable. This example shows how to implement anonymous

```
AnonDB result =
  select(new String[]  "name", "income" ,
         and(lessthan(cell("income"), cell("expenses")),
              equals(cell("job"), constant("Manager"))));
```

Figure 4.13: An SQL-like query using Java function calls.

```
BoolClause lessthan(final Source s1, final Source s2) {
  return new BoolClause() {
    public boolean eval(Object[] row) {
      return ((Integer)s1.getVal(row)).intValue()
           < ((Integer)s2.getVal(row)).intValue();
    }
  };
}

BoolGen lessthan_gen(final SourceGen s1, final SourceGen s2) {
  return new BoolGen() {
    public Code eval(Code row) {
      return $<((Integer)‘Expr(s1.getVal(row))).intValue()
              < ((Integer)‘Expr(s2.getVal(row))).intValue()>$;
    }
  };
}
```

Figure 4.14: Turning a direct query into a code-generating query

database queries for a very simple database.

This implementation can be done by compiling the query into a program that can then be transmitted to the database server and run there, transmitting just the results back to the user. The binary form conceals the criteria and at the same time allows the creation of an optimized query. If so desired, the database server can provide the user with the building blocks for the query in a form that allows for database-specific optimizations.

In this example, we show how a query can be built for a simple relational database. We assume a simple database containing rows of object arrays. The objects in the database can be either strings or integers. We can perform simple SQL-like queries that return new databases as their result. Figure 4.13 shows how a query can be created using Java function calls. The example query returns the name and income of all managers whose income is less than their expenses. Creating such a query from an actual SQL statement is a simple parsing task. By using function objects, the implementations of the query functions take up only a few lines each.

```
public void queryRow(Object[] row, AnonDB newdb) {
  if ((((Integer)row[2]).intValue() <
       ((Integer)row[3]).intValue()) &&
      row[1].equals("Manager")) {
    Object[] newrow = new Object[2];
    newrow[0] = row[0];
    newrow[1] = row[2];
    newdb.addRow(newrow);
  }
}
```

Figure 4.15: The code generated for the query

Figure 4.14 shows one of the original query functions and its code-generating equivalent. The `Source` objects represent either column names or constants, in the form of a function object that will return the appropriate value for the current row. The original function is evaluated once for each row.

In the code-generating version, we manipulate `Code` objects rather than the actual values in the function object. The `SourceGen` objects return code snippets that either get a particular row using direct array lookup or consist of a single integer constant. The `getVal()` method creates code that looks up the column indicated by the `SourceGen` object in the row indicated by `row`. The result is a new piece of code rather than a boolean. Not only can the lookups in the database rows be statically indexed, the result rows can be created using only static indices.

In figure 4.15, we see the code generated for the query in figure 4.13 on a database with the columns Name, Job, Income, Expenses. The code shown here is just the body of the loop iterating over the rows. The column names have been entirely compiled away, and the result rows are created by direct array copying rather than by iterating over the result rows.

To evaluate the performance of the generated database queries, we ran four different queries on a simple database, comparing the generated version with a plain version. The database used had four columns: Name, Job, Income, Expenses. The first two are strings, and the last two are integers. We ran the queries on databases of up to 1 million entries. The results returned from the queries are in the form of new databases with the information copied from the database queried.

The four queries we ran are:

empty: Selects no rows. This query gives the minimum speedup. The running times for this query are shown in figure 4.16. The corresponding SQL statement would be:

Query	Codegen time (micros)	Stddev	Crossover point	Relative speed
Empty	15348	2111	12500	3.0
Managers	16119	2111	7500	10.7
Losses	22882	1816	15000	3.1
Complex	24130	1756	32000	3.4

Table 4.3: Speedups and crossover points for database queries

```
SELECT name FROM db WHERE Job = Mangler
```

losses: Selects managers whose income is less than their expenses. This query is slightly more complex and selects 14% of the entries. The running times for this query are shown in figure 4.17. The corresponding SQL statement would be:

```
SELECT name, income FROM db
WHERE income < expenses AND job = Manager
```

managers: Selects all info from all managers. This query selects 57% of the entries, and extracts all information from them. It illustrates how much faster the generated code is at creating the result database. The running times for this query are shown in figure 4.18. The corresponding SQL statement would be:

```
SELECT name, income, job, expenses FROM db WHERE job = Manager
```

complex: Selects no rows by evaluating a complex statement. This isolates the time spent evaluating the query statement and spends no time building a result database. The running times for this query are shown in figure 4.19. The corresponding SQL statement would be:

```
SELECT name FROM db
WHERE income < expenses AND job = Manager
  AND income < expenses AND job = Worker
```

The asymptotic speeds and crossover points are shown in table 4.3. Crossover points fall in the area of 1000 to 10000 database entries, which by todays standards are small databases. The asymptotic speed goes from 3 times as fast for the empty query to over 10 times as fast for the managers query.

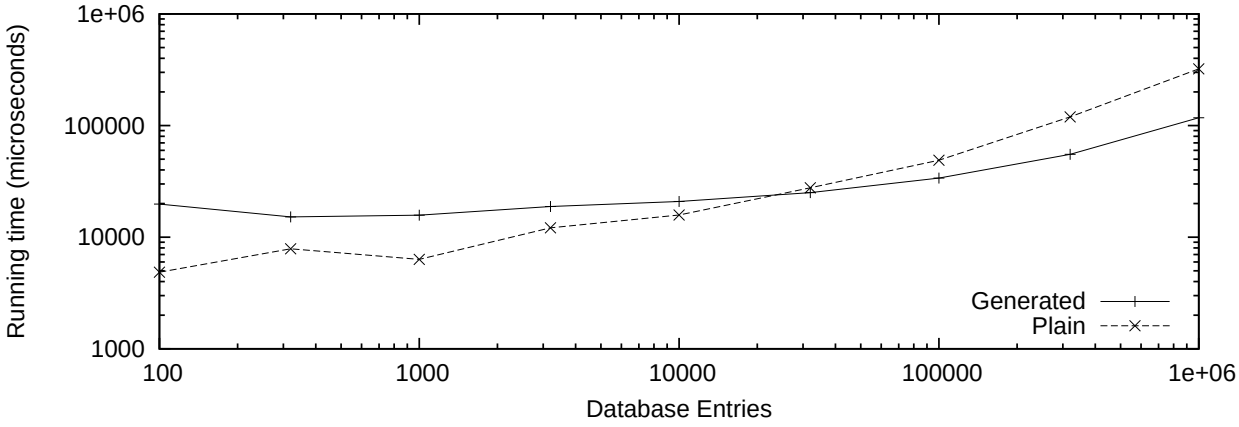

Figure 4.16: Execution time of “empty” query (logarithmic scale)

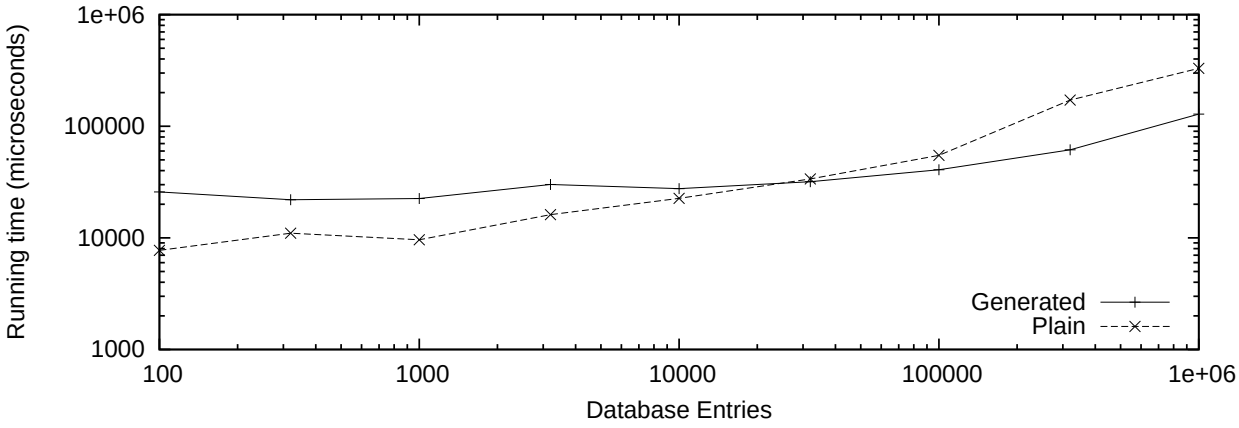

Figure 4.17: Execution time of “losses” query (logarithmic scale)

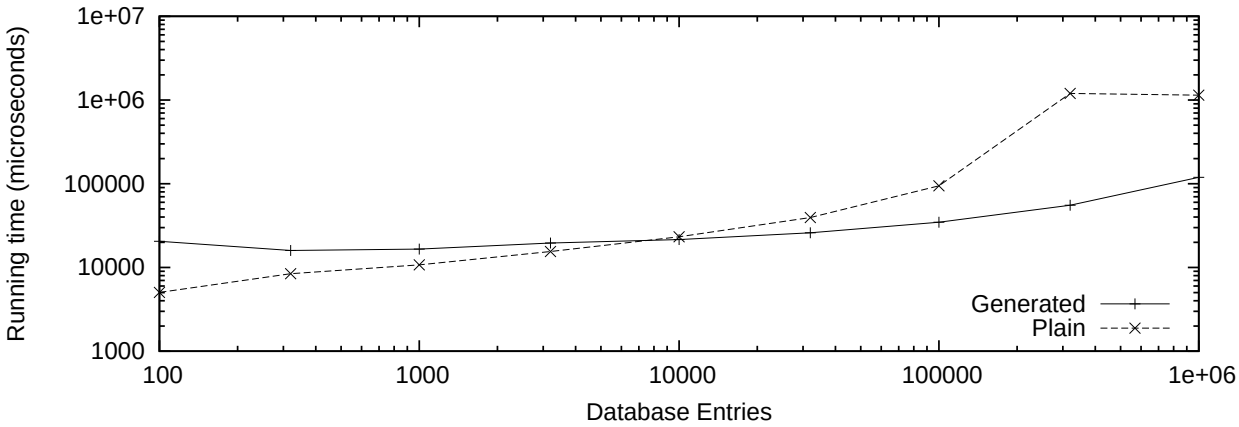

Figure 4.18: Execution time of “managers” query (logarithmic scale)

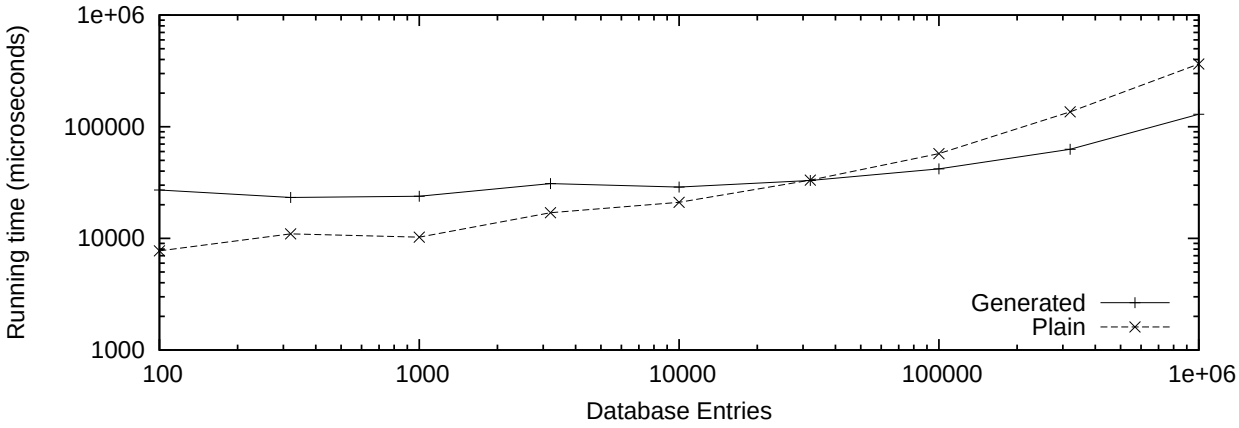

Figure 4.19: Execution time of “complex” query (logarithmic scale)

There is a difference in speed gain of only 13% between the empty query and the complex query. Since both of those queries create an empty result database, this difference is the gain in evaluating a more complex query condition with code generation. At the same time, the difference between the empty query and the managers query is a factor of 3.6. Since the selection criteria for those two queries are equally complex, this speedup is entirely due to faster creation of the result database. This difference in speedup indicates that while modest speedup is gained by using code generation to evaluate the query condition, assembling the result database using specialized code yields significant speedups. The code generator for the condition checking is significantly more complex than the code generator for the result builder, yet yields comparable speedup, indicating that using code generation for the result builder alone would be a simple way to speed up queries.

This example implements a small but important part of database queries. The code-generating version was easy to implement, and would be easy to extend to implement joins and other important features of full databases. Thus, not only does Jumbo allow a simple implementation of anonymous queries, it yields a speed gain in the process.

4.3.5 High-Level Numeric Algorithms

This example is taken from Hardwick and Sipelstein’s paper on using Java as an intermediate language[33]. In their paper, they describe the translation from NESL, a high-level language for vector and matrix calculations[8], through VCODE, a stack-based intermediate form with powerful vector and matrix primitives, into Java, implementing each VCODE primitive as a Java method.

A high-level domain-specific language like NESL is a good target for code-generating systems. Direct translations like that of Hardwick and Sipelstein use a separate method for each primitive, causing the creation of numerous intermediate vectors and matrices. Using a code-generating approach, we can perform implicit loop fusion and do away with extraneous loops and allocations.

```
function linefit(x, y) =
let
  n    = float(#x);
  xa   = sum(x)/n;
  ya   = sum(y)/n;
  Stt  = sum({(x - xa)^2: x});
  b    = sum({(x - xa) * y: x; y}) / Stt;
  a    = ya - xa*b;
  chi2 = sum({(y - a - b * x)^2: x; y});
  siga = sqrt((1.0 / n + xa^2 / Stt)* chi2 / n);
  sigb = sqrt((1.0 / Stt) * chi2 / n)
in
  (a, b, siga, sigb);
```

Figure 4.20: Line-fitting algorithm in NESL

Our example is the line-fitting algorithm shown in Appendix 1 in Hardwick and Sipelstein. Figure 4.20 shows the NESL source code for the algorithm. NESL uses syntax similar to Haskell's list comprehensions for loop iterations. For instance, `{(x - xa)^2: x}` means "create a new vector with each element being the square of the corresponding element of the vector `x` minus `xa`".

A direct translation of the NESL code in figure 4.20 into Java is shown in figure 4.21. The main difference is that infix operations have been replaced with prefix operations. This implementation depends on a number of (overloaded) methods to perform vector operations like subtraction and multiplication. Each function that

```
public void printFit(double[] x, double[] y) {
  double n = (double)x.length;
  double xa = sum(x)/n;
  double ya = sum(y)/n;
  double Stt = sum(square(sub(x, xa)));
  double b = sum(mul(sub(x, xa), y))/Stt;
  double a = ya - xa*b;
  double chi2 = sum(square(sub(sub(y, a), mul(x, b))));
  double siga = Math.sqrt((1.0 / n + xa*xa / Stt) * chi2 / n);
  double sigb = Math.sqrt((1.0 / Stt) * chi2 / n);
  System.out.println("("+a+", "+b+", "+siga+", "+sigb+")");
}
```

Figure 4.21: The line-fitting algorithm translated directly into Java

```
public static Code makePrintCodegenFit(final CG_Vector x,
                                       final CG_Vector y) {
  return $<
    ‘Stmt(assign("n", len(x)));
    ‘Stmt(assign("xa", div(sum(x), "n")));
    ‘Stmt(assign("ya", div(sum(y), "n")));
    ‘Stmt(assign("Stt", sum(square(sub(x, "xa")))));
    ‘Stmt(assign("b", div(sum(mul(sub(x, "xa"), y)), "Stt")));
    double a = ya - xa*b;
    ‘Stmt(assign("chi2",
          sum(square(sub(sub(y, "a"), mul(x, "b"))))));
    double siga = Math.sqrt((1.0 / n + xa*xa / Stt) * chi2 / n);
    double sigb = Math.sqrt((1.0 / Stt) * chi2 / n);
    System.out.println("("+a+", "+b+", "+siga+", "+sigb+")");
    >$;
}
```

Figure 4.22: The line-fitting algorithm translated into a code-generating form

returns a vector creates a new array to hold the result. This implementation is not the VCODE translation normally used for NESL, but the semantics are similar. While some inlining can be done, the intermediate arrays are difficult to optimize away.

Figure 4.22 shows how the algorithm for line fitting looks using the code-generating syntax. Apart from the abstraction on assignment and variable names, this code looks the same as the direct translation. Notice that when no vectors are involved, the evaluation can be done using normal infix syntax. The resulting code has only one iteration for each line (resulting from the use of `sum`), the remaining iterations have been fused into statements within the sum iteration.

The code-generating version abstracts on the assignment operation, using overloading to ensure correct typing of the assigned values. Figure 4.23 shows two examples of code generation functions. Each vector operation becomes a function object that performs abstraction on the assignment. `CG_Vector` and `CG_Double` are both function object types, representing vectors and doubles, respectively. They each provide an `assign` method that given a variable name will return code that assigns the expression value to that variable. The vector types also provide a `len` method that will generate code to get the length of the vector.

Figure 4.23(a) shows the method for subtracting a number from each element of a vector. It assigns the two relevant values to intermediate variables, then assigns the result of the subtraction to the variable named in the argument (`v`). The index variable `i` is passed along to the given vector, and will eventually be used to look up the value in an actual array. Note how the vector is not traversed in any way, but just a single

```
static CG_Vector
       sub(final CG_Vector c,
           final CG_Double n) {
  return new CG_Vector() {
    Code len()
    { return c.len(); }
    Code assign(String v,
                String i) {
      String x = gensym("x");
      String n1 = gensym("n");
      return $<
        ‘Stmt(c.assign(x, i))
        ‘Stmt(n.assign(n1))
        double ‘v = ‘x-‘n1;
      >$;
    }
  };
}
```

(a) Vector subtraction

```
static CG_Double
       sum(final CG_Vector c) {
  return new CG_Double() {
    Code assign(String v) {
      String i = gensym("i");
      String x = gensym("x");
      return $<
        double ‘v = 0;
        for (int ‘i = 0;
             ‘i<‘Expr(c.len());
             ‘i++) {
          ‘Stmt(c.assign(x, i))
          ‘v += ‘x;
        }>$;
    }
  };
}
```

(b) Vector summation

Figure 4.23: Two of the code-generating methods

element is handled.

In figure 4.23(b), we see how to produce code for summing the values of a vector. Summation is the only operation currently implemented that iterates over the vector; all other operations simply operate on the elements. We pass down the index to the expressions inside the sum and assign the resulting value to the variable given as argument.

The abstraction on assignments is necessary because of Java’s distinction between expressions and statements. The right-hand side of an assignment must be an expression, and thus cannot contain a for-loop. This problem could be eliminated by introducing a “block-expression”, essentially a statement that always has a value. Experiments with manually simulating block expressions show that the extra assignments account for about 50% of the time spent. However, we do not currently want to extend Java with more than the minimum syntax required to support code generation. The fact that these extra assignments are not optimized away by HotSpot shows the limited range of its optimizations.

Figure 4.24 shows a (decompiled) example of the code generated by this system. This fragment is generated from the line assigning “chi2”, which involves vector-number calculations, vector-vector calculations, element-wise squaring of a vector, and the sum over a vector. All the element-wise operations have been reduced to simple operations and assignments, avoiding 4 of 5 loops and all 4 allocations compared to the

```
double chi2 = 0;
for (int i_45 = 0;
     i_45<y_27.length;
     i_45++) {
  double x_50 = y_27[i_45];
  double x_48 = (x_50-a);
  double x_51 = x_26[i_45];
  double x_49 = (x_51*b);
  double x_47 = (x_48-x_49);
  double x_46 = (x_47*x_47);
  chi2 = (chi2+x_46);
}
```

Figure 4.24: Code generated for the line `chi2 = sum({(y - a - b * x)^2:  x; y});` (decompiled)

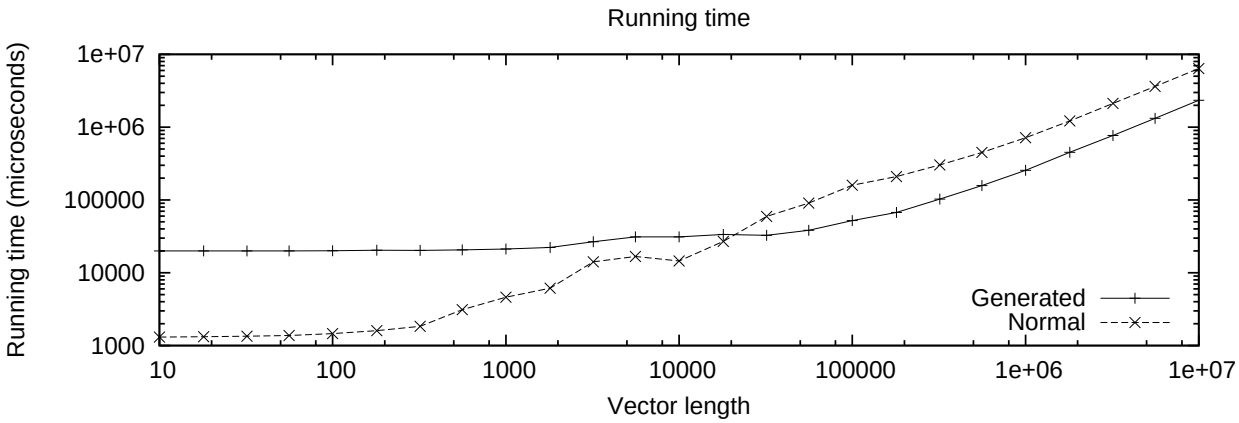

Figure 4.25: Execution time of line-fit (logarithmic scale)

direct implementation. Additionally, the code is of simple and regular form, making optimizations easy.

Figure 4.25 shows the execution time in microseconds for executing the code with varying vector sizes. The “Normal” line is the execution time of the direct translation that uses a method call for each vector operation. The “Generated” line is the time spent in the Jumbo version, including compilation, but excluding the time spent loading the Jumbo classes. Above 10,000,000 elements, Java ran out of memory. The cross-over point is at 20,000 elements in the vector, and the speed of the generated version goes towards approximately 2.6 times that of the normal version.

This example does show how Jumbo’s code generation can be used to perform powerful transformations. Not only are all the method calls heavily inlined, several full loops have been fused together. While this example shows a significant speed increase, the increase is not as much as would be expected for such an

powerful optimization. Once again, we see the influence of the HotSpot compiler. An equivalent system implemented in a fully compiled language should get an even bigger speedup due to additional opportunities for optimizing. If we added a "block-expression" to Jumbo, the reduced amount of copy assignment would bring the speed increase up to about a factor of 5.

The code-generating version is remarkably similar to a parsed version of the original NESL code. The building blocks are simple to write and understand, and overall Jumbo shows itself as a useful tool for implementing domain-specific languages, as well as for performing high-level optimizations.

4.4 Other examples

In this section, we discuss various other examples, both from the literature and from our own tests, and how Jumbo would be able to handle them.

Macros: Jumbo can be used in a manner similar to macros in Scheme[11]. This requires that all code except for macro definitions be quoted, and the compilation is a two-stage process, where the first stage compiles the macros, and the second stage uses the macros and compiles the rest of the code. This may be useful in light of the fact that Java has no macro system of its own.

Data structures: Because Jumbo allows the creation of new types, it is possible to use it to create specialized data structures. This can be used to create polymorphic types, or potentially for other uses. Since no other code-generating system for a statically typed language allows the creation of entirely new types, little research has been done so far into what can be done in this direction.

Eliminating tail recursion: Poletto et al.[66] has an example where a tail-recursive binary search algorithm is specialized with regards to a certain data set. This effectively inlines the entire data set as a series of if-statements, eliminating array lookups and function calls. Jumbo could do the same, but the size of the generated function would preclude run-time optimization in most cases, thus not giving any benefits.

Exponentiation: Many systems involve exponentiation of large entities like matrices. Reducing the number of multiplications necessary for the exponentiation could save significant amounts of time. The Russian Peasant algorithm reduces the number somewhat, and Knuth[47] shows that for some cases,

it is possible to reduce it even further. Since this can be tested without multiplying the large entities, a code generating exponentiation algorithm could be useful.

Unrolling: The dot product example shows one kind of unrolling that works well for small, static vectors. However, for large vectors or vectors where the contents are not known, partial unrolling[1] is a well-known optimization technique. Especially in parallel systems, this allows more optimizations, but in Java, there is less gain from this technique. Only if subexpressions inside the loop body can be made static can significant speedups be achieved.

Marshalling: Marshalling of arguments for remote procedure calls in, e.g., CORBA[32] is done with compile-time generation of stub code. Jumbo could be used to generate the stubs at run-time. For function calls known at compile time, Jumbo offers no speed-up, but allows the marshalling code to be generated without external tools. Jumbo might however be used to create new methods in CORBA components at run time.

Function iteration: Newton's method, as well as most fractal calculations, involve iterating a specific function numerous times. In the case of Newton's method, the derivative function can be automatically created and fully inlined. For fractal calculations, even a slight optimization will be repaid many times over. Jumbo should be of use in both cases.

Filter composition: Both network layers and graphics filters are implemented as a number of primitive filters placed in sequence. Using techniques similar to those of section 4.3.5, these might be optimized, removing loop and invocation overheads. For filters with arguments, more optimizations may be possible because the arguments may be static. However, we cannot blindly inline the code, as we may end up with unoptimizable code.

4.5 Conclusion

This chapter has given some examples of what Jumbo's code generation system can be used for. In some areas, significant speed increases can be achieved by specializing code for specific uses. Typical execution speeds of the generated code are 200% to 300% of the original code, with some examples running about an order of magnitude faster.

The NESL example in particular shows how judicious application of interpreter specialization can be combined with application-specific optimizations to increase running speed. At the same time, as a combinator-based code generation system, Jumbo allows opaque code generation, yielding the benefits of instance-specific code without requiring the disclosure of secrets of implementation or usage.

The crossover point, where the cost of compilation is regained by the higher execution speed, depends on the speed of the code generators as well as on the speed of the final code. In the next chapter, we shall investigate various ways we could speed up the code generation process.

The syntax of Jumbo is simple in practical usage, though somewhat hampered by the necessity of using syntactical categories for many antiquotes. The simplicity encourages experimentation and allows more complex applications. At the same time, Jumbo allows the creation of new types at compile time, opening a new realm of possibilities.

Chapter 5

Optimizing

The Jumbo system described in chapter 3 is stable and publically available. In this and the next chapter, we discuss the possibilities for optimizing Jumbo code. The principal optimizations that seem possible arise directly from the compositional nature of the Jumbo compiler — another advantage over a monolithic compiler. The optimizations discussed in these two chapters are experimental, and have been implemented as a proof of concept. Practical implementation of them is a topic for further research, and beyond the scope of this dissertation.

The previous chapter showed how the crossover point, where the time spent generating code is recouped by faster code, affects the potential applications. The lower the crossover point, the more areas can benefit from code generation. We pursue this goal by investigating optimizations.

There are two areas where we could perform optimizations: On the generated code, or on the code generation process. Optimizing the generated code would increase the overall efficiency when execution time dominates code generation time, but could raise the cross-over point if the optimizations are too expensive. Optimizing the code generation process will lower the cross-over point and thus make the code generator applicable in more situations.

For reasons discussed in section 5.1, we have rejected the idea of optimizing generated code. Optimizing the code generators, on the other hand, appears very promising. In section 5.2, we describe various possible ways to optimize the code generation process. As a proof of concept, we have implemented one of the possible approaches to optimizing the code generators. The results of this implementation are shown in section 5.4, while a full description can be found in the next chapter.

5.1 Optimizing the Generated Code

In this section, we consider the problems inherent in optimizing the code as it is being generated. We will first discuss the problems inherent in optimizing Java byte-code, and then briefly discuss what special problems are inherent to optimizing code fragments.

Some research has been done into how to efficiently generate code in staged compilation. The Fabius system[50] uses as few as six instructions per instruction generated. However, the generated code normally is not subject to optimizations, as that would take too much time while generating code. The fact that code generated this way still gives significant speedups is an indication of the power of staged compilation.

5.1.1 How Much Can Java Byte-code Be Optimized?

Many conventional optimizations, such as strength reduction and register allocation, are relatively ineffective in Java bytecode. The interpretation overhead is the same to perform a division as to perform a logical shift operation. Any attempt at transforming multiplication into a series of additions and shifts would create slower code in all but a few special cases. We cannot fill in delay slots or make use of instruction-level parallelism at the bytecode level. Register allocation is of limited effect, too, as most JIT compilers perform their own register allocation[92, 38]. Since a number of other optimizations mostly work by allowing lower-level optimizations to take place, those are also of diminished use. The JVM, having access to machine code, is expected to perform optimizations, rather than the compiler.

The author's Masters Thesis[10] describes an off-line system (Cream) that performs optimizations directly on bytecode. Cream performs common-subexpression folding, dead-code elimination and loop-invariant removal using an extensive inter-procedural side-effect analysis. Even with full analysis and liberal assumptions about native methods, the speedup was barely measurable.

The Soot system from McGill University[87] transforms Java bytecode into three different internal representations in order to perform various optimizations. When optimizing the SPECjvm98[15] benchmarks, they report speedups of up to 60% on a single instance (a ray-tracing program), but in most cases a more modest speedup of less than 10%.

From the previous, it should be obvious that the prospects of doing standard optimizations at an early stage are poor. Even systems with extensive knowledge about the program do not gain much speedup, and

we will be further hampered either by the fragmented nature of the code, if working at compile time, or by the time restrictions if working at run time.

5.1.2 Optimizing Code Fragments

As the above section shows, there is little to be gained by optimizing Java byte-code. However, run-time code generating systems that generate other languages than Java byte-code, or that creates native code for Java, may benefit from optimizing code fragments. In this section, we discuss the effect that code fragments and code with holes has on performing optimization.

When trying to optimize a code fragment, we face the problem of needing information that is not yet available. There are two distinct kinds of missing information that we will encounter: missing information about the surrounding code, and missing information about holes in the code.

The missing information about surrounding code prohibits a number of optimizations. Unless this information is passed to the code fragment, as for instance done in DynJava, any external reference would be unknown and thus untyped. This lack of types would prohibit a number of optimizations, but larger pieces of code may have some optimization opportunities arise within the fragment.

When a large piece of code contains a hole, we may want to apply aggressive optimizations, but be hindered by the hole. If an expression can contain assignment statements, even a hole of expression kind would prevent most optimizations like constant propagation and common subexpression folding. From our experience with Jumbo, we have found that the code inserted rarely affects the possible optimizations. Most expression holes do not assign any outside variables; many do not even call any methods. Statement holes are more likely to cause problems, as they can contain variable declarations, throw statements, break statements, and are more likely to contain method invocations or assignments.

One way to use the frequent simplicity of holes is to do conditional (assumption-based) optimizations. Say that we want to do constant propagation, and we are targeting one variable `x`. We could assume that the holes do not assign to `x` and go ahead with creating an optimized version where `x` has been constant propagated. When the hole is filled, we check if the inserted code does not assign to `x`. If it does assign to `x`, we will have to fall back on the unoptimized version.

There are two important problems to consider in this scenario: Avoiding combinatorial explosion, and figuring out what assumptions to make. If we make a separate assumption about each hole, we would

get a number of optimized versions exponential in the number of holes. Even for one hole, if we make assumptions about several variables, we again get a number of optimized versions exponential in the number of variables. To avoid this, we should only build assumptions when there are significant benefits, and combine as many assumptions as possible with each set of optimizations. Forced use of variable hygiene would help decrease the number of assumptions required, as we would be able to reason about what variables can be affected by a hole.

Figuring out what assumptions to make is most easily handled by starting with the assumption that the holes do nothing and then figuring out which optimizations give the most advantage. Only when significant optimizations are possible should we consider what assumptions are necessary. The choice of optimizations to attempt is also affected by the use of assumptions: local optimizations are less likely to be blocked by holes than optimizations that span significant amounts of code. Useful assumptions could include whether a variable is read from or written to in the hole, whether there are branches out of the hole, whether global variables or fields are read or modified, and whether any functions are called. Each assumption should be checkable in linear time, so that as little time as possible is spent checking the assumptions at run time.

5.1.3 Optimizing Code at Load-Time

While the code generated by the Jumbo system itself is as optimized as that of Javac, the separate definitions of code may cause some simple patterns of inefficient code. Computer folklore says that (ironically) most compilers do not optimize computer-generated code as well as hand-written code. For example, the example in section 4.3.5 assigns every temporary value to a variable. Experiments with folding the assignments into expressions show that this pattern is responsible for almost 50% of the execution time, even after HotSpot optimizations were applied. Some, but not all, of these could be removed by introducing more powerful constructs in Java, like a “block-expression” (i.e. a sequence of statements that have a value). However, such additions should be considered carefully before they are introduced into a language.

The simplest of use-def analyses would find the extraneous assignments mentioned above, and might improve execution speed at only a slight increase in compilation time. However, load-time optimizations, like JIT compilers, need to balance the improved execution speed and the extra time spent optimizing. The optimizations must be likely to pay off, and not require costly analyses. At the same time, there is the danger that we perform the same optimizations as the JIT would later do, but spend more time on it. We currently

do not consider any load-time optimizations profitable enough to warrant the work involved in implementing them.

5.2 Optimizing the Code Generators

Rather than attempting to optimize the code that is being generated, it appears more promising to optimize the code generation process itself. In this section, we look at various ways such optimization can take place.

Experimenting with the dot product example shown in figure 3.1, we have found that up to 80% of the compilation time is spent in the higher-level combinators when compiling large classes. Smaller classes share more overhead of creating and loading class information, and may spend only about 50% of their time in the combinators. Compiling away the combinators at the earliest possible stage (precompilation) would improve the overall compiler speed.

Rather than attempting traditional optimizations, which have only a limited effect on Java, we can exploit the statically known structure of the code fragments, together with the definitions of the combinator methods, to perform specialization of the code generators. For instance, the expression `$<'x+1>$` is turned into `binOp(PLUS, x, intConst(1))` as part of the compilation process. Such expressions, being composed almost entirely of known functions, and devoid of any holes, are viable targets for attempts at partial evaluation. Additionally, since the combinators are designed in a particularly systematic way, they should reduce easily.

We consider two different ways to reduce the code generation time in this section. First, we look at doing abstract evaluation (also known as abstract interpretation)[16] to evaluate those parts of the code that are statically known. We then introduce the approach described in detail in the next chapter, that of using a rewriting system to perform partial evaluation. Both these approaches are only possible due to the compositional structure of the compiler, which allows us to perform optimizations on the compiler internals.

5.2.1 Partial Evaluation with Abstract Evaluation

The obvious way to perform partial evaluation on the code generators would be to apply some existing partial evaluation system. However, the only two partial evaluation systems available for Java, JSpec[72] and BCS[56], do not support enough of Java to be applicable to Jumbo. In particular, neither implements exceptions, which are used in several core parts of Jumbo.

One possible way of performing partial evaluation is by a form of *abstract evaluation*. In abstract evaluation, a program is evaluated, not for its final value, but for some approximation of the value that may be possible to find at an earlier stage. In this method, we attempt to evaluate the combinators, using the normal `eval` methods, but with a special environment that does not cause an error when variable lookup fails. Combinators that are successful in all lookups create ClosedCode objects abstracted on variable indices. When a combinator fails to look up a variable, field, method or class, it instead creates a special version of the ClosedCode objects that allows reconstruction of the original combinator call. At the end, instead of writing the results to a class file, the ClosedCode object will emit code that re-creates the combinators, except that some will be replaced by objects that directly create Jaemus code.

As an example, consider the expression `x-(y*2)`, and assume that `y` is defined in the surrounding environment as a non-constant integer variable at local index 3, while `x` is not defined in the current environment. The combinators for this expression are shown in figure 5.1(a). When attempting abstract evaluation, the inner part `(y*2)`, being fully known, can be fully evaluated.

```
mathOp(SUB, variable("x"),
       mathOp(MUL, variable("y"),
                   constant(2)))
```

(a) Combinators for the expression `x-(y*2)`

```
mathOp(SUB, variable("x"),
  new Code() {
    ClosedCode eval(Environment env) {
      return new ClosedCode(
         genAbstractLoad(Int, 3)
        .addConstant(2)
        .addMath(MUL, Int),
        env, Int);
    }
  });
```

(b) The expression of figure 5.1(a) reduced by abstract evaluation (simplified)

Figure 5.1: A simple expression and its reduction by abstract evaluation.

In the result of abstract interpretation in figure 5.1(b) (simplified for readability), we see no traces of the method bodies of the combinators involved. Instead, a textual representation of the result of evaluating `y*2` has been generated as a function object, and replaces the combinators. This result is in the form of calls to Jaemus to construct Java byte-code instructions. The first instruction loads `y` from the (abstracted) variable index 3. The next instruction loads the constant 2, and finally an integer multiplication is performed. The other combinators, unable to reduce to Jaemus code, have been reconstructed.

This method suffers from the obvious deficiency that lookup errors totally blocks optimization around them. Only when a subpart of the code is fully defined within the code is there any chance of optimization. This method would be more viable if we had a way to get type information about entities defined outside the code fragment.

This approach is specific to the structures found in Jumbo, and so would not directly apply to other Java code, though the techniques may be used in other places. It would be beneficial if the optimizations were applicable to Java code in general.

5.2.2 Partial Evaluation by Rewriting

Instead of the abstract interpretation method, we decided on a source rewriting system that allows more fine-grained specialization. Not only do we consider rewriting systems a more promising approach, it also has the advantage that it is not specific to Jumbo. This system can be applied to any Java code to attempt to increase performance. Our selection of optimizations and assumptions does reflect the structure of Jumbo, and so may not be the best set of optimizations for typical Java code.

We define a number of simple rewrite rules that, while individually small enough to be easy to implement and verify, together can perform significant optimizations. The rewriting system works on the AST level of lifted code, i.e. the syntactic structure has been turned into combinator calls. The code is first normalized to a flattened state with all names put into fully qualified form. Thereafter, individual method calls are inlined (currently by manual selection), and a set of code-reducing rewriters repeatedly performs constant propagation, arithmetic reductions, copy assignment propagation and other small optimizations. The details of the rewriting system are shown in chapter 6.

A (near optimal) possible result of rewriting is shown in figure 5.2, somewhat simplified for readability. Realistic results will be shown below in section 5.4. Each formal argument to the combinators has been replaced with the actual arguments, and wherever the method bodies became small enough, they have been inlined into the calling method. The first operand to the subtraction does not shrink at all, as we cannot find `x` in the environment. However, the second operand, the multiplication, can shrink significantly when we find `y` in the environment. The resulting bytecode and type is inserted into the body.

The two approaches of abstract interpretation and inlining by rewriting have significantly different characteristics. The inlining method will produce the most reduced combinators possible, whereas the abstract

```
new Code() {
  ClosedCode eval(Environment env) {
    ClosedCode cp1 = variable("x").eval(env);
    Type t = cp1.type.lub(Int);

    return new ClosedCode(cp1.bytecode
                          .addConvert(cp1.type, t)
                          .append(Instr.genLoad(3, Int)
                                  .append(Instr.genConst(2))
                                  .addMath(MUL, Int))
                          .addConvert(Int, t)
                          .addMath(SUB, t),
                          env, t)
                .addInfo(cp1);
  }
};
```

Figure 5.2: The expression of figure 5.1(a) reduced by a hypothetical rewriting system.

interpretation method will only produce reduced code when able to reduce it to Jaemus code. However, the inlining method would likely create larger methods than abstract interpretation, which will increase the time spent loading and verifying the code fragment. Both methods suffer when types are unknown, as overloading cannot then be resolved. Some examples of the effect of the rewriting system are shown in section 5.4.

5.3 Reducing the Load Time of the Compiler

In the cases where the whole compiler is loaded every time code is generated, it may pay off to reduce the size of the compiler. It takes about an order of magnitude longer to load and verify the 860 kilobytes of class files currently comprising the Jumbo compiler than it does to compile and load a new class. Tools such as Jax[86], DashO[67] and Jshrink[84] can compress the class files by 30–50% by removing unneeded classes and methods and by shortening names. Such compression would certainly have a significant effect on the time spent loading the compiler.

However, when the code is used in a distributed manner, a class file optimizer cannot tell which classes and methods are necessary in the end. This approach would be most suitable when using Jumbo as an intermediate language for compiling a higher-level domain-specific language, where the set of combinators used is clearly defined. Some obfuscation, generating shorter method and field names, may also be possible for distributed system, but to a limited extent.

The most drastic way to improve the compiler loading time would be to include the compiler in the

JVM. If the compiler were to be loaded as native code at start-up time, it would bypass the overhead of parsing and verifying the class files entirely. This would require either re-writing the compiler in the same language as the JVM in question, or compiling the current compiler to native code. The first alternative would furthermore allow tight integration with the JVM, including any optimizations that may be available there. In particular, it would be possible to avoid Java bytecode altogether, and compile directly into the JVM's preferred execution format. The downside of this is that the system is not portable — the user's JVM would have to be updated before it will work, and the user would be required to use a JVM that supports code generation.

5.4 Rewriting Examples

As argued in the previous section, Jumbo code generators are, in principle, subject to optimization by applying rules of "programming logic" — inlining and simplification. In this section, we illustrate this by rewriting some examples. Because inlining tends to produce large methods, we have stuck to simple programs, and even then we have made some elisions for readability. The underlying programming logic that we use to perform these optimizations is defined in chapter 6.

5.4.1 Rewriting Integer Constant

We start off with a very simple example: An integer constant (`$<1>$`). We will show how the constant expression can be expanded with inlining and then reduced with the code-reducing rewriters. Eventually, it turns into a method that builds and returns a single `ClosedCode` object.

Before any of the optimizing rewriters are applied, we perform a flattening that makes the code easier to reason about. Flattening makes package and class names explicit and turns complex expressions into a series of simple expressions and assignments to temporary variables.

The initial code (after flattening) is shown in figure 5.3. We first create a `Code` object representing an integer constant and then pass it to `System.out.println` (to avoid the entire calculation being removed as unused). The code that will be rewritten next is marked in bold face.

The only thing we can do here is inline the invocation of `integerConstant`. Since it is a static call, it is easy to inline, as can be seen in figure 5.5 (for clarity, we elide package names as well as the unchanged outermost parts). The original `integerConstant` method being inlined is shown in figure 5.4.

```
import uiuc.Jumbo.Util.*;
import uiuc.Jumbo.Jaemus.*;
import uiuc.Jumbo.Compiler.*;

public class IntConst {
  public static void main(java.lang.String[] argv) {
    uiuc.Jumbo.Compiler.Code c;
    c = uiuc.Jumbo.Compiler.Constructor.integerConstant(1);
    {
      final java.io.PrintStream flatten_0;
      flatten_0 = java.lang.System.out;
      flatten_0.println(c);
    }
  }
}
```

Figure 5.3: Initial integer constant code (`integerConstant` expanded from `$<1>$`

```
public static Code integerConstant(final int constval) {
  return new Code() {
      public ClosedCode eval(Environment env) {
        return new ClosedCode(Instr.genConst(constval),
                    env, Type.int_type,
                    ConstVal.make(constval));
      }
    };
}
```

Figure 5.4: `integerConstant` method being inlined

The one parameter is placed in a generated variable, and a spot is reserved for the return value. The inlined method is surrounded by extra scopes, including one with a label that catches the break statements replacing returns. The inner class is not rewritten except where it refers to variables defined outside it, and to guarantee unique names for variables within it.

The extraneous break at the end of the inlined area is removed with the UnusedBreak rewriter, followed by the UnusedScope rewriter which can now remove all the new scopes. In figure 5.6, we see the more reasonable-looking inlined function. The result value of `eval` is formed from three newly created values and the environment being passed unchanged.

We can now start applying other code reducing rewriters. The assigns to `gen8464_return` and `c` can be reduced by using `gen8464_flatten_32` directly in the print statement. The constant `gen8464_constval` can be propagated into three places. All but two of the variable declarations are now useless and can be removed, leaving the code in figure 5.7.

```
Code c;
final int gen8464_constval;
gen8464_constval = 1;
final Code gen8464_return;
gen8464_break: {
  {
    final Code gen8464_flatten_32;
    gen8464_flatten_32 = new Code() {
      public ClosedCode eval(Environment env) {
        {
          final Instr flatten_27;
          flatten_27 = Instr.genConst(gen8464_constval);
          final Type flatten_28;
          flatten_28 = Type.int_type;
          final ConstVal flatten_29;
          flatten_29 = ConstVal.make(gen8464_constval);
          final ClosedCode flatten_30;
          flatten_30 = new ClosedCode(flatten_27, env,
                                      flatten_28, flatten_29);
          return flatten_30;
        }
      }
    };
    {
      gen8464_return = gen8464_flatten_32;
      break gen8464_break;
    }
  }
}
c = gen8464_return;
{
  final PrintStream flatten_0;
  flatten_0 = System.out;
  flatten_0.println(c);
}
```

Figure 5.5: Integer constant code after first inlining

We can now inline the invocation of `genConst`, which determines which of 10 different constant loading bytecodes to use (see figure 5.8). Since the constant value is known, we can reduce most of it. 12 arithmetic rewrites, 30 constant propagations, 3 removals of constant-valued if statements, and various cleanup rewrites later the `eval` method is reduced to figure 5.9. The `genConst` invocation reduced to an object creation statement that creates bytecode number 4, which has no arguments — `iload_1`.

Now only the invocation of `make` remains, which consists of a single object instance creation. This value will be used if the code is evaluated in a context that can be reduced at compile time, or if assigning to a final variable. The full code of the program after all rewrites are done is shown in figure 5.10. The code

```
Code c;
final int gen8464_constval;
gen8464_constval = 1;
final Code gen8464_return;
final Code gen8464_flatten_32;
gen8464_flatten_32 = new Code() {
  public ClosedCode eval(Environment env) {
    final Instr flatten_27;
    flatten_27 = Instr.genConst(gen8464_constval);
    final Type flatten_28;
    flatten_28 = Type.int_type;
    final ConstVal flatten_29;
    flatten_29 = ConstVal.make(gen8464_constval);
    final ClosedCode flatten_30;
    flatten_30 = new ClosedCode(flatten_27, env, flatten_28, flatten_29);
    return flatten_30;
  }
};
gen8464_return = gen8464_flatten_32;
c = gen8464_return;
final PrintStream flatten_0;
flatten_0 = System.out;
flatten_0.println(c);
```

Figure 5.6: The inlined method after extra breaks and scopes are removed.

```
final Code gen8464_flatten_32;
gen8464_flatten_32 = new Code() {
  public ClosedCode eval(Environment env) {
    final Instr flatten_27;
    flatten_27 = Instr.genConst(1);
    final Type flatten_28;
    flatten_28 = Type.int_type;
    final ConstVal flatten_29;
    flatten_29 = ConstVal.make(1);
    final ClosedCode flatten_30;
    flatten_30 = new ClosedCode(flatten_27, env, flatten_28, flatten_29);
    return flatten_30;
  }
};
final PrintStream flatten_0;
flatten_0 = System.out;
flatten_0.println(gen8464_flatten_32);
```

Figure 5.7: Constant propagation and copy assignment reduction allows more cleanup.

```
public static final Instr genConst(int i) {
  if (i > -2 && i < 6)
    return new NoArg(ICONST_M1+(i+1));
  else {
    if ((i >= Byte.MIN_VALUE) && (i <= Byte.MAX_VALUE)) {
      return new ByteArg(BIPUSH, i);
    }
    else if ((i >= Short.MIN_VALUE) && (i <= Short.MAX_VALUE)) {
      return new ShortArg(SIPUSH, i);
    }
    else {
      Constant c = new ConstantInteger(i);

      return new ConstantArg(LDC1_W, c);
    }
  }
}
```

Figure 5.8: The `genConst` method being inlined

```
public ClosedCode eval(Environment env) {
  final NoArg gen8465_flatten_4241;
  gen8465_flatten_4241 = new NoArg(4);
  final Type flatten_28;
  flatten_28 = Type.int_type;
  final ConstVal flatten_29;
  flatten_29 = ConstVal.make(1);
  final ClosedCode flatten_30;
  flatten_30 = new ClosedCode(gen8465_flatten_4241, env,
                              flatten_28, flatten_29);
  return flatten_30;
}
```

Figure 5.9: The `eval` method after inlining `genConst` and reducing

is being created in the fastest way possible, using three object creations and one field lookup.

5.4.2 Rewriting Dot Product

As an example of rewriting a larger system, we show the essentials of rewriting the DotGen example from section 4.3.1. Figure 5.11 shows the method we will work on. For simplicity, we have stripped it down to code that creates a summation of the integers from 0 to 10. After the quotation syntax has been removed, we get the code seen in figure 5.12. We will again elide the package names and any inessential parts of the code.

The first line of the method can be reduced analogously to that of the previous section. The interesting

```
import uiuc.Jumbo.Util.*;
import uiuc.Jumbo.Jaemus.*;
import uiuc.Jumbo.Compiler.*;

public class IntConst {
  public static void main(java.lang.String[] argv) {
    final Code gen8464_flatten_32;
    gen8464_flatten_32 = new Code() {
      public ClosedCode eval(Environment env) {
        final NoArg gen8465_flatten_4241;
        gen8465_flatten_4241 = new NoArg(4);
        final Type flatten_28;
        flatten_28 = Type.int_type;
        final ConstVal.IntConstVal gen8466_flatten_5862;
        gen8466_flatten_5862 = new ConstVal.IntConstVal(1);
        final ClosedCode flatten_30;
        flatten_30 = new ClosedCode(gen8465_flatten_4241, env,
                                    flatten_28, gen8466_flatten_5862);
        return flatten_30;
      }
    };
    final PrintStream flatten_0;
    flatten_0 = System.out;
    flatten_0.println(gen8464_flatten_32);
  }
}
```

Figure 5.10: The final irreducible code integer constant code.

```
public class DotGenOrig {
  public static Code makeSumCode() {
    Code sumcode = $<0>$;
    for (int i = 1; i < 10; i++) {
      sumcode = $<‘Expr(sumcode) + ‘Int(i)>$;
    }
    return sumcode;
  }
}
```

Figure 5.11: A code generator for the sum from 1 to 10

```
public class DotGenOrig {
  public static Code makeSumCode() {
    Code sumcode = Constructor.integerConstant(0);
    for (int i = 1; i < 10; i++) {
      sumcode = Constructor.binOp(Instr.ADD, sumcode,
                    Constructor.constant(i));
    }
    return sumcode;
  }
}
```

Figure 5.12: The code generator of figure 5.11 with quotation expanded

```
...
final int gen8857_constval = i;
final Code gen8858_flatten_35 = new Code() {
  public ClosedCode eval(Environment env) {
    return new ClosedCode(Instr.genConst(gen8857_constval),
                          env, Type.int_type,
                          new ConstVal.IntConstVal(gen8857_constval));
  }
}
...
```

Figure 5.13: The best possible reduction of `constant(i)` (unflattened)

part is inside the loop, which is where most of the time would be spent in a dot product code generator. If the rewriting system is powerful enough, both the `constant` and the `binOp` operations can be significantly reduced.

First we inline and reduce the `constant(i)` invocation. Using the type of the argument to resolve overloading, it becomes an invocation of `integerConstant`. It inlines and reduces similarly to the previous section, except that the reduction of `Instr.genConst()` is blocked by the fact that `i` is not a static constant, but a variable. Thus the selection of the correct bytecode instruction cannot be resolved statically, and an inlined version would just take up more space in the method. The `make` call, however, can still reduce to an object creation, and we end up with the `constant()` invocation being reduced to that of figure 5.13.

Next we inline the call to `binOp`. The method contains a switch on the operator, which immediately reduces to a call to `mathOp(Instr.ADD, sumcode, gen8858_flatten_35)`, as seen in figure 5.14. That method, though, contains the bulk of the arithmetic work. While the other arithmetic operations are simple, addition in Java is overloaded based on the types of the operands. If either operand

```
int i = 1;
while (true) {
  if (i<10) {
    final Code gen8681_flatten_31;
    gen8681_flatten_31 = new Code() {
      public ClosedCode eval(Environment env) {
        return new ClosedCode(Instr.genConst(i),
                              env, Type.int_type,
                              new ConstVal.IntConstVal(i));
      }
    };
    final Code gen8683_c1 = sumcode;
    sumcode = Constructor.mathOp(6, gen8683_c1, gen8681_flatten_31);
    i++;
  } else break;
}
```

Figure 5.14: The loop after inlining `integerConstant` and `binOp`

is a string, the addition is really string concatenation. This possibility adds complexity to the method. Figure 5.15 shows just the inlined `mathOp` method after reductions based on the statically known operand have been applied.

We see that there are two methods involved: `eval()` merely calls `evalInAdd()` and handles the cleanup after a possible string concatenation (converting a `StringBuffer` into a `String`). When evaluating `evalInAdd()`, we check first to see if both operands are constant, in which case we just make a new constant. Otherwise, if we are in the middle of a string concatenation (if `cp1` is a `StringBuffer`), we simply turn `cp2` into a string and append it. However, if either operand turns out to be a string, we must make string values of them both and concatenate them using a `StringBuffer`. Finally, if we are not dealing with strings, we just convert the operands to appropriate primitive types and apply the math operator to them.

We can now see which conditions we must check to be able to reduce some of the if branches away. The types and the constant-ness of `cp1` and `cp2` are checked, so they are essential targets for reductions. There is no problem in inlining `cp2`, as it is defined just above, and we will be able to tell both its type (`Type.int_type`) and whether it is constant (it is). However, the definition of `cp1` is not unique — it comes from `sumcode`, which can either be defined before the loop or in a previous iteration of the loop.

With the information we know will be available, we can remove only parts of the conditions of the if statements, but we cannot reduce away any of the if statements themselves.

```
...
final Code gen8859_c1 = sumcode;
sumcode = new Code() {
  public ClosedCode evalInAdd(Environment env) {
    final ClosedCode cp1 = gen8859_c1.evalInAdd(env);
    final ClosedCode cp2 = gen8858_flatten_35.eval(env);
    if (cp1.isConstant()&&cp2.isConstant()) {
      final ConstVal c = cp1.constantValue().mathBinOp(6, cp2.constantValue());
      return new ClosedCode(c.genLoadCode(), env, c.getType(), c);
    } else {
      if (cp1.type.isObjectType("java/lang/StringBuffer")) {
        final ClosedCode str2 = Constructor.makeStringValue(cp2);
        return new ClosedCode(cp1.bytecode.append(str2.bytecode),
                              env, cp1.type);
      } else {
        if (cp1.type.isObjectType("java/lang/String") ||
            cp2.type.isObjectType("java/lang/String")) {
          final ClosedCode str1 = Constructor.makeStringValue(cp1);
          final ClosedCode str2 = Constructor.makeStringValue(cp2);
          return new ClosedCode(Instr.genNew("java/lang/StringBuffer")
                                .addDup(1, 0)
                                .addInvokenonvirtual("java/lang/StringBuffer",
                                          " ", new MethodType(Type.void_type))
                                .append(str1.bytecode)
                                .append(str2.bytecode),
                               env, Type.stringbuffer_type);
        }
      }
    }
    final Type t = cp1.type.lub(cp2.type);
    return new ClosedCode(cp1.bytecode.addConvert(cp1.type, t)
                          .append(cp2.bytecode).addConvert(cp2.type, t)
                          .addMath(6, t),
                         env, t);
  }
  public ClosedCode eval(Environment env) {
    final ClosedCode cp = this.evalInAdd(env);
    if (cp.type.isObjectType("java/lang/StringBuffer")) {
      return new ClosedCode(cp.bytecode
                            .addInvokevirtual("java/lang/StringBuffer",
                                "toString", new MethodType(Type.string_type)),
                           env, Type.string_type);
    } else
      return cp;
  }
}
...
```

Figure 5.15: `mathOp` inlined and reduced for static operator ADD (unflattened, surrounding loop code elided)

By inlining `gen8858_flatten_35.eval()` and `cp2.isConstant()`, we get to the code in figure 5.16. While we could continue inlining inside the if statements, we know that those parts are not going to be evaluated, and so inlining them will have little effect except that code growth would make loading and optimizing slower. Only if we could significantly reduce the overall size of the code would we improve the way the code gets optimized in the VM.

While some amount of code has been removed due to these rewrites, the method body itself has grown, adding more work for the run-time compiler. The above code turns out to be about 2% slower than the original code when run on Sun's JVM. This is mostly due to the way HotSpot handles large methods. Running the code on the Kaffe JVM[91] shows a 26% speedup. The Kaffe JVM contains a JIT compiler that compiles each method's bytecode before execution. With such a JIT, there is no penalty for large methods, so the expanded method of figure 5.16 runs significantly faster than the original method invocations.

These two first examples have shown how the rewriting system can be used to perform optimizations, and what problems we run into when optimizing the combinators. We can inline resolvable method calls, and reduce a number of different constructs by applying simple rules. However, there are some aspects of the combinators that make them more difficult to optimize: the environments are blocked by being passed through method invocations, missing type information hinders a number of reductions, and loops make it difficult to resolve method calls. These problems should be the subject of future research.

5.4.3 Type Checking

An alternative use of the rewriters is to check whether a piece of quoted code is type safe. The combinators check for type safety as part of the compilation process, in order to handle type conversion correctly. If the types do not match, an exception is thrown. If we can reduce the code to where we are certain that an exception is thrown, we can know at compile time that a type error has occured.

This approach can be tested by reducing an expression like `$<-"a">$`, in which we use a simple integer operator on a string value. After parsing and normalizing, this turns into the code shown in figure 5.17.

The first step would be to inline the call to `stringConstant` and clean it up, getting the code shown in figure 5.18. Like the `integerConstant` calls shown in the previous sections, this reduces to creation of a four-tuple, including the type, which is the predefined type for strings. Since we are mainly interested in the type, we will stop the inlining of `stringConstant` here.

```
public ClosedCode evalInAdd(Environment env) {
  final ClosedCode cp1 = gen8858_c1.evalInAdd(env);
  final Instr gen8862_flatten_30 = Instr.genConst(gen8857_constval);
  final Type gen8862_flatten_31 = Type.int_type;
  final ClosedCode gen8862_flatten_33 =
     new ClosedCode(gen8862_flatten_30, env, gen8862_flatten_31,
                    new ConstVal.IntConstVal(gen8857_constval));
  if (cp1.isConstant()) {
    final ConstVal c = cp1.constantValue()
                          .mathBinOp(6, gen8862_flatten_33.constantValue());
    return new ClosedCode(c.genLoadCode(), env, c.getType(), c);
  } else {
    if (cp1.type.isObjectType("java/lang/StringBuffer")) {
      final ClosedCode str2 = Constructor.makeStringValue(gen8862_flatten_33);
      return new ClosedCode(cp1.bytecode.append(str2.bytecode), env, cp1.type);
    } else {
      if (cp1.type.isObjectType("java/lang/String") ||
          gen8862_flatten_31.isObjectType("java/lang/String")) {
        final ClosedCode str1 = Constructor.makeStringValue(cp1);
        final ClosedCode str2 = Constructor.makeStringValue(gen8862_flatten_33);
        return new ClosedCode(Instr.genNew("java/lang/StringBuffer")
                              .addDup(1, 0)
                              .addInvokenonvirtual("java/lang/StringBuffer", " ",
                                                   new MethodType(Type.void_type))
                              .append(str1.bytecode).append(str2.bytecode),
                             env, Type.stringbuffer_type);
      }
    }
  }
  final Type t = cp1.type.lub(gen8862_flatten_31);
  return new ClosedCode(cp1.bytecode.addConvert(cp1.type, t)
                        .append(gen8862_flatten_30)
                        .addConvert(gen8862_flatten_31, t)
                        .addMath(6, t), env, t);
}
public ClosedCode eval(Environment env) {
  final ClosedCode cp = this.evalInAdd(env);
  if (cp.type.isObjectType("java/lang/StringBuffer")) {
    return new ClosedCode(cp.bytecode
                          .addInvokevirtual("java/lang/StringBuffer", "toString",
                                            new MethodType(Type.string_type)),
                          env, Type.string_type);
  } else
    return cp;
}
```

Figure 5.16: After inlining `eval()` and `isConstant()` calls, further reductions are pointless.

```
Code c;
final Code flatten_0;
flatten_0 = Constructor.stringConstant("a");
c = Constructor.negate(flatten_0);
```

Figure 5.17: Code for the illegally-typed expression `$<-"a">$`

```
Code c;
final Code gen8466_flatten_101;
gen8466_flatten_101 = new Code() {
  public ClosedCode eval(Environment env) {
    final uiuc.Jumbo.Jaemus.Instr flatten_95;
    flatten_95 = uiuc.Jumbo.Jaemus.Instr.genConst("a");
    final uiuc.Jumbo.Jaemus.Type flatten_96;
    flatten_96 = uiuc.Jumbo.Jaemus.Type.string_type;
    final ConstVal flatten_97;
    flatten_97 = ConstVal.make("a");
    final ClosedCode flatten_98;
    flatten_98 = new ClosedCode(flatten_95, env, flatten_96, flatten_97);
    return flatten_98;
  }
};
c = Constructor.negate(gen8466_flatten_101);
```

Figure 5.18: The code of figure 5.17 after `stringConstant` has been inlined

Next, we inline the call to `negate` and clean that up, getting the code shown in figure 5.19. This shows the exception we are looking for, which gets thrown if the operand is not numeric. Not shown is the code that actually creates the code. The variable guarding the if with the throw is `flatten_163`, which is derived from the call to `flatten_161.isNumeric()`. `flatten_161`, in turn, derives from the call to `gen8466_flatten_101.eval()`. Since that is a call on a known object, we can perform the inlining of `eval`.

Inlining `eval` and cleaning up gives us the code in figure 5.20. The body of the string-generating object is completely inlined, and the type is now directly used in the call to `is_numeric`. The `ClosedCode` object cannot be reduced yet, as it is used in parts further down.

The `isNumeric` method simply returns false when given a string type, so in figure 5.21 we see that `flatten_163` became true, and the true-branch of the if is now on the main flow path. Since the method now always reaches a throw statement, we can conclude that there is a compilation error.

This can in principle be extended to find any compilation error statically, but is in practice limited by two factors: Fragmentation and complexity. Any occurrence of an antiquote in a code fragment will be

```
Code c;
final Code gen8466_flatten_101;
gen8466_flatten_101 = new Code() {...};
final Code gen8465_flatten_176;
gen8465_flatten_176 = new Code() {
  public ClosedCode eval(Environment env) {
    final ClosedCode c1;
    c1 = gen8466_flatten_101.eval(env);
    final uiuc.Jumbo.Jaemus.Type flatten_161;
    flatten_161 = c1.type;
    final boolean flatten_162;
    flatten_162 = flatten_161.isNumeric();
    final boolean flatten_163;
    flatten_163 = !flatten_162;
    if (flatten_163) {
      final uiuc.Jumbo.Jaemus.Type flatten_158;
      flatten_158 = c1.type;
      final java.lang.String flatten_159;
      flatten_159 = ("Negating non-numeric type "+flatten_158);
      final java.lang.Error flatten_160;
      flatten_160 = Constructor.error(flatten_159, env);
      throw flatten_160;
    }
    ...
  }
};
c = gen8465_flatten_176;
```

Figure 5.19: The code of figure 5.18 after `negate` has been inlined (some parts elided)

impossible to check fully. A few antiquotes, namely those that lift constant values, can be checked, but the majority found in actual use, namely names, expressions and statements, give no useful information for this kind of correctness checker.

Similarly, use of non-hygienic variables and accesses of new fields, methods and classes can make it impossible to find the type of an expression, thus making rewriter-based correctness checking less likely to work.

Even if the appropriate types are available for correctness checking, there remains the problem of whether these type checks are feasible for more complex code. As can be seen in these examples, the inlining process can cause significant growth in code size. The final code size does not matter for correctness checks, as the expanded code will be discarded once checking is finished. However, large intermediate results may strain the time and memory constraints, making correctness checks only applicable for small code fragments.

```
...
Environment gen8467_env;
gen8467_env = env;
final uiuc.Jumbo.Jaemus.Instr gen8467_flatten_95;
gen8467_flatten_95 = uiuc.Jumbo.Jaemus.Instr.genConst("a");
final uiuc.Jumbo.Jaemus.Type gen8467_flatten_96;
gen8467_flatten_96 = uiuc.Jumbo.Jaemus.Type.string_type;
final ConstVal gen8467_flatten_97;
gen8467_flatten_97 = ConstVal.make("a");
final ClosedCode gen8467_flatten_98;
gen8467_flatten_98 = new ClosedCode(gen8467_flatten_95, gen8467_env,
                                    gen8467_flatten_96, gen8467_flatten_97);

final boolean flatten_162;
flatten_162 = gen8467_flatten_96.isNumeric();
final boolean flatten_163;
flatten_163 = !flatten_162;
if (flatten_163) {
  final java.lang.String flatten_159;
  flatten_159 = ("Negating non-numeric type "+gen8467_flatten_96);
  final java.lang.Error flatten_160;
  flatten_160 = Constructor.error(flatten_159, env);
  throw flatten_160;
}
...
```

Figure 5.20: The code of figure 5.19 after `eval` has been inlined (some parts elided)

```
...
uiuc.Jumbo.Jaemus.Instr.genConst("a");
final uiuc.Jumbo.Jaemus.Type gen8467_flatten_96;
gen8467_flatten_96 = uiuc.Jumbo.Jaemus.Type.string_type;
ConstVal.make("a");
final java.lang.String flatten_159;
flatten_159 = ("Negating non-numeric type "+gen8467_flatten_96);
final java.lang.Error flatten_160;
flatten_160 = Constructor.error(flatten_159, env);
throw flatten_160;
...
```

Figure 5.21: The code of figure 5.20 after `isNumeric` has been inlined (some parts elided)

The AST part of the Jumbo parser can determine the type of any expression, and could be made to check if there are type errors in the code. However, using that approach would require building an additional system. The rewriters, because of the structure of the compiler, can be made to do typechecking with a minimal amount of effort.

5.5 Conclusion

In order to make code generation applicable in more situations, we need to lower the cross-over point. This can be done either by generating better code or by generating the code faster. Optimizing the generated code is limited by time constraints, and it is difficult to optimize normal Java code anyway. Instead it seems more promising to try to optimize the code generation step.

We have considered several different ways to speed up code generation. We could try to use a partial evaluator, but none is available that can handle the combinators. We could try to make the static portions of the quoted faster through abstract evaluation. However, abstract evaluation seems likely to be blocked by holes in the code. Instead, we have chosen to implement a source rewriting system that can take advantage of the regular style of the combinators to perform structural specialization.

In this section, we have shown how our rewriting system can be applied for both optimizating and type checking. The idea of using inlining and simple rewrite rules has been shown to work, though structural obstacles limit applicability. Significant optimizations were not achieved on the HotSpot JVM, but with the Kaffe JVM, a speedup of 26% was achieved for code generation of a simple unrolling of a loop.

The main roadblocks of further reductions are lack of type information, loss of environment information, and the inability to inline ambiguously defined methods. Types of holes and irreducible combinators limit what reductions can be done based on type information. Similarly, environment information is lost if we cannot tell when an environment is passed through a combinator unchanged. In both cases, future research may make it possible to analyse the combinators to deduce this information. Ambiguously defined methods, i.e., methods invoked on a variable whose value may come from several places, could also be the subject of future research. It should be possible to inline the method when the exact class or method definition can be determined.

The idea of exploiting the compositional nature of the Jumbo compiler to optimize the code generators has shown to be viable. Because the code fragments are stored in a way that carries meaning, a general-

purpose optimizer can statically optimize the code generation of the separate code fragments. In the next chapter, we give details on implementation of the rewriting system.

Chapter 6

The Rewriting System

6.1 Introduction

As explained in the previous chapter, we have focused on optimizing the code generators, as it seems to be the approach most likely to lower the break-even point for code generation. In this chapter we present the source-to-source rewriting system we have developed to optimize the code generators. The rewriting is based on a normalization process and a set of analyses, as described in sections 6.2 and 6.3. The normalization is performed by a set of rewrite rules called normalizers, which do not depend on the analyses, but massage the code into a simpler form. The analyses provide use-def information for variables and fields, which in turn allow us to perform inlining, loop unrolling, and a number of code reducing rewrite rules, as described in sections 6.4, 6.5, and 6.6. Section 6.7 describes how to use the rewriting system. In section 6.8, we examine the effect of the rewriting system and its limitations, which point the way to further research.

In a rewriting system[71, 20], a series of rewrite rules ("rewriters") are applied to some structured data, each changing the structure slightly when applicable. By repeated application, the structure is transformed into the desired state. If the rewriters are all known to make some progress towards the desired goal when applied, they can be automatically applied until no rewriter applies. The progress requirement implies that the rewriters do not perform the identity transformation, that they cannot undo each other, and that no set of rules can be applied indefinitely. The first two implications are easy to verify. The requirement of finiteness is equivalent to showing that a program halts. However, if we can show that some measure of the size of the data does not grow (or cannot grow without bound), there is necessarily a limit to the number of application that can be performed. Note that rewriters that are not applied automatically, or are not applied repeatedly, do not need to make progress.

For each rewriter, including the normalizers, we present a formalized description of its actions. A rule description of the form

Algebraic $[\![\text{BINARY}(e_1, o, e_2)]\!] \Rightarrow \quad e_2$
if $\quad e_1 = \text{LITERAL}(0)$,
$\quad\quad o =$ "plus"

reads: Rewriter "Algebraic" transforms "binary" nodes in the AST containing the two subexpressions e_1 and e_2 and operand o to e_2 if e_1 is the literal 0 and the operand o is the "plus" operand. A rewriter may be described by several rules. If more than one rule can apply to a node, the first rule is picked.

Except where otherwise indicated, metavariables are of the following types:

$n \in$ AST Nodes

$e \in$ AST Expressions, i.e. nodes with a type

$v \in$ AST Expressions representing local variables

$f \in$ Field names

$c \in$ Classes or class names

$a \in$ AST Expressions representing method invocation arguments

$p \in$ Method parameters

$s \in$ Statements

$m \in$ Methods

$t \in$ Types

The Cons operator that prepends an item to a list is denoted ::, while a single colon $a : t$ is used to indicate that a is of type t. We will omit the LITERAL() marking when it can be inferred from context.

The following functions are used throughout the formal descriptions. The Defs and Uses functions use information provided by the use-def analysis. The Target function uses information provided by the flow analysis. These three functions are not available to the normalizers, which operate independently of the

analyses. The remaining functions are handled by examining the AST, and do not require the analyses. Except for TypeOf, the functions return AST nodes or collections of AST nodes.

Node Decl(v) = the declaration of a variable (a VarDecl or Parameter node).

Set Defs(v) = the set of definitions of a variable, i.e. assignment, method parameter or increment expression nodes. When applied to a specific use of a variable, the set of definitions that may reach the use.

Set Uses(v) = the set of uses of a variable, i.e. Var nodes. When applied to a specific definition of a variable, the set of uses that may be reached by that definition.

Node Target(s) = the target of a break or continue statement, i.e. either a labeled statement list, a while statement or a break statement.

Set Statements(m) = the set of all statement nodes in a method, including sub-statements.

List Parents(n) = the list of ancestors of a node in the AST.

List Constructors($c, a_1, \ldots, a_n$) = the list of constructors that may be invoked by a constructor call `new` c`(`$a_1, \ldots, a_n$`)`, i.e., the constructor invoked, the constructor it in turn invokes, up to the constructor for `Object`.

Type TypeOf(e) = the result type of e, in particular the return type of methods.

While the notation $\{e_1, \ldots, e_n\}$ represents a set, $[e_1, \ldots, e_n]$ represents an ordered list, possibly with duplicates. The rewriters will be distinguished from auxiliary functions by their arguments being surrounded by ⟦double square brackets⟧. Nodes in the AST are indicated by the use of small caps. For instance, the (fictitious) rewriter shown below removes assignments that are being immediately overwritten, but only if the expression being removed is some literal.

Overwritten $\llbracket \textsc{Statements}(s_1, \ldots, s_n, \textsc{Assignment}(v, e_1), \textsc{Assignment}(v, e_2), s_{n+1}, \ldots, s_m) \rrbracket \Rightarrow$
$\textsc{Statements}(s_1, \ldots, s_n, \textsc{Assignment}(v, e_2), s_{n+1}, \ldots, s_m)$
if $e_1 = \textsc{Literal}(\_)$

The AST nodes whose meaning or arguments are not straightforward are described below. Note that for legibility, some of the names here are shorter than those used in the code.

ANONYMOUSCLASS($c_s, a_1, \ldots, a_n, b$): An anonymous class creation expression, with c_s being the super class and b the body. In Java: `new` c_s`(`$a_1, \ldots, a_n$`) { `b` }`.

CLASS(c, c_s, b): A non-anonymous class definition, with c_s being the super class and b the body. In Java: `class` c `extends` c_s `{` b `}`.

CONSTRUCTOR($p_1, \ldots, p_n, s, b$): A constructor declaration with parameters $p_1, \ldots, p_n$, explicit super constructor call s, and body b.

EXPRESSIONSTATEMENT(e): An expression at statement level. The only expressions allowed inside EXPRESSIONSTATEMENT() are assignments, method invocations, object constructions and increment/decrement expressions.

FIELDACCESS(t, f): An access to a field f on the target object t. t may be a string constant, a object-valued expression, or ϵ to indicate the `this` object.

FQCN($n_1, \ldots, n_k$): A fully qualified class name, i.e. a dotted list $n_1.\cdots.n_k$ that resolves to a class name. These are created by the FQCN normalizer. Can also be FQCN(c) to indicate the fully qualified name of the given class.

LABEL(l, s): A statement s with label l.

LITERAL(): Any simple constant. In many cases, we leave out the AST indicator when talking about a specific constant, e.g. ASSIGN($v, 0$) assigns the integer literal 0 to v.

METHODINVOCATION($e_t, m, a_1, \ldots, a_n$): An invocation of method m on target e_t with arguments $a_1, \ldots, a_n$. t may be a string constant, a object-valued expression, or ϵ to indicate the `this` object.

NEWOBJECT($c, a_1, \ldots, a_n$): An object creation expression `new` c`(`$a_1, \ldots, a_n$`)`.

QUALIFIEDNAME($n_1, \ldots, n_k$): A dotted list n_1`.`$\cdots$`.`n_k that cannot be a class name only, i.e. it must be a simple variable or a field lookup.

QUALIFIEDNAMEORCLASS(): A dotted list n_1`.`$\cdots$`.`n_k that may be a class name. It is a subclass of QUALIFIEDNAME(), that can only occur as the target of a FIELDACCESS() or METHODINVOCATION().

SCOPEDSTATEMENTS($s_1, \ldots, s_n$): A list of statements that defines a scope. Variables defined inside the scope are not visible outside.

STATEMENTS($s_1, \ldots, s_n$): A list of statements that does not define a scope.

SWITCHCASE($e_1, \ldots, e_n, s$): A part of a switch statement containing several `case` e_i`:` labels (labels in Java are compile-time constant expressions) and one statement.

THIS(c): A `this` expression. THIS() is the unqualified `this`, THIS(c) is a qualified c`.this` as used in inner classes to refer to outer classes.

UNARY(o, e): A unary expression, with o being the operator.

VARDECL($(t_1, v_1, e_1), \ldots, (t_n, v_n, e_n)$): A local variable declaration statement with several variables being defined and optionally initialized. After flattening (described below), only one variable is defined per VARDECL(), and with no initializer.

6.2 Normalizing

Java has a number of syntactic shortcuts that allow for a less cumbersome programming style. Thus, there are sometimes several distinct ways to write the same thing. While such redundancy is good for the programmer, it makes it harder to analyze the code. A simple predicate such as "is statement A the same as statement B?" becomes harder to answer correctly. Figure 6.1 shows an example of how the same expression can take different forms. This section describes the *normalization* process that we apply to all loaded code in order to simplify the rewriters.

Normalization is achieved by means of applying a set of *normalizers*, rewriters that transform the code into a normal form. This normal form is not "strong" in the sense that two different expressions in normal form must represent different calculations, but it allows simpler assumptions for the rewriters and decodes some of the more complex parts of Java. Note that the normalizers are only applied once to each node, so they need not make progress in the sense described in section 6.1. In particular, some of the normalizers allow the identity transformation or increase the size of the code.

The normal form created by the normalizers has the following characteristics:

1. All class names are fully expanded with their package name and outer class names.

```
package Even.More;
class Redundant {
  static int staticmethod() { return 42; }
  void redundancies() {
    staticmethod();
    this.staticmethod();
    Redundant.staticmethod();
    Redundant.this.staticmethod();
    Even.More.Redundant.staticmethod();
    Even.More.Redundant.this.staticmethod();
  }
}
```

Figure 6.1: Six different ways to make the exact same method invocation.

2. All QualifiedNameExpression nodes (dotted lists) are either simple variables or fully qualified class names, not field accesses. Variables and fully qualified class names are represented as subclasses of QualifiedNameExpression, while field accesses are explicitly represented with FieldAccess nodes.

3. All field lookups and method invocations have an explicit target.

4. No `this` expression referring to the innermost class is qualified with a class name. All `this` expressions referring to non-innermost classes are qualified with a fully qualified class name.

5. No do- or for-loops exist.

6. No expression contains a subexpression that is not a fully qualified name, a literal, a variable, or a `this`-expression. The two exceptions are that assignment expressions may contain any expression as either their left-hand side or their right-hand side, but not on both sides, and that the target of a field access or method invocation may be an anonymous class creation expression.

7. No statement other than ExpressionStatement directly contains expressions other than literals, variables or `this`-expressions.

8. No variable declaration declares more than one variable.

9. No variable declaration contains initializers.

The `this`-expressions are kept inside expressions by the flattener because anonymous inner classes do not have a name for their type. Thus assigning an anonymous `this` would coerce its type to that of the superclass, removing access to new fields or methods declared in the anonymous class.

In order to reduce the complexity of the later normalizers, we assume that any classes referenced are at least normalized up to the previous step of normalization. To enforce this invariant in the presence of circular references, we use a pipeline of classes being normalized. At the start of the pipeline, classes are loaded from source files. At the end, the fully normalized classes are cached in a class table for later lookup. As invariants, the pipeline is empty whenever a method outside the normalizers asks for a class, and the previous steps of the pipeline are empty when we perform any normalization rewriting.

Before starting to normalize a class, we know that no previous steps have classes that need to be normalized. Whenever during normalization we refer to a class that is not already loaded, we load the corresponding source file and insert any classes in it into the start of the pipeline. We then call the normalizers recursively for the steps of the pipeline prior to the current one. Once that call returns, the pipeline before the current step is empty, and all classes are normalized at least up to the current step.

Some files, notably the standard `java.lang` files, are not available as source, and thus cannot be normalized. Instead, we wrap these files in a thin object that implements a common lookup interface, and move them directly from the loader into the class table. Since no source is available, the flattening and syntactic sugar normalizers are irrelevant. Additionally, class names are always fully qualified in class files and so do not need to be expanded. Compiled classes are necessary for determining the types and flags of methods and fields, but cannot be used for inlining. Decompiling compiled classes to allow inlining is outside the scope of this project; the interested reader can use any of a number of available decompilers to obtain source if so desired.

6.2.1 FQCN

One of the most complex parts of parsing Java is *fully qualified class names* (FQCNs) and their resemblance to field lookup. A dotted list of (one or more) identifiers can be either a local variable optionally followed by field lookups, a series of field lookups, or a class name (with or without its package) optionally followed by field lookups. Since package names and inner class names are separated by dots as well, the dotted list cannot be properly understood until we can perform class lookup to find fields. Finding fields in turn is complicated by the requirement to find fields in superclasses and in outer classes, and by the fact that inner fields can shadow outer local variables. The parser, not having access to superclasses, has no way to discern the various meanings of a dotted list.

To avoid this complication while doing complex rewritings of the code, the first normalizer decodes dotted lists into field lookups. To do this, we must first check if the first part of the dotted list is a variable or field, in this or enclosing classes. If not, it must be a class name or part thereof, possibly followed by some field lookups. If the dotted list occurs before a method call, the entire dotted list may be a class name (the dotted list is then known in the AST as a QualifiedNameOrClass), otherwise it must at least end in a field lookup (known in the AST as a QualifiedName)[1].

The FQCN normalizer, once it has decoded the dotted list, transforms it into a fully qualified class name, a variable name or a `this` expression, possibly followed by a series of field lookups. This satisfies parts 1, 2, and 3 of the requirements for the normal form. Additionally, the FQCN normalizer resolves THIS() nodes to satisfy 4.

The FQCN normalizer can be described by the following helper functions and rules:

isField$(n, e) =$ false, if n is a local variable or parameter in the method containing e
true, if n is a field of the class containing e or any of its superclasses
isField(n, e'), if the (anonymous) class containing e is within the expression e'
false otherwise

isVariable$(n, e) =$ true, if n is a local variable or parameter in the method containing e
false, if n is a field of the class containing e or any of its superclasses
isVariable(n, e'), if the (anonymous) class containing e is within the expression e'
false otherwise

FQCN $[\![e]\!] \Rightarrow$ FIELDACCESS$(\ldots$FIELDACCESS$($VARIABLE$(n_1), n_2), \ldots n_i)$
if $e \in \{$QUALIFIEDNAME$(n_1, \ldots, n_i),$ QUALIFIEDNAMEORCLASS$(n_1, \ldots, n_i)\}$,
isVariable(n_1, e)

FQCN $[\![e]\!] \Rightarrow$ FIELDACCESS$(\ldots$FIELDACCESS$($THIS$(), n_1) \ldots, n_i)$
if $e \in \{$QUALIFIEDNAME$(n_1, \ldots, n_i),$ QUALIFIEDNAMEORCLASS$(n_1, \ldots, n_i)\}$,
isField(n_1, e),
n_1 is defined in the innermost class

[1] Since the parser cannot determine what part of a dotted list is a field lookup, a dotted list cannot possibly occur as the target of a field lookup until after this normalizer is finished.

FQCN $[\![e]\!] \Rightarrow$ FIELDACCESS(... FIELDACCESS(THIS(c), n_1) ..., n_i)

if $e \in \{$QUALIFIEDNAME($n_1, \ldots, n_i$), QUALIFIEDNAMEORCLASS($n_1, \ldots, n_i$)$\}$,

isField(n_1, e),

n_1 is defined in the outer class c

FQCN $[\![e]\!] \Rightarrow$ FIELDACCESS(... FIELDACCESS(FQCN($n_1, \ldots, n_k$), n_{k+1}) ..., n_i)

if $e \in \{$QUALIFIEDNAME($n_1, \ldots, n_i$), QUALIFIEDNAMEORCLASS($n_1, \ldots, n_i$)$\}$,

$\neg$isVariable(n_1, e),

$\neg$isField(n_1, e),

$k < i$,

n_{k+1} is a static field of the class designated by $n_1.\cdots.n_k \wedge$

$\forall j < k : n_{j+1}$ is not a static field of the class designated by $n_1.\cdots.n_j$

FQCN $[\![e]\!] \Rightarrow$ FQCN($n_1, \ldots, n_i$)

if $e =$ QUALIFIEDNAMEORCLASS($n_1, \ldots, n_i$),

$\neg$isVariable(n_1, e),

$\neg$isField(n_1, e),

$\forall j < i : n_{j+1}$ is not a static field on the class designated by $n_1.\cdots.n_j$

FQCN $[\![$THIS(c)$]\!] \Rightarrow$ THIS()

if c is the directly enclosing class

For the purpose of the remaining parts of the FQCN rewriter, we let TARGETABLE(t, X) mean either FIELDACCESS(t, X) or METHODINVOCATION(t, X), where X is respectively a field name or a method name and arguments.

FQCN $[\![$TARGETABLE($t \in \{\epsilon,$ THIS()$\}, X$)$]\!] \Rightarrow$ TARGETABLE(FQCN(c), X)

if c is the innermost class defining X,

X is statically defined on c

FQCN $[\![$TARGETABLE($t \in \{$QUALIFIEDNAME(c), THIS(c)$\}, X$)$]\!] \Rightarrow$ TARGETABLE(FQCN(c'), X)

if c' is the innermost class not inside c (but possibly including) defining X,

X is statically defined on c'

FQCN $[\![\text{TARGETABLE}(t \in \{\epsilon\}, X)]\!] \Rightarrow \quad \text{TARGETABLE}(\text{THIS}(), X)$

if X is non-statically defined on the directly enclosing class

FQCN $[\![\text{TARGETABLE}(t \in \{\epsilon, \text{THIS}()\}, X)]\!] \Rightarrow \quad \text{TARGETABLE}(\text{THIS}(c), X)$

if c is the innermost class defining X,

X is non-statically defined on c, and c is not the innermost class enclosing X

FQCN $[\![\text{TARGETABLE}(\text{THIS}(c), X)]\!] \Rightarrow \quad \text{TARGETABLE}(\text{THIS}(c'), X)$

if c' is the innermost class not inside c defining X,

X is non-statically defined on c'

```
package FQCN;
public class FQCNTest {
  int x;
  static int y;
  int getSum() {
    return x+y;
  }
  public static void
    main(String[] argv) {
    FQCNTest t = new FQCNTest();
    System.out.println(t.getSum());
  }
}
```

(a) Code before FQCN rewriting

```
package FQCN;
public class FQCNTest {
  int x;
  static int y;
  int getSum() {
    return this.x+FQCN.FQCNTest.y;
  }
  public static void
    main(java.lang.String[] argv) {
    FQCN.FQCNTest t =
      new FQCN.FQCNTest();
    java.lang.System.
      out.println(t.getSum());
  }
}
```

(b) Code after FQCN rewriting.

Figure 6.2: The effect of FQCN rewriting.

Figure 6.2 shows some of the transformations the FQCN rewriter performs: Package names, static targets and `this` invocations are made explicit.

6.2.2 ForWhile and DoWhile

To reduce the number of cases being handled by other rewriters, these two rewriters transform for-loops and do-loops into while-loops. The transformation of do-loops creates a copy of the body of the loop, but we do not consider this copying a severe problem. Since the original and copied body are in separate scopes, the

variables do not clash. These normalizers satisfy requirement 5 for the normal form. Figure 6.3 shows an example of the transformation performed by the DoWhile rewriter.

ForWhile $[\![\textsc{For}(s_i, e_t, s_u, s_b)]\!] \Rightarrow \quad \textsc{ScopedStatements}(s_i, \textsc{While}(e_t, \textsc{Statements}(s_b, s_u)))$

DoWhile $[\![\textsc{Do}(s_b, e_t)]\!] \Rightarrow \quad \textsc{Statements}(\textsc{ScopedStatements}(s_b), \textsc{While}(e_t, s_b))$

```
public class DoWhile {
  int sumTo(int x) {
    int sum = 0;
    do {
      sum += x;
      x--;
    } while (x > 0);
    return sum;
  }
}
```

(a) Code before DoWhile normalization

```
public class DoWhile {
  int sumTo(int x) {
    int sum = 0;
    {
      sum += x;
      x--;
    }
    while (x>0) {
      sum += x;
      x--;
    }
    return sum;
  }
}
```

(b) Code after DoWhile normalization

Figure 6.3: The effects of DoWhile normalization

6.2.3 Flattening

The flattening rewriter performs two main actions: It extracts complex expressions from statements like `if` and `throw`, replacing them with variable accesses, and it breaks complex expressions in assignments and method invocations into smaller expressions in a similar way. Taking expressions out of statements is simple, except for the test expression in the `while` statement, which must be evaluated inside the loop. Rather than placing the flattened while test in front of the `while` statement, we move it inside and test it to see if we should break out of the loop.

Breaking complex expressions into simple expressions requires assigning the value of each subexpression to a temporary variable. In order to do this, we must be able to find the types of the subexpressions. Three kinds of expressions cannot be fully flattened out: literals (since the "null" literal has no proper type), "this" expressions (since it has no named type inside anonymous classes), and anonymous classes being used

as targets of method invocations or field lookups (since anonymous classes have no named type)[2]. Therefore, those expressions are left in place. Variables, while technically expressions themselves, are needed as part of the flattened expressions, and so are left in place as well. For the remaining expressions, we assign their value to a newly generated variable of the appropriate type and replace the expression with a reference to that variable.

First we define some helper functions. NeedFlatten checks whether a given expression needs to be flattened. Flatten1 performs the actual flattening of one expression, if necessary. It declares a fresh variable and assigns the value of the recursively flattened expression to the variable. The declarations and assignments are stored in a STATEMENTS() being passed along, and the expression returned is a lookup of the fresh variable. FlattenExpr flattens all subexpressions in an expression, again storing new statements in a statement list passed along. In particular, FlattenExpr replaces the conditional (`?:`) expression with an if statement. This normalizer satisfies requirement 6 for the normal form.

$$\text{NeedFlatten}(e) = \begin{array}{ll} \text{false, if } e \in \{\text{FQCN}(\_), \text{VAR}(\_), \text{THIS}(\_), \text{LITERAL}(\_)\} \\ \text{true otherwise} \end{array}$$

$$\text{Flatten1}(e, s) =$$
$$(v, s' :: [\text{VARDECL}(t, v), \text{ASSIGN}(v, e')]), \text{if} \quad \begin{array}{l} \text{NeedFlatten}(e) \wedge (e', s') = \text{FlattenExpr}(e, s) \wedge \\ t = \text{TypeOf}(e) \wedge v \text{ is fresh.} \end{array}$$
$$(e, s) \text{ otherwise}$$

$$\text{FlattenExpr}(\text{CONDITIONAL}(e_1, e_2, e_3), s) = (v, s'')$$
$$\begin{array}{l} \text{if } \; t = \text{TypeOf}(e_2) \sqcup \text{TypeOf}(e_3), \\ \quad v \text{ is fresh}, \\ \quad (e_1', s') = \text{Flatten1}(e_1, s), \\ \quad (e_2', s_2) = \text{Flatten1}(e_2, []), \\ \quad (e_3', s_3) = \text{Flatten1}(e_3, []), \\ \quad s'' = s' :: [\text{VARDECL}(t, v), \text{IF}(e_1', s_2 :: [\text{ASSIGN}(v, e_2')], s_3 :: [\text{ASSIGN}(v, e_3')])] \end{array}$$

[2] An anonymous class used in other than method invocations or field lookups do not need any other type than that of its superclass, as fields and methods not present in its (named) superclass can only be accessed directly after creation.

FlattenExpr(FIELDACCESS(ANONYMOUSCLASS($c, a_1, \ldots, a_n$), f)) =

s_n :: [FIELDACCESS(ANONYMOUSCLASS($c, a'_1, \ldots, a'_n$), f)]

if (a'_1, s_1) = Flatten1(a_1, []),

(a'_2, s_2) = Flatten1(a_2, s_1),

...

(a'_n, s_n) = Flatten1(a_n, s_{n-1})

FlattenExpr(METHODINVOCATION(ANONYMOUSCLASS($c, a_1, \ldots, a_n$), $m, A_1, \ldots, A_k$)) =

s_{n+k} :: [METHODINVOCATION(ANONYMOUSCLASS($c, a'_1, \ldots, a'_n$), $m, A'_1, \ldots, A'_k$)]

if (a'_1, s_1) = Flatten1(a_1, []),

(a'_2, s_2) = Flatten1(a_2, s_1),

...

(a'_n, s_n) = Flatten1(a_n, s_{n-1}),

(A'_1, s_{n+1}) = Flatten1(A_1, s_n),

...

(A'_n, s_{n+k}) = Flatten1(A_n, s_{n+k-1})

For any other expression (EXPRESSION$_x$($e_1, \ldots, e_n$)), we simply flatten any subexpressions.

FlattenExpr(EXPRESSION$_x$($e_1, \ldots, e_n$), s) = (EXPRESSION$_x$($e'_1, \ldots, e'_n$), s'_n)

if (e'_1, s'_1) = Flatten1(e_1, s),

(e'_2, s'_2) = Flatten1(e_2, s'_1),

...

(e'_n, s'_n) = Flatten1(e_n, s'_{n-1})

Now that we can flatten expressions, we merely need to pick those statements that directly contain expressions and flatten those expressions. Statements within statements (such as statement lists inside if statements) are found by the AST traversal and need not be flattened. We have the distinction between statement-level traversal and expression traversal in order to ensure that new statements created by flattening an expression are placed immediately before the expression.

All new variables created as part of the flattening process are declared final. Final variables are easier to reason about, as they cannot change their values.

The first part of the flattener fulfills requirements 8 and 9 for the normal form by splitting variable declarations into one declaration per variable and putting any initial assignment into a separate statement.

Flattening $[\![\textsc{VarDecl}((t_1, v_1, e_1), \ldots, (t_n, v_n, e_n))]\!] \Rightarrow \quad s$

if $(e'_1, s'_1) = \text{FlattenExpr}(e_1, [])$,

$s_1 = s'_1 :: [\textsc{VarDecl}(t_1, v_1, null), \textsc{Assign}(v_1, e'_1)]$,

$\vdots$

$(e'_n, s'_n) = \text{FlattenExpr}(e_n, [])$,

$s_n = [\textsc{VarDecl}(t_n, v_n, null), \textsc{Assign}(v_n, e'_n)]$,

$s = [s_1, \ldots, s_n]$

A while loop is flattened into an always-true loop whose body starts by calculating the value of the test. If the test fails, a break exits the loop. Placing the test inside the loop ensures that the flattened test expression is evaluated at each iteration.

Flattening $[\![\textsc{While}(e, s)]\!] \Rightarrow \quad \textsc{While}(\text{true}, s'')$

if $(e', s') = \text{Flatten}(e, [])$,

$s'' = s' :: [\textsc{If}(e', s, \textsc{Break}())]$

Flattening $[\![\textsc{If}(e, s_t, s_f)]\!] \Rightarrow \quad s' :: [\textsc{If}(e', s_t, s_f)]$

if $(e', s') = \text{Flatten1}(e, [])$

Flattening $[\![\textsc{Return}(e)]\!] \Rightarrow \quad s :: [\textsc{Return}(e')]$

if $(e', s) = \text{Flatten1}(e, [])$

Flattening $[\![\textsc{Throw}(e)]\!] \Rightarrow \quad s :: [\textsc{Throw}(e')]$

if $(e', s) = \text{Flatten1}(e, [])$

Flattening $[\![\textsc{Synchronized}(e)]\!] \Rightarrow \quad s :: [\textsc{Synchronized}(e')]$

if $(e', s) = \text{Flatten1}(e, [])$

The case labels in a switch statement must all be compile-time constant expressions. However, they may still be complicated expressions that require flattening.

$$\text{FlattenSwitchCase}(\textsc{SwitchCase}(e_1, \ldots, e_n, s_{case}), s) = (\textsc{SwitchCase}(e'_1, \ldots, e'_n, s_{case}), s_n),$$
$$\text{where} \quad (e'_1, s_1) = \text{Flatten1}(e_1, s),$$
$$\ldots,$$
$$(e'_n, s_n) = \text{Flatten1}(e_n, s_{n-1})$$

Flattening $[\![\textsc{Switch}(e, c_1 : \textsc{SwitchCase}(\_), \ldots, c_n : \textsc{SwitchCase}(\_), c_{default} : \textsc{SwitchCase}(\_))]\!] \Rightarrow$
$$s_{default} :: \left[\textsc{Switch}(e', c'_1, \ldots, c'_n, c'_{default})\right]$$
$$\text{if} \quad (e', s) = \text{Flatten1}(e, []),$$
$$(c'_1, s_1) = \text{FlattenSwitchCase}(c_1, s),$$
$$\ldots,$$
$$(c'_n, s_n) = \text{FlattenSwitchCase}(c_1, s_{n-1}),$$
$$(c'_{default}, s_{default}) = \text{FlattenSwitchCase}(c_1, s_n),$$

Assigns are handled in two different ways depending on what is on the left-hand side. If it is a field access or an array access, we flatten the left-hand side down to just the field access or array access, and the right-hand side down to a variable, literal or `this` expression. Otherwise, the left-hand side is a simple variable, and we flatten the right-hand side down to an expression with no subexpressions. After this flattening, there is only one expression inside each assignment.

Additionally, the operator-assign constructs (`+=`, `*=`, etc.) are transformed into normal assignments. Since the left-hand side of the assignment is flattened down to no more than a field or array lookup, there are no extra side-effects due to this, and the field or array lookup would have to be done twice anyway.

$$\text{FlattenOpAssign}(e_1, \text{" = "}, e_2) = \textsc{Assign}(e_1, \text{" = "}, e_2)$$
$$\text{FlattenOpAssign}(e_1, o, e_2) = \textsc{Assign}(e_1, \text{" = "}, \textsc{Binary}(e_1, o, e_2))$$

Flattening $[\![\text{EXPRESSIONSTATEMENT}(\text{ASSIGN}(e_1, o : \text{Operator}, e_2))]\!] \Rightarrow$

$s' :: [\text{EXPRESSIONSTATEMENT}(\text{FlattenOpAssign}(e'_1, o, e'_2))]$

if $e_1 = \text{FIELDACCESS}(e_f, \_)$,

NeedFlatten(e_f),

$(e'_2, s) = \text{Flatten1}(e_2, [])$,

$(e'_1, s') = \text{FlattenExpr}(e_1, s)$

Flattening $[\![\text{EXPRESSIONSTATEMENT}(\text{ASSIGN}(e_1, o : \text{Operator}, e_2))]\!] \Rightarrow$

$s' :: [\text{EXPRESSIONSTATEMENT}(\text{FlattenOpAssign}(e'_1, o, e'_2))]$

if $e_1 = \text{ARRAYACCESS}(e_a, e_i)$,

NeedFlatten$(e_a) \vee$ NeedFlatten(e_i),

$(e'_2, s) = \text{Flatten1}(e_2, [])$,

$(e'_1, s') = \text{FlattenExpr}(e_1, s)$

Flattening $[\![\text{EXPRESSIONSTATEMENT}(\text{ASSIGN}(e_1, o : \text{Operator}, e_2))]\!] \Rightarrow$

$s :: [\text{EXPRESSIONSTATEMENT}(\text{FlattenOpAssign}(e_1, o, e'_2))]$

if $(e'_2, s) = \text{Flatten1}(e_2, [])$

The only other permissible expressions at statement level are method invocations and increment/decrement expressions. These are flattened down to expressions with no subexpressions. By now, all statements are flattened to satisfy requirement 7 for the normal form.

Flattening $[\![\text{EXPRESSIONSTATEMENT}(e)]\!] \Rightarrow$

$s :: [\text{EXPRESSIONSTATEMENT}(e')]$

if $(e', s) = \text{FlattenExpr}(e, [])$

Figure 6.4 shows an example of the effect of the Flattening normalizer. Flattening obviously creates an inordinate number of new temporary variables. These variables would be easily removed by an optimizing compiler, but few Java compilers expect such code, and so just pass it on to the JVM. The JVM in turn may or may not be able to optimize this, though again it is not a style expected by the JVM implementors, so it may not be handled optimally.

```
public class Point {
  double x,y;
  final double hDist(Point p) {
    return p.x-x;
  }
  boolean isLeftOf(Point p) {
    return hDist(p) > 0;
  }
}
```

(a) Original method and invocation

```
public class Point {
  double x;
  double y;
  final double hDist(Point p) {
    {
      final double flatten_0;
      flatten_0 = p.x;
      final double flatten_1;
      flatten_1 = this.x;
      final double flatten_2;
      flatten_2 = (flatten_0-flatten_1);
      return flatten_2;
    }
  }
  boolean isLeftOf(Point p) {
    {
      final double flatten_3;
      flatten_3 = hDist(p);
      final boolean flatten_4;
      flatten_4 = (flatten_3>0);
      return flatten_4;
    }
  }
}
```

(b) Flattened method and invocation

Figure 6.4: The effect of the Flattening normalizer

Section 6.6 describes a way to undo the flattening once the rewriters are done. In extreme cases, the number of temporary variables could overflow the Java Virtual Machine's maximum of 255 variables in a method. Thus undoing the flattening can be necessary to even compile the resulting code.

6.3 Analyses

Three layers of analysis of the abstract syntax tree form the basis for the rewriters. The first layer is a simple analysis of the ways abrupt flow may occur in the program. Abrupt flow includes exceptions thrown and caught, break and continue statements, and finally clauses. Using the abrupt flow analysis, we can build a full flow analysis of the statements. This analysis links together expressions by execution flow. On top of this, the use-def analysis links together declarations, assignments and uses of local variables and method parameters. This allows us to perform various propagations and remove unused code.

The analyses are implemented using AspectJ[43] aspects to add the traversal functions and fields holding

the results to nodes. Instead of changing the files implementing the AST itself, this allows us to keep the fields and methods related to the analyses in a separate file. As an additional bonus, the error messages generated by the AspectJ compiler, `ajc`, turned out to be more informative than those of `javac`.

6.3.1 Abrupt Flow Analysis

The abrupt flow analysis is the lowest layer of analysis in the rewriting system. It simplifies the flow analysis by locating any non-local flow possible in the program.

The abrupt flow is found using a single traversal of the AST. During traversal, each node returns the set of abrupt completions that could happen during its execution and that are not handled by itself or its sub-nodes. For instance, a `break` statement always returns itself, but a `while` statement with an unlabeled `break` statement inside it does not return the break statement – it has been handled by the `while` statement. Nodes that can cause abrupt completions include `break` and `continue` statements and any statement or expression that can cause an exception to be thrown.

While the Java specification states that exceptions in Java are precise[29, section 11.3.1], following this to the letter would mean that every bytecode instruction[3] could cause an exception. This is clearly unworkable, as it would allow no code transformations to take place at all. The author has earlier argued that for purposes of program analysis, most unchecked (i.e. non-userdefined) exceptions should be ignored[10, pages 40–42]. These exceptions are the result of errors in the program, the compiler or the virtual machine, rather than just an irregular occurrence that a programmer would check for. Following this argument, the only unchecked exceptions that are considered in the abrupt flow analysis are `ArithmeticException`, `IndexOutOfBoundsException`, `SecurityException`, `ClassNotFoundException`, and `InterruptedException`. These only occur in a few, well-defined places. All user-defined exceptions must be considered.

We currently take a conservative stance on exception handlers. Rather than attempting to figure out whether an exception handler can handle all exceptions thrown within its body, we assume that any exception handler will catch some, but not necessarily all, exceptions thrown within it. Thus, abrupt flow proceeds from any place that may throw an exception both into the exception handler and out of the catch body. To improve this, we would have to remember the set of exceptions that can be thrown inside an exception

[3] With the possible exception of `Nop`

handler and remove those that the handler catches.

The abrupt flow analysis does not handle the `finally` construct yet.

During the analysis, each node that handles abrupt completions stores the set of nodes in the AST that may lead to it through abrupt completion. This stored information is then used in the flow analysis described below.

6.3.2 Flow Analysis

Given the information on non-local control flow found by the abrupt flow analysis, and the local control flow inherent in the AST nodes, we can now generate a flow graph for the program. This is done with a traversal over the AST, with a fixed-point iteration calculating the backflow of each loop construct.[4]

During the traversal, we pass a set of inflow nodes into each node visited. The node then sets up local information in the node itself and in the nodes in the inflow set, connecting the nodes with forwards and backwards flow. It then handles any flow within itself, such as in `if` or `while` statements. Finally, it generates the outflow set. In most cases, this set simply contains one node, but break statements and if-branches may add more nodes. The enclosing node uses the outflow set as the inflow set for the next node in sequence, if any. For instance, a statement list passes the outflow set from one statement as the inflow set to the next, while an `if` statement uses the outflow set from the test as the inflow set for both branches.

To reduce the number of nodes that carry flow analysis information, we chose not to perform flow analysis within expressions. This leads to a view of the program where expressions perform actions, while statements are mostly abstracted into the control flow graph. In retrospect, this choice may not have been the best, as it significantly complicated the use-def analysis.

6.3.3 Use-Def Analysis

With the full control flow information available, we can go on to analyze how local variables and method parameters are declared, assigned, and read. This is the most complex of the three analyses, in part due to the way the flow analysis operates at the statement level rather than the expression level, and in part due to the presence of inner classes, for which use-def analysis of free variables presents additional complexities.

[4] Fixed-point iteration was chosen rather than a single iteration with backpatching because it makes it easier to show that the backflows are correctly calculated. It does not cause a significant performance hit.

The use-def analysis is a fixed-point iteration traversal of the flow graph, with sub-traversals of the ASTs for expressions. The set of current definitions is passed around during the traversal, and the nodes are annotated with pointers from definitions (assignments) to possible uses, from uses (reads) to possible definitions, and between all definitions and uses and their declarations.

There are three different traversers in use in the use-def analysis. Firstly, there is the `OuterTraverser`, which traverses class definitions to find method bodies. For each method body, a fix-point traverse traverses the flow graph, starting with a set of definitions containing only the method parameters. This traversal performs fixed point iteration to evaluate the effect of cycles in the flow graph. Each expression within a flow graph node is then traversed by `UseDefTraverser` in a tree-like manner to harvest definitions[5] and uses.

When an inner class is encountered inside a method body, the bodies of any methods inside that class must be separately traversed, but with access to the definitions in the outer method, in case they are accessed in the inner method. To that purpose, a new `UseDefTraverser` object is created, and the current `UseDefTraverser` is stored in it as the outer traverser.

Static Single Assignment form (SSA)[17] would have eased the analysis somewhat, and indeed most places where definitions are used we want to know that there is only one definition. However, we felt that the extra transformations and handling of the ϕ-nodes would outweigh most of the benefits. Indeed, the handling of multiple possible definitions has given few problems, and allows some rewriters like TightenType to easily access all possible definitions.

The most significant problems in the use-def analysis were caused by improper cleanup of the use-def information on the AST, and inner classes in loops. To simplify the rewriters, we did not require them to adjust the use-def information after each rewrite. Instead, we collect all rewrites that are found in one scan, apply them together, and then throw away the use-def information. Some information was not properly disposed of, causing intermittent errors in the later analyses. Inner classes require a separate analysis because control flow from the enclosing class does not enter the inner class. However, since outer variables can be captured, and might have different values in different loop iterations, special care must be taken to repeat the analysis of inner classes in loops.

A working use-def analysis for Java programs is a valuable tool, not only for the rewriters, but for any kind of analysis and optimization tool. While we do assume that the FQCN, ForWhile and DoWhile

[5] Java allows assignments as part of expressions.

normalizers have been applied, we do not rely on flattening, as that would make it impossible to implement the SingleUse unnormalizer described in section 6.6.1. Thus, the flattening transformation, which can cause slowdowns if not undone, is not necessary to use this analysis for other purposes.

6.4 Code-expanding Rewriters

The rewriting system performs a number of optimizations by repeated application of a set of simple rewrite rules. The only special assumptions the rewriters make about the code are those guaranteed by the normalizers. However, the rewriters are designed to work best with the style of Java found in the Jumbo combinators: many final variables, fields and methods, objects that are used as tuples, function objects, and other features reminiscent of functional programming. The rewriters rely not only on the normalizers and the underlying analyses described above, but also on the ability to determine the type of any expression.

We give an example of the transformations done by each rewriter. For the sake of legibility, the examples are not in fully flattened form where it makes no difference for the rewriter in question.

The first two rewriters, the expanding rewriters, perform transformations that may increase the size of the program, in the hope that further transformations can then decrease the size significantly. Since they are not applied automatically, they do not need to make progress in the sense of section 6.1.

6.4.1 Inlining

The Inlining rewriter is the most complex rewriter in the system. It will attempt to inline a method invocation where the actual method can be determined. The only methods that can be inlined are those marked final, marked static, or that appear in a final class (including anonymous inner classes). Adding more careful analyses of subclassing and type restrictions would allow more methods to be inlined.

In order to preserve the semantics of call-by-value parameters, the actual arguments to the method are stored into generated variables rather than copied throughout the inlined method. Similarly, return statements in the method are replaced with an assignment to a generated variable followed by a break statement. While we could avoid this extra copying by analyzing whether the variables are assigned to inside the method, the code-reducing rewriters can remove any extraneous definitions anyway without the need for extra analysis.

```
public class Point {
  double x;
  double y;
  final double hDist(Point p) { ... }
  boolean isLeftOf(Point p) {
    {
      final double flatten_3;
      {
        final Point gen5_this;
        gen5_this = this;
        Point gen5_p;
        gen5_p = p;
        final double gen5_return;
        gen5_break: {
          {
            final double gen5_flatten_0;
            gen5_flatten_0 = gen5_p.x;
            final double gen5_flatten_1;
            gen5_flatten_1 = gen5_this.x;
            final double gen5_flatten_2;
            gen5_flatten_2 = (gen5_flatten_0-gen5_flatten_1);
            {
              gen5_return = gen5_flatten_2;
              break gen5_break;
            }
          }
        }
        flatten_3 = gen5_return;
      }
      final boolean flatten_4;
      flatten_4 = (flatten_3>0);
      return flatten_4;
    }
  }
}
```

Figure 6.5: The effect of the Inlining rewriter on the code in Figure 6.4(b). The original body of hDist has been elided.

For non-static methods we need to replace references to `this` and `super` with references to the appropriate classes. Anonymous inner classes, having no named type, cannot have their `this` instance directly assigned. Thus, when inlining a method on an anonymous inner class, we must name the class with a unique name and assign a new instance of the named class to the `this` instance. Named inner classes inside methods can be inlined if care is taken to ensure unique naming in case the same named class is inlined multiple times in one method. We currently do not allow inlining of methods with named inner classes, as it is rarely seen in the combinators.

Figure 6.5 shows an example of what an inlined method looks like. The method arguments, the `this`

reference, and the return value are all stored in generated variables, and all variables in the method have unique names assigned to them. The `return` statement is replaced by an assignment to the return value followed by a break. At the end of the inlining, the result is assigned to the variable that originally received the return value. This is required to allow final variables to be assigned values from methods that may have several return statements.

Figure 6.5 also shows that inlining adds a number of extraneous scopes, variables and breaks. Rather than complicating the Inlining rewriter with avoiding these, they are removed by separate rewriters.

To define inlining, we must first define several helper functions. In these functions, ν_x means the unique name generated for x in this inlining. Prelude sets up the variables being generated for the target and the method arguments.

$$\begin{aligned}\text{Prelude}(e_t, a_1, \ldots, a_i) = \; & s_{this}, \\ & \textsc{VarDecl}(\text{TypeOf}(a_1), \nu_1), \textsc{Assign}(\nu_1, a_1), \\ & \vdots \\ & \textsc{VarDecl}(\text{TypeOf}(a_i), \nu_i), \textsc{Assign}(\nu_i, a_i),\end{aligned}$$

where

$$\begin{aligned}s_{this} = \; & \textsc{Statements}(\textsc{VarDecl}(\text{TypeOf}(e_t), \nu_{\text{this}}), \textsc{Assign}(\nu_{\text{this}}, e_t)), \text{ if } e_t \neq \epsilon \\ & \epsilon \text{ otherwise}\end{aligned}$$

NewBody performs the replacements inside the copy of the method body. In this, $e\,[a/b]$ means e with all occurrences of b replaced by a.

$$\begin{aligned}\text{NewBody}(m, r) = \text{BodyOf}(m)\,[\; & \nu_{\text{this}}/\textsc{This}(), \nu_1/p_1, \nu_i/\ldots, p_i, \epsilon/\textsc{Class}(\_), \\ & \textsc{Statements}(\textsc{Assign}(\nu_{\text{ret}}, e), \textsc{Break}(r))/\textsc{Return}(e), \\ & \textsc{Break}(r)/\textsc{Return}(\epsilon), \nu_{l_1}/l_1, \ldots, \nu_{l_n}/l_n\;]\end{aligned}$$

where $l_1, \ldots, l_n =$ variables declared in m,

$p_1, \ldots, p_1 =$ formal parameters of m,

DoInline puts the inlined method together, adds a label for the return replacements, and handles assignment of the return value, if any.

$$\text{DoInline}(m, e_t, a_1, \ldots, a_i, \epsilon) = \textsc{Label}(r, \textsc{Statements}(\text{Prelude}(e_t, a_1, \ldots, a_i), \text{NewBody}(m, r)))$$

DoInline$(m, e_t, a_1, \ldots, a_i, v) =$

STATEMENTS(VARDECL(TypeOf$(m), \nu_{\text{ret}}$), DoInline$(m, e_t, a_1, \ldots, a_i, \epsilon)$, ASSIGN$(v, \nu_{\text{ret}})$)

where ν_{ret} is fresh

The MakeThis function handles the proper definition of `this` in the inlined method. Static methods have no this, and most non-static methods can just use the target they were invoked on. However, anonymous classes must be named for the type of `this` to be correct. Thus, when the target of a method invocation is an anonymous class, we create a named class with the same body plus an extra constructor that calls the constructor of the superclass. We then use a new instance of the named class as replacement for `this` in the inlined method. The MakeThis function returns a tuple of a new class definition and the expression that should be used as the method invocation target.

MakeThis$(_, m) = (\epsilon, \epsilon)$ if m is static

MakeThis(ANONYMOUSCLASS$(c_s, a_1, \ldots, a_n, s), m) =$

(CLASS$(c, c_s$, STATEMENTS(CONSTRUCTOR$(p_1, \ldots, p_n$, SUPERCALL$(p_1, \ldots, p_n)), s)),$

NEWOBJECT$(c, a_1, \ldots, a_n)$)

where c is a fresh class name

MakeThis$(t, m) = (\epsilon, t)$

We are now ready to define the inlining rewriter itself. The rewriter detects four variants of inlining opportunities — whether or not the result is being assigned, and whether or not the method is static — and calls DoInline. The FindMethod function finds the method on the given expression that matches the name and arguments.

Inlining $[\![$EXPRESSIONSTATEMENT(METHODINVOCATION$(e_t, n, a_1, \ldots, a_i))]\!] \Rightarrow$

STATEMENTS(c, e)

if $(c, e'_t) =$ MakeThis(e_t),

$m =$ FindMethod(TypeOf$(e_t), n, a_1, \ldots, a_i)$,

m is a final method,

$e =$ DoInline$(m, e'_t, a_1, \ldots, a_i, \epsilon)$,

Inlining $[\![\text{ASSIGN}(\text{VAR}(v), \text{METHODINVOCATION}(e_t, n, a_1, \ldots, a_i))]\!] \Rightarrow \text{STATEMENTS}(c, e)$

if $(c, e'_t) = \text{MakeThis}(e_t)$,

$m = \text{FindMethod}(\text{TypeOf}(e_t), n, a_1, \ldots, a_i)$,

m is a final method,

$e = \text{DoInline}(m, e'_t, a_1, \ldots, a_i, v)$,

Inlining $[\![\text{EXPRESSIONSTATEMENT}(\text{METHODINVOCATION}(e_t, n, a_1, \ldots, a_i))]\!] \Rightarrow e$

if $m = \text{FindMethod}(\text{TypeOf}(e_t), n, a_1, \ldots, a_i)$,

m is a static method,

$e = \text{DoInline}(m, \epsilon, a_1, \ldots, a_i, \epsilon)$,

Inlining $[\![\text{ASSIGN}(\text{VAR}(v), \text{METHODINVOCATION}(e_t, n, a_1, \ldots, a_i))]\!] \Rightarrow e$

if $m = \text{FindMethod}(\text{TypeOf}(e_t), n, a_1, \ldots, a_i)$,

m is a static method,

$e = \text{DoInline}(m, \epsilon, a_1, \ldots, a_i, v)$,

6.4.2 WhileUnroll

In some cases, unrolling a loop can provide more opportunities for optimizing. Although this is more usable for languages with low-level optimizations, there may be cases where unrolling can help in a Java program. A short constant-length loop can be completely unrolled, or the first iteration can be unrolled to make the remaining loop have similar definitions from inflow and backflow.

Since `for` and `do` loops are transformed into `while` loops as part of normalization, we need only deal with `while` when attempting unrolling. The WhileUnroller rewriter unrolls a single iteration of a while statement. It creates a new `if` statement guarding one copy of the loop body followed by the loop itself. Variables defined inside the loop body must be replaced with uniquely named variables, so that when later reductions remove scopes around them, they do not clash with the original variables. Figure 6.6 shows how a while loop is unrolled.

WhileUnroll $[\![\text{WHILE}(e_t, s_b)]\!] \Rightarrow \text{IF}(e_t, s_b[\nu_i/v_i], \text{WHILE}(e_t, s_b))$

if v_i is the set of variables defined in s_b

```
int sumSquares(int max) {
  int sum;
  sum = 0;
  int x;
  x = 0;
  while (true) {
    final boolean flatten_0;
    flatten_0 = (x < max);
    if (flatten_0) {
      final int flatten_1;
      flatten_1 = (x*x);
      sum = (sum+flatten_1);
      x = (x+1);
    } else break;
  }
  return sum;
}
```

(a) Before WhileUnroll

```
int sumSquares(int x) {
  int sum;
  sum = 0;
  if (true) {
    {
      final boolean unroll1_flatten_0;
      unroll1_flatten_0 = (x < max);
      if (unroll1_flatten_0) {
        final int unroll1_flatten_1 = (x*x);
        sum = (sum+unroll1_flatten_1);
        x = (x+1);
      } else break;
    }
    while (true) {
      final boolean flatten_0;
      flatten_0 = (x < max);
      if (flatten_0) {
        final int flatten_1;
        flatten_1 = (x*x);
        sum = (sum+flatten_1);
        x = (x+1);
      } else break;
    }
  }
}
```

(b) After WhileUnroll

Figure 6.6: Unrolling one iteration of a while loop. Variables defined inside are renamed to avoid later name clashes.

6.5 Code-reducing Rewriters

The bulk of the rewriters are reducing rewriters that can never increase the size of a program (in terms of AST nodes), and will usually reduce the size. While some of them perform common optimizations like constant propagation and arithmetic reduction, many of them mainly serve to clean up extraneous code created by the expanding rewriters. This allows the expanding rewriters to be simpler.

The overall approach to designing the code-reducing rewriters is to make many small rewriters rather than a few big ones. Some similar rewrites, such as constant propagation through fields under varying conditions, are split into several rewriters when the checks necessary to verify their applicability are significantly different. This may lead to some overlap in the effects of the rewriters, but ensures that each rewriter is as simple as possible to write, debug and understand.

6.5.1 UnusedBreak

Since the Inlining rewriter replaces return statements with labeled breaks, we can end up with "spaghetti code" if the original method has many returns. However, any break that does not affect the control flow of the code can be removed by the UnusedBreak rewriter. These unnecessary breaks are introduced when a return statement occurs at the end of the control flow in a method. Figure 6.7 shows how some breaks can be removed, even if they are not the syntactically last statement in a block.

```
public class UnusedBreak {
  int breaks(boolean b) {
    A: {
      {
        if (b) break A;
      }
      if (b) break A;
      else b = false;
    }
  }
}
```

(a) Two breaks in a method

```
public class UnusedBreak {
  int breaks(boolean b) {
    A: {
      {
        if (b) break A;
      }
      if (b) ;
      else b = false;
    }
  }
}
```

(b) The second break can be removed by UnusedBreak

Figure 6.7: How UnusedBreak can remove Break statements at the end of control flow.

To check whether a break can be removed, we must check all statements enclosing the break statement, out to its target. For each enclosing statement list, we must find ourselves in the very last statement, or removing the break could cause statements to be skipped. Furthermore, if the break breaks out of a while loop or a switch construct, we leave the break in place.

UnusedBreak $[\![s = \text{BREAK}(l)]\!] \Rightarrow \quad \epsilon$

if $\quad p_0 = s, \text{Parents}(s) = (p_1, p_2, \ldots, p_n),$

$\text{Target}(s) = p_k,$

$\forall i : 1 < i \leq k, p_i = \text{STATEMENTS}(s_1, s_2, \ldots, s_m) : s_m = p_{i-1},$

$\forall 1 < i < k : p_i \notin \{\text{WHILE}(W), \text{SWITCH}(S)\}$

In many cases where returns occur before the end of the control flow for a method, a simple change in the programming style can make the method more amenable to being reduced after inlining. For example, while the code example in Figure 6.8(a) leads to spaghetti-like breaks when inlined, the functionally equivalent code in Figure 6.8(b) does not create any breaks that cannot be removed, as all the returns are at the end

```
int quadrant(float x, float y) {
  if (x > 0) {
    if (y > 0) return Q_NE;
    return Q_SE;
  }
  if (y > 0) return Q_NW;
  return Q_NE;
}
```

(a) Non-removable

```
int quadrant(float x, float y) {
  if (x > 0) {
    if (y > 0) return Q_NE;
    else return Q_SE;
  } else {
    if (y > 0) return Q_NW;
    else return Q_NE;
  }
}
```

(b) Removable

Figure 6.8: The use of `return` inside `if` determines whether breaks introduced by inlining can be removed.

of the control flow anyway. The latter style is similar to that of some functional languages, where an if statement always has both a true branch and a false branch. However, there are cases where such a style would lead to overly complex control flow, particularly when checking a number of predicates, each of which cause a return.

6.5.2 UnusedScope

Extraneous scopes are introduced by a number of rewriters, in particular inlining and flattening, in order to ensure a scope for temporary variables. These scopes not only clutter the code, they also make it harder for other rewriters to prove that a rewrite is legal. Fortunately, many newly introduced scopes can be removed immediately.

Only three situations can prevent a scope from being removed:

- The scope contains several statements that must be grouped inside an `if`, `while` or similar statement.
- A variable is defined inside the scope, and a variable of the same name is defined after the end of the scope.
- A `break` statement occurs within the scope that breaks out of exactly that scope.

The first situation is benign, as such scopes are not introduced by other rewriters. The second situation is rare, so we take the conservative approach of never deleting a scope if a variable defined directly in that scope has the same name as a variable defined anywhere else in the method. Only the third situation happens with any frequency, mostly when the Inlining rewriter has introduced breaks for return statements. As mentioned in 6.5.1, a change of programming style can allow many such breaks to be removed.

```
void f (boolean b) {
  if (b) {
    A:{
      int c;
      B:{
        int a;
      }
    }
  } else {
    C: {
      int d;
      D:{
        break C;
      }
    }
    int a;
  }
}
```

(a) Before UnusedScope

```
void f(boolean b) {
  if (b) {
    int c;
    B: {
      int a;
    }
  } else {
    C: {
      int d;
      break C;
    }
    int a;
  }
}
```

(b) After UnusedScope

Figure 6.9: Scopes being removed by UnusedScope rewriter.

Figure 6.9 shows which scopes can be removed by UnusedScope and which not. Scope `B` is retained because we cannot prove that `a` is not defined twice. Scope `C` is retained because it is the target of a break. If the UnusedBreak rewriter is applied to this, the break would disappear, and scope `C` could be removed as well.

UnusedScope $[\![s = \text{SCOPEDSTATEMENTS}(s_1, s_2, \ldots, s_n)]\!] \Rightarrow \quad (s_1, s_2, \ldots, s_n)$

if the surrounding node is SCOPEDSTATEMENTS() or STATEMENTS(),

$m =$ the method containing s,

$\forall b = \text{BREAK}(\_) \in \text{Statements}(s_1, s_2, \ldots, s_n)$:

$s \neq \text{Target}(b)$,

$\forall s_v = \text{VARDECL}(t, v) \in \text{Statements}(s_1, s_2, \ldots, s_n)$:

$\forall s_{v'} = \text{VARDECL}(t', v') \in \text{Statements}(m)$:

$v \neq v' \vee s_v = s_{v'}$

```
int foo() {
  int v = 3;
  int u = v;
  int w;
  if (u == 2) w = u;
  else {
    int x = u+1;
    w = x;
  }
  return w;
}
```

(a) Before CopyAssigment

```
int foo() {
  int v = 3;
  int u = v;
  int w;
  if (v == 2) w = v;
  else {
    int x = v+1;
    w = x;
  }
  return w;
}
```

(b) After CopyAssignment

Figure 6.10: CopyAssignment can change u to v, but cannot change w — not only is w multiply defined, the second definition cannot leave the scope

UnusedScope $[\![s = \text{LABEL}(l, \text{SCOPEDSTATEMENTS}(s_1, s_2, \ldots, s_n))]\!] \Rightarrow \quad (s_1, s_2, \ldots, s_n)$

if the surrounding node is STATEMENTS(_),

$m =$ the method containing s,

$\forall b = \text{BREAK}() \in \text{Statements}(s_1, s_2, \ldots, s_n)$:

$s \neq \text{Target}(b)$,

$\forall s_v = \text{VARDECL}(t, v) \in \text{Statements}(s_1, s_2, \ldots, s_n)$:

$\forall s_{v'} = \text{VARDECL}(t', v') \in \text{Statements}(m)$:

$v \neq v' \vee s_v = s_{v'}$

6.5.3 CopyAssignment

A copying assignment occurs when a variable is assigned the value of another variable. In that case, we would like to directly change any uses of the second variable to uses of the first variable. To do that, we must ensure that the variable does not change between definition and use. While this rewrite is small, it occurs frequently with inlined code.

The MayMove function checks whether it is legal to move a variable to replace an expression in the AST. It checks that scoping is obeyed, and that the definition of the variable cannot change between the two places. Conservatively, we check stability of the definition by checking that all definitions of the variable anywhere in the method come before the variable being moved. Figure 6.10 shows some cases where CopyAssignment

can be used, and some where it cannot.

MayMove$(v_{from}, e_{to}) =$ true, if v_{from} dominates $e_{to} \wedge v_{from}$ is in scope at $e_{to} \wedge$
$\forall e_{def} \in$ Defs(Decl$(v_{from})) : e_{def}$ dominates v_{from}
false otherwise

CopyAssignment $[\![\text{ASSIGN}(v_1, v_2)]\!] \Rightarrow$ ASSIGN(v_1, v_3)
if Defs$(v_2) = \{v_2 = v_3\}$,
MayMove(v_3, v_1)

6.5.4 FieldValue

Final fields are frequently used in Java to define constants. These fields may be found in places that cannot be accessed by simpler rewrites. For instance, copying an expression out of an object field may cause unforeseen side effects, or a variable may not have any meaning outside of its method. Instead of copying the first definition encountered for a field, this rewriter attempts to trace the contents of fields, variables and arrays as far back as possible. If in the end the contents turn out to be movable, and all the intermediate steps can be proven sufficiently constant (by being final or assigned immediately before), it will replace the field with the contents found.

In figure 6.11, the field lookup `flatten_1.h` can be traced back through the field `flag`, through the constructor, to the field `A`, whose initializer we can move. The values assigned to `h` and `flag`, while fixed, cannot be moved to replace the field lookup, as that could cause code duplication (of `new Fields(a)`) or violate scope rules (for `x`). This particular example is similar in structure to the `Type` class used in Jumbo.

```
class Fields {
  static final int A = 1;
  final Fields flag = new Fields(A);
  final int h;
  Fields(int x) { h = x; }
  int sumFields() {
    Fields flatten_1 = this.flag;
    return flatten_1.h;
  }
}
```

(a) Before FieldValue

```
class Fields {
  static final int A = 1;
  final Fields flag = new Fields(A);
  final int h;
  Fields(int x) { h = x; }
  int sumFields() {
    Fields flatten_1 = this.flag;
    return 1;
  }
}
```

(b) After FieldValue

Figure 6.11: `flatten_1.h` can be replaced with 1, even though it must be traced though two fields.

The helper function ExpressionContents attempts to find a movable definition of an expression by following definitions as far as possible.

ExpressionContents($\textsc{ArrayAccess}(e_a, e_i)) = e_x$

if $e_t =$ ExpressionContents(e_a),

$\textsc{Literal}(x) =$ ExpressionContents(e_i),

Parents$(e_t) = [p_1, \ldots]$,

$\textsc{Assign}(\textsc{Var}(v), \textsc{ArrayCreation}(t, e_l)) = p_1$,

$\forall u \in$ Uses(Decl$(v))$: $\textsc{Assign}(\textsc{ArrayAccess}(u, e_u), e_x) \vee$

$\textsc{Assign}(\textsc{Var}(\_), \textsc{ArrayAccess}(u, \_))$

$\forall e_u$: ExpressionContents$(e_u) = \textsc{Literal}(x)$,

e_u dominates e_a,

e_t dominates e_u

ExpressionContents($\textsc{FieldAccess}(e, f)) = a_k$

if $e_t =$ ExpressionContents(e),

$e_t = \textsc{NewObject}(c, a_1, \ldots, a_i)$,

f is defined on c,

f is non-static and final,

f is assigned $\textsc{Var}(p_k)$ in the constructor,

p_k is final,

a_k dominates e

ExpressionContents($\textsc{FieldAccess}(e, f)) = e_f$

if $e_t =$ ExpressionContents(e),

f is defined on e_t,

f is final,

f is initialized to e_f

ExpressionContents(FIELDACCESS$(e, f)) = e_a$

if $e_t =$ ExpressionContents(e),

$e_t =$ NEWOBJECT$(c, a_1, \ldots, a_i)$,

f is defined on c,

f is non-static and final,

f is assigned e_a in the constructor,

$e_a =$ LITERAL(x)

$$\text{ExpressionContents}(\text{VAR}(v)) = \begin{cases} d, & \text{if Defs}(v) = \{d\} \wedge d \text{ dominates } v, \\ \epsilon & \text{otherwise} \end{cases}$$

ExpressionContents(LITERAL(x)) = LITERAL(x)

ExpressionContents(THIS$()$) = THIS$()$

ExpressionContents(THIS(c)) = THIS(c)

ExpressionContents(_) $= \epsilon$

FieldValue $[\![e_f =$ FIELDACCESS$(e, f)]\!] \Rightarrow$ VAR(v)

if VAR$(v) =$ ExpressionContents(e_f),

MayMove(v, e)

FieldValue $[\![e_f =$ FIELDACCESS$(e, f)]\!] \Rightarrow e'$

if $e' =$ ExpressionContents(e_f)

$e' \in \{$LITERAL(x), THIS$(_)$, UNARY$(v)\}$

6.5.5 Untupling

Jumbo uses a number of tuple-like objects, i.e. objects where all fields are final and initialized via constructor parameters. In this rewriter, we find uses of fields whose definition we can locate, and replace the uses with the definition. In order to ensure that the field has the assigned value, we check that the object being assigned to is created in view and does not escape to a place we cannot check. We also need to check that the assignment to the field is the only one, and that we can move the expression being assigned to the use. Unlike the FieldValue rewriter, Untupling allows us to propagate values through non-final fields. Figure 6.12 shows an example of a field lookup that Untupling can rewrite.

```
class Tuple {
  final int a;
  Tuple(int x) { a = x; }
}
class UseTuple {
  int storeAndLoad(int x) {
    Tuple t1 = new Tuple(x);
    return t.a;
  }
}
```

(a) Before Untupling

```
class Tuple {
  final int a;
  Tuple(int x) { a = x; }
}
class UseTuple {
  int storeAndLoad(int x) {
    Tuple t1 = new Tuple(x);
    return x;
  }
}
```

(b) After Untupling

Figure 6.12: Untupling can move a non-final variable through an object field.

The helper function MayEscape checks through the chain of constructors to see if the object may be able to escape at any point.

MayEscape$(c, a_1, \ldots, a_n) = true$

iff $\exists c_i \in$ Constructors$(c, a_1, \ldots, a_n) : \exists s \in$ Statements(c_i) :

$s =$ THROW(_)$\vee$

$s =$ NEWOBJECT(_)$\vee$

$s =$ METHODINVOCATION(_)$\vee$

$s =$ ASSIGN(f, e), where f is a static field or a field defined in another object

Untupling $[\![e =$ FIELDACCESS$(v, f)]\!] \Rightarrow \quad a_i$

if e is not the left-hand side of an assignment,

Defs$(v) = \{e_c =$ NEWOBJECT$(c, a_1, \ldots, a_n)\}$,

f is defined in c,

Constructors$(c, a_1, \ldots, a_n) = [m, \ldots]$,

$m =$ CONSTRUCTOR$(p_1, \ldots, p_n, s, b)$,

$\exists$ exactly one ASSIGN(FIELDACCESS(THIS$()$, $f)$, $e_f) \in b$,

$e_f = (p_i \wedge$ MayMpve$(a_i, v)) \vee e_f =$ LITERAL(x),

$\neg$mayEscape(e_c),

$\forall u \in$ Uses(Decl(v)) :

(ASSIGN(FIELDACCESS$(u, f), e_a) \wedge e$ dominates $u)\vee$

(ASSIGN(FIELDACCESS$(u, f')) \wedge f \neq f'$)

6.5.6 ArrayLength

The ArrayLength rewriter attempts to propagate the length of an array from its initialization to its use. The initialization of the array must dominate the use. Because of flattening, the expression used in the initialization is a variable or a literal. If it is a variable, we must check that it is available at the use of the array.

ArrayLength $[\![\text{FIELDACCESS}(e_1, \text{“length”})]\!] \Rightarrow \quad e$

if Defs$(n_1) = \{\text{ASSIGN}(e_1, \text{ARRAYCREATION}(t, e))\}$,

$e = \text{LITERAL}(\_)$

ArrayLength $[\![\text{FIELDACCESS}(e_1, \text{“length”})]\!] \Rightarrow \quad e$

if Defs$(n_1) = \{\text{ASSIGN}(e_1, \text{ARRAYCREATION}(t, e))\}$,

$e = \text{VAR}(v)$,

e dominates e_1,

$\forall e' \in$ Defs(Decl(v)) : e' dominates e,

v is in scope at e_1

6.5.7 Arithmetic

Arithmetic rewriting aims at reducing arithmetic expressions. Since expressions are flattened, the operands of binary expressions can only be literals, variables or `this` expressions. None of the operand expressions can have any side-effects, so we can reduce logical expressions without having to worry about whether side-effects are affected. We also reduce multiplication by 0 or 1 and additions and subtractions with 0, as well as operations on literal strings. By checking what the possible definitions of variables compared to `null` are, we can also reduce some object comparisons. The current implementation does not nearly exhaust the numerous ways we can do arithmetic rewritings, but covers those cases we are most likely to encounter when rewriting the combinators.

NonNull$(e) =$ true, if $\forall e' \in$ Defs(e) :

$e' \in \{\text{NEWOBJECT}(c, a_1, \ldots, a_n), \text{ANONYMOUSCLASS}(s, a_1, \ldots, a_n)\}$

false otherwise

Arithmetic $[\![\text{BINARY}(e_1, \|, e_2)]\!] \Rightarrow$ true, if e_1 or e_2 is true

Arithmetic $[\![\text{BINARY}(e_1, \&\&, e_2)]\!] \Rightarrow$ false, if e_1 or e_2 is false

Arithmetic $[\![\text{BINARY}(\text{true}, \&\&, e)]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(e, \&\&, \text{true})]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(\text{false}, \|, e)]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(e, \|, \text{false})]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(e_1, *, e_2)]\!] \Rightarrow$ 0, if e_1 or e_2 is 0

Arithmetic $[\![\text{BINARY}(1, *, e)]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(e, *, 1)]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(0, +, e)]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(e, +, 0)]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(e, -, 0)]\!] \Rightarrow e$

Arithmetic $[\![\text{BINARY}(0, -, e)]\!] \Rightarrow \text{UNARY}(-, e)$

Arithmetic $[\![\text{BINARY}(\text{null}, =, \text{null})]\!] \Rightarrow$ true

Arithmetic $[\![\text{BINARY}(\text{null}, \neq, \text{null})]\!] \Rightarrow$ false

Arithmetic $[\![\text{BINARY}(\text{null}, =, e)]\!] \Rightarrow$ false
if NonNull(e)

Arithmetic $[\![\text{BINARY}(\text{null}, !=, e)]\!] \Rightarrow$ true
if NonNull(e)

Arithmetic $[\![\text{BINARY}(e, =, \text{null})]\!] \Rightarrow$ false
if NonNull(e)

Arithmetic $[\![\text{BINARY}(e, !=, \text{null})]\!] \Rightarrow$ true

if NonNull(e)

Arithmetic $[\![\text{BINARY}(\text{LITERAL}(l_1), o, \text{LITERAL}(l_2))]\!] \Rightarrow l$

if l is the evaluation of o on l_1 and l_2

Arithmetic $[\![\text{UNARY}(!, e \in \{\text{false}, \text{true}\})]\!] \Rightarrow \neg e$

Arithmetic $[\![\text{UNARY}(-, \text{LITERAL}(x))]\!] \Rightarrow -x$

6.5.8 ConstantPropagation

The ConstantPropagation rewriter finds uses of variables where the only possible definition assigns a constant value. This is similar to CopyAssignment, but does not require checking if the value can be moved, and so allows propagating in more cases.

ConstantPropagation $[\![\text{ASSIGN}(v_1, \text{VAR}(v_2))]\!] \Rightarrow \text{ASSIGN}(v_1, \text{LITERAL}(x))$

if Defs(v_2) $= \{\text{LITERAL}(x)\}$

6.5.9 Switch

If the expression used in a `switch` statement is a constant value, we can remove the switch, leaving only the case used. If the case used has control flow going into the next case, we must preserve that case as well, etc. Figure 6.13 shows a switch that can be reduced due to the test being constant. The cases 3, 4, 5 and 6 are kept, but the default cannot be reached and so is discarded.

ControlTo(n, n') = true, iff control flow can pass from n to n'.

Switch $[\![\text{SWITCH}(\text{LITERAL}(x), c_1 : \text{SwitchCase}, \ldots, c_n : \text{SwitchCase})]\!] \Rightarrow$

$\text{SCOPEDSTATEMENTS}(s')$

if $\forall i : 0 < i \leq n : c_i = \text{SWITCHCASE}(e_{i,1}, \ldots, e_{i,k}, s_i)$,

$\exists i, j : e_{i,j} = x$,

$\exists m : i \leq m \leq n : \forall t : i \leq t < m : \text{ControlTo}(s_t, s_t + 1)$,

$m = n \vee \neg\text{ControlTo}(s_m, s_m + 1)$,

$s' = \text{STATEMENTS}(s_i, \ldots, s_m)$

```
int switcher(int x) {
  switch (3) {
    case 1: return 0;
    case 2: return 2;
    case 3: case 4: x += 1;
    case 5: if (x == 3) return x;
    case 6: return x+1;
    default: return x;
  }
}
```

(a) Before Switch

```
int switcher(int x) {
  x += 1;
  if (x == 3) return x;
  return x+1;
}
```

(b) After Switch

Figure 6.13: A constant-value switch statement can be reduced, but we must preserve any fall-through.

Switch $[\![\text{SWITCH}(\text{LITERAL}(x), c_1 : \text{SwitchCase}, \ldots, c_n : \text{SwitchCase})]\!] \Rightarrow$

$\text{SCOPEDSTATEMENTS}(s')$

if $\forall i : 0 < i \leq n : c_i = \text{SWITCHCASE}(e_{i,1}, \ldots, e_{i,k}, s_i)$,

$\nexists i, j : e_{i,j} = x$,

$\exists i : c_i = \text{DEFAULTCASE}(s_i)$,

$\exists m : i \leq m \leq n : \forall t : i \leq t < m : \text{ControlTo}(s_t, s_t + 1)$,

$m = n \vee \neg\text{ControlTo}(s_m, s_m + 1)$,

$s' = \text{STATEMENTS}(s_i, \ldots, s_m)$

6.5.10 TightenType

If the declared type of a variable is less specific than what is being assigned to it, we can rewrite the variable declaration to have a more specific type. This in turn may help the Inlining rewriter prove that a method invocation on that variable can safely be inlined. If there is more than one possible assignment, we pick the most specific class that complies with all the assignments. Because the FQCN rewriter makes static field accesses and method invocations use explicit class names for targets, the semantics of those are preserved. Figure 6.14 shows how a variable type can be promoted.

TightenType $[\![\text{VARDECL}(t, v)]\!] \Rightarrow \quad \text{VARDECL}(t', v)$

if $\forall e' \in \text{Defs}(v) : e' = \text{ASSIGN}(v, e) \wedge \text{TypeOf}(e) \sqsubseteq t'$

```
class C1 {}
class C2 extends C1 {}
class C3 extends C1 {}
class Subclasses {
  void foo(C3 x) {
    Object o = new C2();
    if (x != null) o = x;
  }
}
```

(a) Before TightenType

```
class C1 {}
class C2 extends C1 {}
class C3 extends C1 {}
class Subclasses {
  void foo(C3 x) {
    C1 o = new C2();
    if (x != null) o = x;
  }
}
```

(b) After TightenType

Figure 6.14: The variable `o` can be promoted to type `C1`, the common superclass of the two objects assigned to it.

6.5.11 IfReduction

When the test in an `if` statement is a constant value, we can reduce the statement to either its true branch or its false branch. Since other rewriters should move any constant value into the test, we need but check for the degenerate case of a literal expression.

IfReduction $[\![\text{IF}(\text{true}, s_t, s_f)]\!] \Rightarrow \quad s_t$

IfReduction $[\![\text{IF}(\text{false}, s_t, s_f)]\!] \Rightarrow \quad s_f$

6.5.12 UnusedDef

When a definition (assignment) of a variable is never used, the assignment can be removed. This in turn may allow the declaration of that variable to be removed. However, if the expression in the assignment could have side effects, we cannot remove the assignment. Since the expression is flattened, it is easy to determine if it can have side effects; only method invocations, array accesses and field accesses can have side effects. In figure 6.15, the assignment from a method call, while unused, is not rewritten, as a method call might have side effects.

UnusedDef $[\![\text{ASSIGN}(v, e)]\!] \Rightarrow \quad \epsilon$

if $\text{Uses}(v) = \emptyset$,

$e \notin \{\text{METHODINVOCATION}(\_), \text{ARRAYACCESS}(\_), \text{CAST}(\_)\}$

```
int foo() {
  int x = 3;
  int y = x;
  y = Integer.parseString("4");
  y = 6;
  return y;
}
```

(a) Before UnusedDef

```
int foo() {
  int x;
  int y;
  y = Integer.parseString("4");
  y = 6;
  return y;
}
```

(b) After UnusedDef

Figure 6.15: The first assignments is used, and cannot be removed, but the second one is unused and can safely be removed. The third one may have side effects, and the last one is used. In a second application of UnusedDef, the first assignment is no longer used and can be removed.

6.5.13 UnusedDecl

Declarations of variables that are never used nor assigned to can be immediately removed. Since the normalization process guarantees that declarations have no initializers, this rewriter merely needs to check that there are no definitions for it in the use-def analysis.

UnusedDecl $[\![\text{VARDECL}(v)]\!] \Rightarrow \quad \epsilon$

if $\text{Defs}(v) = \emptyset$

6.5.14 UnusedReturn

This rewriter removes assignments of return values from method invocations and object creations when the assigned value is never used. The possibility of removing the method invocation or object creation is left to other rewriters, as that may involve checking for side effects. This allows some assignments to be reduced that UnusedDef cannot handle. Figure 6.16 shows the three different assignments that UnusedReturn can reduce.

```
int foo() {
  int x = Integer.parseInt("2");
  Object o = new ArrayList();
  Object o1 = new Object() { int x; };
}
```

(a) Before UnusedReturn

```
int foo() {
  Integer.parseInt("2");
  new ArrayList();
  new Object() { int x; };
}
```

(b) After UnusedReturn

Figure 6.16: Return values that can safely be discarded.

UnusedReturn $[\![\text{ASSIGN}(v, e)]\!] \Rightarrow \quad e$

if $\quad e \in \{\text{METHODINVOCATION}(\_), \text{NEWOBJECT}(\_), \text{ANONYMOUSCLASS}(\_)\},$

$\quad \text{Uses}(v) = \emptyset$

6.5.15 UnusedObject

If we can prove that an object creation call does not have any side effects, and the value returned is never used, we can remove the call. Any object created outside of an assignment (i.e. at statement level) cannot be used in the calling method.

To ensure that the object creation has no side effects, we check the constructor to ensure that no methods other than constructor methods are called, that no other objects are created, that no exceptions can be thrown, and that no assignments are done to static fields or fields in other objects. This must be checked recursively in calls to `this` and `super` constructor calls. This ensures not only that no other objects are affected by the call, but also that the created object cannot escape to be used somewhere else. Figure 6.17 shows how side effects in the constructors affect the ability to apply UnusedObject. Because of flattening, the arguments to the object creation call cannot cause any side effects, and so can safely be removed. The helper function MayEscape is defined in section 6.5.5.

```
class Empty {}
class Empty2 extends Empty {
  Empty2() { super(); }
}
class Escape { Escape() {
  this.toString();
}
class Escape2 {
  static int x;
  Escape2(int a) { x = a; }
}
int foo() {
  new Empty();
  new Empty2();
  new Escape();
  new Escape2(3);
}
```

(a) Before UnusedObject

```
class Empty {}
class Empty2 extends Empty {
  Empty2() { super(); }
}
class Escape { Escape() {
  this.toString();
}
class Escape2 {
  static int x;
  Escape2(int a) { x = a; }
}
int foo() {
  new Escape();
  new Escape2(3);
}
```

(b) After UnusedObject

Figure 6.17: Object creations with no side effects can be removed, but those with possible side effects cannot.

UnusedObject $[\![\text{EXPRESSIONSTATEMENT}(\text{NEWOBJECT}(c, a_1, \ldots, a_n))]\!] \Rightarrow \epsilon$

if $\neg\text{MayEscape}(c, a_1, \ldots, a_n)$

6.5.16 UnusedFieldAssign

While the UnusedDef rewriter can handle variable assignments that are never used, it is more complicated to remove assignments to fields. If an object can escape to other methods, we cannot hope to prove that an assigned field is never read. Only if we can prove that the object will not escape the current scope can we safely remove an unused field assignment.

We have seen in the Untupling rewriter how to prove that an object cannot escape its constructor. If at the same time the object cannot escape the scope where the constructor was called, and no reads are made from the field in question, the field assign is never used. We check this by ensuring that all uses of the variable containing the object are either assignments to the field in question, or else reads from other fields. This rules out that the object is passed to other methods or otherwise escapes, as any such use would not be a field read. Figure 6.18 shows how some field writes can be removed.

```
class Assigns {
  int f, g;
  void bar() { }
  int foo() {
    Assigns a1 = new Assigns();
    Assigns a2 = new Assigns();
    Assigns a3 = new Assigns();
    a1.f = 3;
    a2.f = 4;
    int flatten_1 = a2.f;
    a1.f = flatten_1;
    a3.f = 5;
    a3.bar();
    System.out.println(a1.g);
  }
}
```

(a) Before UnusedFieldAssign

```
class Assigns {
  int f, g;
  void bar() { }
  int foo() {
    Assigns a1 = new Assigns();
    Assigns a2 = new Assigns();
    Assigns a3 = new Assigns();
    a2.f = 4;
    int flatten_1 = a2.f;
    a3.f = 5;
    a3.bar();
    System.out.println(a1.g);
  }
}
```

(b) After UnusedFieldAssign

Figure 6.18: UnusedFieldAssign can remove `a1.f` assigns, but `a2.f` is used and `a3` might escape in `bar()`. Note that `a2.f` can be removed once the UnusedDef rewriter has removed the assignment to `flatten_1`.

UnusedFieldAssign $[\![\textsc{Assign}(\textsc{FieldAccess}(v, f), e)]\!] \Rightarrow \quad \epsilon$

if $\quad \text{Defs}(v) = \{\textsc{Assign}(v, \textsc{NewObject}(c, a_1, \ldots, a_n))\},$

$\neg\text{MayEscape}(c, a_1, \ldots, a_n),$

$\forall e_u \in \text{Uses}(v) :$

$\textsc{Assign}(\textsc{FieldAccess}(e_u, f), e') \vee$

$\textsc{FieldAccess}(e_u, f'),$ where $f' != f$

6.6 Unnormalizing

As a final step after optimizations are done, we can unnormalize the transformed code to get rid of temporary variables and other inefficiencies introduced by the rewriters. This is done partly to increase readability by humans, but mostly to make it easier to compile the code. Some Java compilers, including Jumbo and Javac, are notoriously simplistic in their register allocation, and could easily run out of registers on flattened code. Unflattening is done by two rewriters: SingleUse and WhileIf. Note that these only unflatten those things that may have an impact on execution speed. The separation of variable declarations and initializers, for instance, is left in place, as it does not change the bytecode created.

6.6.1 SingleUse

The SingleUse rewriter attempts to undo the effects of the Flattening normalizer, in order to avoid extraneous variable declarations and assignments. Doing this is complicated by the need to preserve order of evaluation of expressions, in particular for method invocations, field accesses and array accesses.

The rewriter is somewhat simplified by noting that the variables involved in an expression are frequently evaluated just prior to the expression. This would be the case in all places if flattening was the only rewrite that had occurred. Other rewriters may have changed this somewhat, but our experience is that the flattened structure is mostly preserved as is.

In order to ensure preservation of evaluation order, the rewriter assembles each expression from its subexpressions last to first. When considering whether to move the definition of a subexpression, it must first show that no subexpressions later in the expression might need to be moved, and then show that no subexpressions earlier in the expression might be dangerous to move the definition across.

SingleUse must not only replace the use of a variable with its definition, it must also remove the definition to avoid code duplication. We cannot use the UnusedDef rewriter to remove the definition, as the right-hand side of the assignment might be a method invocation, which UnusedDef cannot remove. While most other rewriters merely return a replacement value for each node, which is then applied after the rewriter has traversed the tree, SingleUse must make two changes, and therefore needs to change the AST directly. Since any such change can break the use-def analysis, each traversal with the SingleUse rewriter replaces at most one variable.

After SingleUse has been applied to unflatten all that can be unflattened, a number of unused variable declarations are left behind. The UnusedDecl rewriter does not depend on flattening to function correctly, so it can remove these afterwards.

For the definition of UnusedDef, we let X() stand for any AST node that directly contains subexpressions, and $e \rightarrow e'$ stand for an in-tree replacement of the node e by the node e'. The first helper function, IsDangerous, checks whether a node is dangerous to move an expression across. Conservatively, we consider only the very simplest expressions to be nondangerous: Literals, `this`-expressions, variables and class names.

IsDangerous($\textsc{Literal}(\_)$) = false

IsDangerous($\textsc{QualifiedName}(\_)$) = false

IsDangerous($\textsc{This}(\_)$) = false

IsDangerous(x) = true

The NothingBefore helper function checks the expression containing the use we are interested in, and makes sure that no prior expressions (in evaluation order) would be dangerous to move the definition across. Again, we take a conservative approach.

NothingBefore(e) = true, if e is not an Expression

NothingBefore(e) = true, if Parents(e) = $\{e_p, \ldots\}$,
$e_p = \text{X}(e_1, \ldots, e_{j-1}, e, e_{j+1}, \ldots, e_n)$,
$\forall i \in [1 \ldots j-1]$: $\neg$IsDangerous(e_i),
NothingBefore(e_p)

NothingBefore(e) = false

The IsMovable helper function checks whether a particular variable v can be moved. There must be nothing dangerous in the expression prior to v, and there must be no dangerous statements between v and its definition. The only non-dangerous statements are variable declarations and assignments involving non-dangerous subexpressions. It returns the definition of v if found, and ϵ otherwise.

IsMovable$(e) = e'$

if $e = v$,

Defs$(v) = \{e_v = \text{ASSIGN}(v, e')\}$,

Uses$(v) = \{e\}$,

NothingBefore(e),

$s =$ the first StatementList enclosing e,

$s = \{\ldots, s_{j-1}, e_v, s_{j+1}, \ldots, s_{k-1}, s, s_{k+1}, \ldots\}$

$\forall i \in [j+1, \ldots, k-1]$: $s_i = \text{VARDECL}(\_)$, or

$s_i = \text{ASSIGN}(e_1, e_2)$, and ¬IsDangerous$(e_1)$, and ¬IsDangerous$(e_2)$

IsMovable$(e) = \epsilon$

In the rewriter itself, we look for an expression or statement with a movable subexpression. We move applicable subexpressions last-to-first, so that the moved expressions do not block further unflattenings. Because of the systematic way flattening has been performed, this works most of the time.

SingleUse $[\![\text{X}(e_1, \ldots, e_n)]\!] \Rightarrow \text{X}(e_1, \ldots, e_{j-1}, e_j \rightarrow e'_j, e_{j+1}, \ldots, e_n)$

if $\exists j \in [1 \ldots n]$: $\forall i \in [j+1 \ldots n] : IsMovablee_i = \epsilon$,

$\forall i \in [1 \ldots j-1] : \neg IsDangerouse_i$,

IsMovable(e_j). Defs$(e_j) = \{e_v = \text{ASSIGN}(v, e')\}$,

$e_v \rightarrow \epsilon$

Figure 6.19 shows how the SingleUse rewriter works by unflattening each expression last-to-first. To avoid blocking each other, the last argument to `overlap` is unflattened first. Once `overlap` cannot be further unflattened itself, it can be moved into the call to `println`. Notice that as soon as SingleUse has been applied, the other rewriters, except UnusedDecl, are no longer safe to apply, as they assume full flattening.

```
final java.io.PrintStream flatten_0;
flatten_0 = java.lang.System.out;
final java.lang.String flatten_1;
flatten_1 = argv[0];
final java.lang.String flatten_2;
flatten_2 = argv[1];
final int flatten_3;
flatten_3 = this.overlap(flatten_1, flatten_2);
flatten_0.println(flatten_3);
```

(a) Before SingleUse

```
final java.io.PrintStream flatten_0;
flatten_0 = java.lang.System.out;
final java.lang.String flatten_1;
flatten_1 = argv[0];
final java.lang.String flatten_2;
final int flatten_3;
flatten_3 = this.overlap(flatten_1, argv[1]);
flatten_0.println(flatten_3);
```

(b) The first SingleUse unflattens the last argument to `overlap`

```
final java.io.PrintStream flatten_0;
final java.lang.String flatten_1;
final java.lang.String flatten_2;
final int flatten_3;
java.lang.System.out.println(this.overlap(argv[0], argv[1]));
```

(c) After three more applications, the original expression is reformed

```
java.lang.System.out.println(this.overlap(argv[0], argv[1]));
```

(d) An application of UnusedDecl removes the extraneous declarations

Figure 6.19: Unflattening of nested function calls

6.6.2 WhileIf

The WhileIf rewriter attempts to undo the normalization of while loops. Since the condition in the while loop must be flattened, but the flattened version is not an expression, the condition must be moved into the loop, in the form of an if-break construct, and the while condition becomes the constant `true`. This rewriter attempts to find this pattern and turn it back into a while loop. It requires the test in the if-statement to be completely unflattened.

WhileIf $[\![\text{WHILE}(\text{true}, \text{STATEMENTS}(\text{IF}(e, s, \text{break})))]\!] \Rightarrow \quad \text{WHILE}(e, s)$

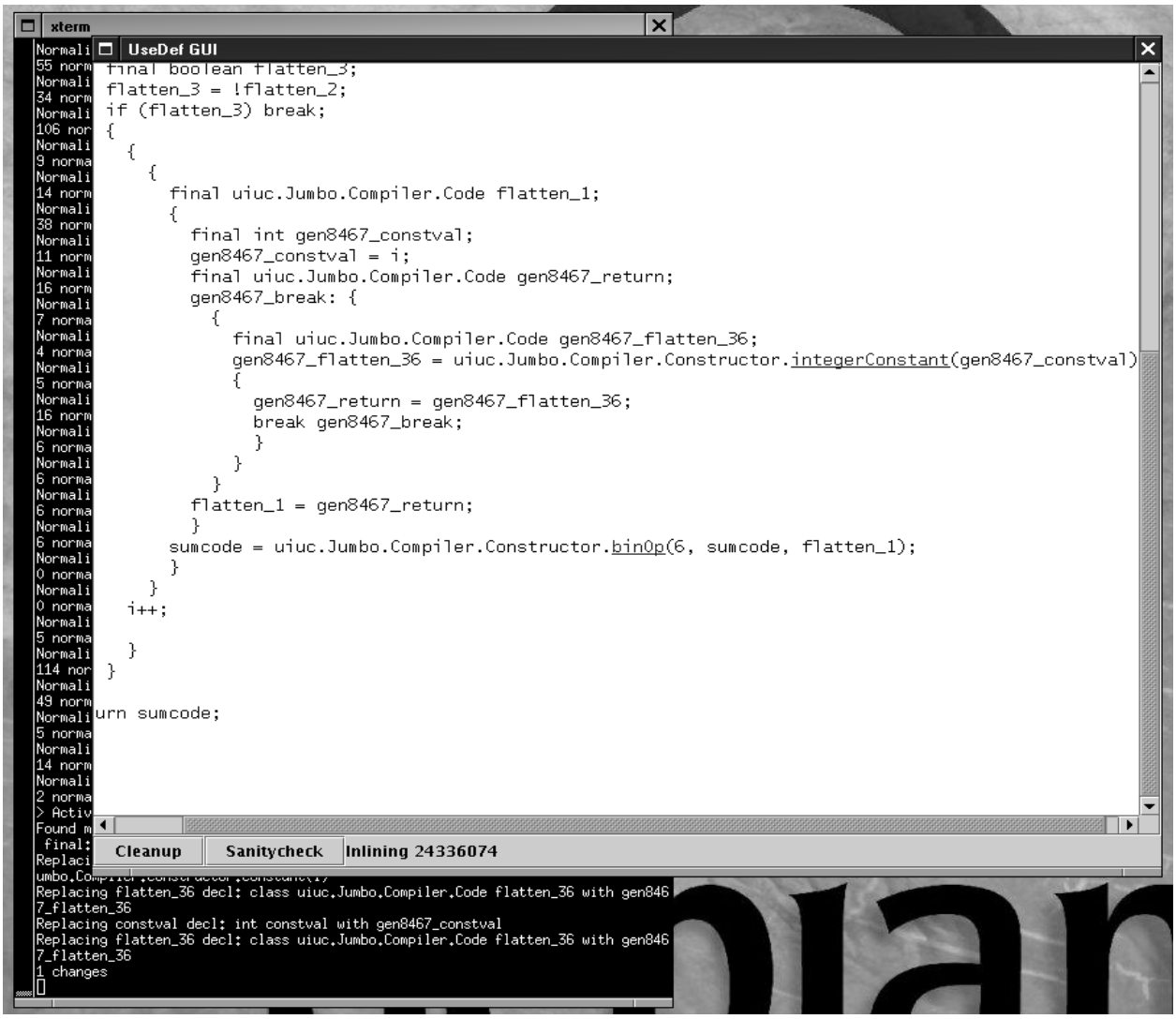

Figure 6.20: Selecting rewriting opportunities

6.7 Usage

The rewriting system has some degree of automation built in. Since the code-reducing rewriters are safe to apply multiple times, we have a fix-point function that applies all the code-reducing rewriters repeatedly until none of them make any changes. This allows the user to concentrate on the code-expanding rewriters and their proper application. Automating the application of the code-expanding rewriters is outside the scope of this dissertation.

We have added a simple graphical user interface to the rewriters using the Java Swing interface. It allows the user to select where to apply Inlining, and to apply the fix-point iteration of the code-reducing

```
public class DotGenOrig {
  public static Code makeSumCode() {
    Code sumcode = Constructor.integerConstant(0);
    System.out.println(sumcode);
    for (int i = 1; i < 10; i++) {
      sumcode = Constructor.binOp(6, sumcode,
                                  Constructor.constant(i));
    }
    return sumcode;
  }
}
```

Figure 6.21: A stripped-down dot-product generator

rewriters. For debugging purposes, it also allows doing a sanity check on the AST structure. Additionally, a command-line interface running simultaneously allows more fine-grained interaction and examination of the AST.

Figure 6.20 shows the user interface in use, just after an inlining has been performed. Method invocations are underlined, and the user can click them to apply inlining. Pressing “Cleanup” will start the code-reducing process that will hopefully remove most of the clutter added by the inlining rewriter.

6.8 Evaluation

As seen in section 5.4, even the comprehensive set of rewriters described above cannot deal effectively with the unknowns encountered in staged code. In this section, we assess some problems that limit the viability of the rewriting approach.

6.8.1 Problems Inherent to the Rewriters

A number of problems have turned up during our tests that show limitations in what a rewriting system can do. Figure 6.21 shows a stripped-down version of the dot-product generator of section 4.3.1. It produces code that sums the integers from 0 to 9. This simple program shows a number of problems that can be expected to appear in any attempt at rewriting except for the most trivial.

After inlining `binOp` and `constant` and further methods in them, and reducing the results with the rewriters, we end up with the code shown in figure 6.22. We can now see some problems that block further rewriting:

```
public class DotGenOrig {
  public static Code makeSumCode() {
    Code sumcode = Constructor.integerConstant(0);
    System.out.println(sumcode);
    int i = 1;
    while (true) {
      if (!(i<10)) break;
      final int gen8857_constval = i;
      final Code gen8859_flatten_35;
      gen8859_flatten_35 = new Code() {
        public ClosedCode eval(Environment env) {
          return new ClosedCode(Instr.genConst(gen8857_constval), env,
                                Type.int_type, ConstVal.make(gen8857_constval));
        }
      };
      final Code gen8858_c1 = sumcode;
      sumcode = new Code() {
        public ClosedCode evalInAdd(Environment env) {
          final ClosedCode cp1 = gen8858_c1.evalInAdd(env);
          Environment gen8861_env = env;
          final Instr gen8861_flatten_30 = Instr.genConst(gen8857_constval);
          final Type gen8861_flatten_31 = Type.int_type;
          final ConstVal gen8861_flatten_32 = ConstVal.make(gen8857_constval);
          final ClosedCode gen8861_flatten_33;
          gen8861_flatten_33 = new ClosedCode
            (gen8861_flatten_30, gen8861_env, gen8861_flatten_31, gen8861_flatten_32);
          if (cp1.isConstant()&&(gen8861_flatten_32!=null)) {
            final ConstVal c = cp1.constantValue().mathBinOp(6, gen8861_flatten_33.constantValue());
            return new ClosedCode(c.genLoadCode(), env, c.getType(), c);
          } else {
            if (cp1.type.isObjectType("java/lang/StringBuffer")) {
              final ClosedCode str2 = Constructor.makeStringValue(gen8861_flatten_33);
              return new ClosedCode(cp1.bytecode.append(str2.bytecode), env, cp1.type);
            } else {
              if (cp1.type.isObjectType("java/lang/String")||
                  gen8861_flatten_31.isObjectType("java/lang/String")) {
                final ClosedCode str1 = Constructor.makeStringValue(cp1);
                final ClosedCode str2 = Constructor.makeStringValue(gen8861_flatten_33);
                return new ClosedCode
                  (Instr.genNew("java/lang/StringBuffer").addDup(1, 0)
                    .addInvokenonvirtual("java/lang/StringBuffer", " ", new MethodType(Type.void_type))
                    .append(str1.bytecode).append(str2.bytecode), env, Type.stringbuffer_type);
              }
            }
          }
          final Type t = cp1.type.lub(gen8861_flatten_31);
          return new ClosedCode(cp1.bytecode.addConvert(cp1.type, t).append(gen8861_flatten_30)
                                .addConvert(gen8861_flatten_31, t).addMath(6, t), env, t);
        }
        public ClosedCode eval(Environment env) {
          final ClosedCode cp = this.evalInAdd(env);
          if (cp.type.isObjectType("java/lang/StringBuffer")) {
            return new ClosedCode(cp.bytecode.addInvokevirtual("java/lang/StringBuffer",
                                                               "toString", new MethodType(Type.string_type)),
                                  env, Type.string_type);
          } else return cp;
        }
      };
      i++;
    }
    return sumcode;
  }
}
```

Figure 6.22: The stripped-down dot-product generator after several inlinings and reductions (irrelevant method definitions removed)

- A key variable, `sumcode`, has multiple possible definitions. It can either be defined by the initialization step or by the previous iteration step. Thus, we cannot inline `gen_8858_c1`, leaving behind a large amount of irreducible inlined code that will only serve to limit optimization in the run-time system. Even loop unrolling would not help, as there would still be two class definitions with different free variables.

- The call to `Constval.make` could be inlined, but because `gen8857_constval` is the variable `i`, a number of further reductions in the inlined code are not possible.

- Even if the `Constval.make` call could be reduced so we can resolve `gen8861_flatten_32 != null`, it would turn out to be true, because `i` is the constant within the loop. That would leave `cp1.isConstant()` as the condition, so as long as `cp1` is unresolved, we cannot get rid of the first if branch.

- Without knowing the type of `cp1`, we cannot remove the code that handles the possibility of this addition being a string concatenation.

Other tests point out that since the environment is passed through every combinator, any combinator that cannot be sufficiently reduced will block the propagation of the environment. If this happens, even locally defined variables cannot be found in the environment, greatly reducing the potential for optimizations.

As mentioned in 6.2.1, type lookup in Java is somewhat complex. Even with a working type lookup system, there are some types, fields and methods that cannot be found, namely those that are being defined at run time. To be able to understand these, we would have to provide a description of the classes being generated. Even in the cases where class definitions are simple enough to be understood at compile time, it would be impractical to discover and process those extra definitions. In many cases, members or class names are being generated as part of the run-time code generation process, and so cannot be known at compile time at all.

6.8.2 Functional Style Java

Larry Wall is quoted as saying “Real programmers can write assembly code in any language”. This implies that no matter how many modern features a language has, it is possible to entirely avoid using them in favor of an old-fashioned assembly-like style. The result is usually inefficient and unmaintainable.

In a similar fashion, the attempt to use a rewriting system to optimize Java encourages a certain style of programming more amenable to rewriting, namely functional programming style. This is expressed in an avoidance of side-effects, the use of tuple-like classes to carry information, the use of "function objects", and a tendency to make methods final, static and public, to allow maximum optimization.

While such a style is richly rewarded in highly functional languages, it is stifling in Java, as the language syntax and libraries are ill suited for it. Furthermore, the compilers and virtual machines, not expecting wide-spread use of function objects, tail-recursive methods and final fields, do not perform optimizations on them that would be natural in compilers for functional programming languages. This style also restricts the use of some of the features of Java that could otherwise be of help, such as method overriding, field updating, and the standard data structures.

The functional style best handled by the rewriters fits well with the combinators, as they are also based on concepts more frequently found in functional languages. However, more liberal use of objects have also been helpful in implementing several parts of the system. This suggests that using a functional language with objects, such as OCaml, may yield good results.

6.9 Conclusion

As part of our investigation of run-time code generation using compositional techniques, we have designed and implemented a rewrite system that performs optimizations on Java source code. Based on a normalization of AST structures, three layers of program analysis and a number of simple rewrite rules, we can perform aggressive optimizations on any Java code. By explotiing the compositional nature of Jumbo, we have been able to perform significant reductions of code generators. This shows that rewrite systems can be used to perform optimization, and that compositional code generators are particularly amenable to aggressive compile-time optimization.

The rewriters do not achieve their full potential on the Jumbo code generators due to limited type information and problems in passing environments through unexpandable methods. The type information may be made available through subclassing the Code type to indicate the required type of expressions, possibly using the generics system of Java 1.5. The environment problem can be addressed by noting that most combinators pass the environment through them unchanged. If this fact can be made available to the rewriters, significantly more reductions of the code generators should possible.

Comparing the task of making the three analyses in the rewriter system with the equivalent analyses on Java byte-code in the author's Master's thesis[10], in most areas it is easier to analyze Java source. In particular, the fact that loops and switch constructs are explicit rather than built out of conditionals and gotos makes loop handling much easier. On the other hand, inner classes are difficult to handle at the source level, while they are compiled down to ordinary classes in Java byte-code. Java byte-code is in flattened form already, but the stack elements need to be handled. It is worth noting that the Soot system[87], which also operates on Java byte-code, transforms it into an internal form where loops and switches are explicit and stack handling is abstracted away.

The use of a set of normalizers to bring Java source trees towards a normal form that has less syntactic sugar could be useful in other systems doing transformations on Java source code. While some redundant ways of expressing a programming idiom can be useful for the programmer[6], they complicate automatic transformations of source code. The fewer ways a particular action can be performed, the fewer cases the transformation system has to deal with. This normalizing process, as well as the flow and use-def analyses, could be useful in a number of other settings, as they do not depend on the code being in any particular style.

[6]Indeed, it forms one of the central tenets of Perl: "There is more than one way to do it."

Chapter 7

Conclusion

In this dissertation, we have shown the usefulness of distributed run-time code generation. Through the implementation of Jumbo, a compiler for a code-generating version of Java, we have shown that run-time code generation can give a speedup of several factors, and in some cases up to an order of magnitude. Additionally, because code fragments are distributed in binary form, the system can be used to distribute code without disclosing the source, while retaining efficiency.

A system like Jumbo has several potential uses: It can be the basis for high-level domain-specific languages; it can allow client-server systems to create efficient code that is a mixture of undisclosed client code and server code; it can be used to codify programming constructs that are otherwise only informally defined; or it can be used as a generic staging system to increase performance of stageable programs.

Jumbo is implemented using a compositional semantics implemented in Java. To the best of our knowledge, Jumbo is the first compositional compiler for a major imperative language. Compositionality has turned out to be a viable way to structure the compiler, and allowed for a simple implementation.

The use of a compositional semantics makes it almost trivial to produce a run-time compiler that allows combining code from disparate sources. The code fragments are treated as first-class citizens in Jumbo. They can be created at any place, passed between functions or between machines, and eventually combined and compiled to efficient code. This allows a flexibility and power hitherto unseen for imperative run-time code generation systems. Additionally, compositional semantics means that each code fragment is a self-contained code generator that can be optimized using standard techniques.

7.1 Status

We have extended the Java language with a simple syntax for creating and composing code fragments. This syntax makes it simple for the programmer to create code at run-time. We have implemented the extended Java in a compositional compiler called Jumbo. The current version of Jumbo supports most of Java 1.2 plus our extensions. Inner classes are supported, though with some limitations; in particular multiple levels of nesting are not supported. A few esoteric features of Java are not supported either.

Jumbo allows the use of hygienic variables, though for purposes of experimentation it still allows unhygienic variables as well. The quotation syntax in some cases requires tagging with syntactic categories for most antiquotations and some quotations. A full description of the Jumbo extension of Java is given in chapter 3.

Jumbo is able to compile itself. Jumbo does not attempt to perform any dependency analysis, nor does it check for all semantic errors in the source. Erroneous Java source can yield illegal byte code, though the byte-code verifier in the Java virtual machine will catch such code.

Chapter 4 shows several examples of the possible uses of code generation with Jumbo. The examples show that Jumbo can be used in a wide variety of situations, and that it can give a significant speedup. The crossover point, where the cost of generating code is recovered by the faster code, is reached for realistic size problems.

In chapter 5, we consider possible ways of improving the overall performance of the system. We can optimize either the code we generate or the code generators. Opportunities to optimize the code are severely limited by time constraints, the fragmented nature of the code involved in code generation, and the mid-level nature of Java bytecode. We conclude that optimizing the generated code is not a viable way to improve performance.

In section 5.2, we look at various ways to improve the speed of the code generators. We conclude that one of the most promising approaches is to take advantage of the regular structure of the code generating combinators by performing aggressive optimizations. Section 5.4 shows some examples of how these optimizations can happen, and also how the same system that performs optimizations can be used as a type checker. For a simple example, we get a speedup of 26% when using the Kaffe VM, but no speedup in Sun's JVM due to the nature of Sun's HotSpot JIT compiler.

In chapter 6, we describe the details of our optimizers. They are implemented as a rewrite system, using

a normalization process and three layers of intra-procedural analysis. In a rewriting system, each rewriting rule is as simple as possible to allow easy reasoning and correctness checking. Repeated application of the rules can perform significant optimizations. The system performs inlining of methods, loop unrolling and related local code reduction. Inlining and loop unrolling must currently be selected manually, but the remaining rewrite rules can be applied automatically.

7.2 Using Java for Code Generation

Most research on run-time code generation has been based on functional languages like ML or Scheme. We chose to use Java partly because of its widespread usage, and partly because its object model and bytecode backend offered significant simplification of the code generation system. An earlier similar system implemented in SML/NJ did not scale as easily, and did not get close to the completeness of Jumbo.

The choice of the same language for both source language and implementation language significantly simplifies implementation. In particular, it ensures that handling of constant values is the same at compile time and runtime. Additionally, it allows a hiding of implementation details inside classes, and the use of function objects to provide varied functionality.

While the implementation was simplified by these choices, certain aspects of the usage were hampered by Java's imperative nature. In particular, the differentiation between statements and expressions means that some idioms from functional languages cannot be directly transferred. For instance, the NESL example in section 4.3.5 suffers from the lack of a "block expression" that would allow execution of loops inside expressions. Additionally, special syntactic category annotation is required to differentiate the various syntactic elements, whereas in functional languages, almost everything is an expression.

Developing the rewriters has provided valuable insight into details of the Java language. The normalizers indicate several places that are problematic for compiler writers, and show that knowing the type of expressions in the code is sufficient to disambiguate these constructs. However, it should be noted that complete type information cannot be had without knowing the definitions of all superclasses involved. Full normalization yields a simplified Java that would be an excellent target for various kinds of code manipulation.

For future language designers, it is worth noting that one of the biggest problems, both in the rewriters and in Jumbo itself, has been the overloading of the dot ('.') operator. If class names had used a different operator (maybe ':' or '::'), a multitude of problems would have been avoided, as a dot would always mean

field lookup or method invocation. This seems like a particularly unfortunate choice since only relatively rarely is the dot operator used in class names in practice, but any tool working with Java must handle dots correctly.

7.3 Code Generation in the Java Virtual Machine

Assessing the current implementation is complicated by the HotSpot run-time system. As mentioned in section 4.2, the dynamic optimization system makes it more difficult to get timings that relate to real-world performance. Additionally, we have experienced some counterintuitive behavior due to HotSpot. In some cases, we have achieved significant speedups by the use of "outlining", i.e., extracting loop bodies into separate methods. This extraction triggers the optimizer and improves running time, whereas most compilers strive to perform the opposite action, inlining, to improve performance.

These problems suggest the idea of adding the code generation system to the run-time system instead of having it as Java classes. This would make the system usable only on those virtual machines that support it, restricting usage to places where we know which virtual machine we will be using. However, it opens up a number of performance-enhancing possibilities:

- The need to generate and parse Java bytecode would disappear. The code generators could interface directly with the internal representation of the run-time system.
- The code generators could be built to match the optimizer in the run-time system.
- The code generators could direct the optimizer to the areas most likely to need optimizing.
- The code generators could be written in a language that does not have the interpretive overhead of Java.
- Several problems that currently contribute a significant portion of the code generators, such as class lookup and variable allocation, would be handled by the run-time system.

Allowing the code generators to interface directly with the internals of the virtual machine would allow us to create code in the virtual machines internal format rather than as Java byte codes. Since the verification process of the JVM is defined on byte codes, it would be possible to create code that has not been verified,

adding a severe security hole to the Java system. If we want to be able to use code generation outside of a totally trusted system, we would need to be able to prove, at a level comparable to the current verification system, that the code generators cannot produce invalid code. Masuhara and Yonezawa explore a type-safe code generation system in their DynJava system [65], which shows how some of the constraints can be checked, at the price of a less appealing syntax.

7.4 Future Research

The use of compositional semantics for code composition and generation has proven viable, and further research in this direction could be warranted. Below are some possible directions such research could take.

The idea of using code fragments in binary form to allow distribution of code that cannot be easily understood by the reader should be further investigated. There are security aspects to consider, for instance how to avoid code fragments doing more than they were supposed to. Indeed something as simple as a denial-of-service attack through infinite loops may need to be prevented. Further uses of the system should also be considered, in particular the possibilities of creating a client-server exchange system where the client and the server cooperate on building efficient code without either being tied to a particular internal representation.

The type system of Java has been one of the main roadblocks of the rewriting system, and it would be worthwhile to consider a system with explicit types. The DynJava system[65] shows one way to do this, though at the loss of the possibility of creating new types at run time. If a usable type system for the quoted code can be designed, it may allow much faster code generation through either a more powerful rewriting system or a template-based code generation system.

Java 1.2 added inner classes with variable capture. Java 1.5 adds generic types, a feature long known in the functional language community as polymorphic types. Thus, there is certainly an influence on Java development from functional languages. Since most research on staged compilation has been done in functional languages, one could consider what other features could be added to Java to make it more amenable to code generation. In the case of Jumbo, a tuple-like class that allows fast copying with update of final fields would significantly improve the environment handling.

The rewriting system may also be useful for other purposes. In particular the normalizing rewriters can be helpful for any project that manipulates Java code. The analyses might be used for optimization systems

or for binding-time analysis for a partial evaluation system.

Further work on optimizing the code generators could be valuable. The possibility of using general-purpose optimizers or partial evaluators to optimize the code generation for each code fragment is unique to Jumbo and should be explored. The rewriting system may be expandable to avoid the current roadblock and to perform more inlining and more powerful analyses.

A template-based code generation system similar to Tempo or BCS might faster than our system. However, it is unclear how templates can be implemented in a compositional manner to allow code from different sources to be combined. This is an area that merits contemplation.

The anonymous database system points to interesting possibilities for using code generation in database systems. The motivating example in [45] is a code generator for joins. Since joins are the single most time-consuming part of databases, there may be ways to significantly improve database speed by using code generation to create highly optimized join code.

An interesting possibility for future research is to implement a Jumbo-like system for OCaml[52, 51, 68], an ML-like language with object-oriented features. It combines the light-weight and expression-based syntax of functional languages with object-oriented features that simplify the internal structures. Additionally, it has a simple bytecode format, but can also be compiled directly to efficient machine code.

7.5 Concluding Remarks

We have shown that it is viable to use compositional semantics to make a run-time code generation system for Java. This system not only allows for specialization with regards to run-time invariants, but also allows the transmission of code-generators for the purpose of exchanging code fragments in an opaque manner. This allows efficient code to be generated at a remote site without requiring access to source code, query terms or other sensitive information.

We have also shoown that a rewriting system can be used to do aggressive general-purpose optimization of Java. By applying the rewriting system to code generators, we have also shown that the compositional nature of Jumbo allow static optimization of the code generators. Thus, despite the fact that we allow code to be created in separate sources, the fact that our code fragments carry their meaning with the makes it possible to perform static optimizations on them.

As seen in section 5.2, there are many ways in which the compilation process can be improved, ranging from partial evaluation methods to embedding the compiler in the run-time system. It is not yet clear which of the proposed methods would yield the most benefit. While embedding the compiler has the greatest potential for speed improvement, it would not be as portable as the current Java implementation. Increasing compilation speed should be the main goal of future research.

References

[1] Alfred V. Aho, Ravi Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] Danny Backx, Rick Scott, Alexander Mai, and many others. Lesstif. URL: `http://www.lesstif.org/`.

[4] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer society, June 1998.

[5] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 191–199. ACM Press, 1993.

[6] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 4–12, San Antonio, Texas, January 1999. URL: `http://www.brics.dk/~pepm99/programme.html`.

[7] Malcolm Beattie. The Perl compiler: Translating Perl to C and bytecode. *The Perl Journal*, 1(2), Summer 1996.

[8] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie-Mellon University, April 1993.

[9] P. Charles and D. Shields. The Jikes compiler, 1999. URL: `http://www.ibm.com/developerworks/oss/jikes`.

[10] Lars R. Clausen. Intensional and extensional aspects of Java byte code. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1998.

[11] William Clinger and Jonathan Rees, editors. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991. URL: `ftp://cs.indiana.edu/pub/scheme-repository/doc/r4rs.ps.Z`.

[12] Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[13] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 54–72. Springer-Verlag, Heidelberg, Germany, February 1996.

[14] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[15] Standard Performance Evaluation Corporation. Spec jvm98 benchmarks, 1998. URL: `http://www.spec.org/osg/jvm98`.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[18] Krzysztof Czarnecki and Ulrich W. Eisenecker. Static metaprogramming in C++. In *Generative Programming: Methods, Techniques, and Applications*, chapter 8, pages 251–279. Addison-Wesley, 2000.

[19] Markus Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[20] Nachum Dershowitz. *A Taste of Rewrite Systems*, volume 693 of *Lecture Notes in Computer Science*, pages 199–228. Springer Verlag, 1993.

[21] Dawson Engler. Incorporating application semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages*, pages 103–118, Santa Barbara, California, USA, October 15–17 1997.

[22] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of POPL 1996*, pages 131–144, January 1996.

[23] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. Technical Report TR-97-04-06, Department of Computer Science and Engineering, University of Washington, January 1997.

[24] Jay Fenlason, Kenny Woodland, Mike Thome, Jon Payne, and many others. Nethack. URL: `http://www.nethack.org/`.

[25] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Co., 1990.

[26] Brian Fox, Chet Ramey, and many others. Bourne again shell. URL: `http://www.gnu.org/software/bash/bash.html`.

[27] M. Frigo and S. G. Johnson. The fastest fourier transform in the west. Technical Report MIT/LCS/TR-728, Massachusetts Institute of Technology, 1997.

[28] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, May 1983.

[29] J. Gosling, B. Joy, G. Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, June 2000.

[30] Paul Graham. Beating the averages, April 2001. URL: `http://www.paulgraham.com/paulgraham/avg.html`.

[31] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[32] Object Management Group. *Common Object Request Broker: Architecture and Specification Revision*. John Wiley & Sons, Incorporated, September 1993.

[33] Jonathan C. Hardwick and Jay Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, Carnegie-Mellon University, August 1996.

[34] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 411–428, September 1993.

[35] John Hatcliff, Torben Mogensen, and Peter Thiemann, editors. *Partial Evaluation: Practice and Theory*, volume 1706. Springer-Verlag, 1999.

[36] Urs Holzle. Adaptive optimization for self: Reconciling high performance with exploratory programming. Technical Report STAN-CS-TR-94-1520, CS Department, Stanford University, 1994.

[37] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[38] J. Jones and S. Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, 2000.

[39] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[40] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components I: Source-level components. In *Generative and Component-Based Software Engineering (GCSE'99)*, volume 1799 of *LNCS*, pages 49–62, September 1999.

[41] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components II: Binary-level components. In *Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, volume 1924 of *LNCS*, pages 28–50. Springer Verlag, September 2000.

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

[43] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[44] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 1241. Springer-Verlag, June 1997.

[45] Graham Kirby, Ron Morrison, and David Stemple. Linguistic reflection in Java. *Software — Practice and Experience*, 28(10):1045–1077, 1998.

[46] Konstantin Knizhnik. JavaGO: Java global optimizer, October 1998. URL: `http://www.ispras.ru/~knizhnik/javago/ReadMe.htm`.

[47] Donald E. Knuth. *The Art of Computer Programming : Fundamental Algorithms*, volume 1. Addison–Wesley, third edition, 1997. ISBN: 0–201–89683–4.

[48] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, Cambridge, Massachusetts, August 1986. ACM SIGPLAN/SIGACT/SIGART.

[49] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, Helsinki, May 1996.

[50] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):1–5, 1998.

[51] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, Inria, Institut National de Recherche en Informatique et en Automatique, 1991.

[52] Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[53] M. E. Lesk and E. Schmidt. Lex — a lexical analyser generator. Technical report, Bell Labs, Murray Hill, NJ, USA, 1975.

[54] Tim Lindholm and Franki Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading/Massachusetts, second edition, 1999.

[55] Morten Marquard and Bjarne Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, Department of Computer Science, Universitetsparken 1, 2100 Copenhagen O., Denmark, April 1992.

[56] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization: A portable approach to generating optimized specialized code. In *Programs as Data Objects, Second Symposium, PADO 2001*, June 2001.

[57] Peter Mattis, Spencer Kimball, and Manish Singh. Gnu Image Manipulation Program. URL: `http://www.gimp.org`.

[58] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[59] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings, NATO Conference on Software Engineering*, Garmisch, Germany, October 1968.

[60] Steve Meloan. The Java HotSpot performance engine: An in-depth look. June 1999. URL: `http://developer.java.sun.com/developer/technicalArticles/Networking/Ho%tSpot/index.html`.

[61] Kim Mens, Cristina Lopes, Bedir Tekinerdogan, and Gregor Kiczales. Aspect-oriented programming. In J. Bosch and S. Mitchell, editors, *ECOOP'97 Workshop Reader*, volume 1357, pages 481–494. Springer-Verlag, 1998.

[62] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional Computing Series, 1996.

[63] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 132–142. IEEE Computer Society Press, 1998.

[64] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, , and Y. Kimura. OpenJIT: an open-nended, reflective JIT compiler framework for Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 362–387, 2000.

[65] Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. DynJava: Type safe dynamic code generation in Java. In *Proceedings of the 3rd JSSST Workshop on Programming and Programming Languages (PPL2001)*, January 2001.

[66] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999. URL: `http://amsterdam.lcs.mit.edu/tickc/`.

[67] preEmptive Solutions. Java obfuscator, Java optimizer, Java shrinker. URL: `http://www.preemptive.com/tools/`.

[68] D. Remy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[69] Armin Rigo. **P**syco: The Python specializing compiler, December 2001.

[70] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[71] P. H. Schmitt. A survey of rewrite systems. In *1st Workshop on Computer Science Logic (CSL '87)*, pages 235–262, Berlin - Heidelberg - New York, October 1988. Springer.

[72] Ulrik P. Schultz and Charles. Consel. Automatic program specialization for Java. Technical Report PB-551, DAIMI, University of Aarhus, 2000.

[73] Mauricio J. Serrano, Rajesh Bordawekar, Samuel P. Midkiff, and Manish Gupta. Quicksilver: a quasi-static compiler for Java. *ACM SIGPLAN Notices*, 35(10):66–82, October 2000.

[74] Charles Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, 1995.

[75] Vivek P. Singhal. *A Programming Language for Writing Domain-Specific Software System Generators*. PhD thesis, University of Texas at Austin, 1996.

[76] Marty Sirkin, Don Batory, and Vivek Singhal. Software components in a data structure precompiler. In *Intl. Conf. on Software Eng.*, pages 437–446, 1993.

[77] Bjarne Stroustrup, editor. *The C++ Programming Language*. Addison Wesley, second edition, 1993.

[78] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[79] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1), 2000.

[80] The Java HotSpot performance engine architecture: A white paper about Sun's second generation performance technology. Technical report, Sun Microsystems Incorporated, April 1999.

[81] Sun Microsystems. *Java 2 SDK Documentation*, 1999.

[82] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of PEPM-97*, volume 32 of *ACM SIGPLAN Notices*, pages 203–217, June 1997.

[83] The GTK+ Team. The GIMP toolkit. URL: `http://www.gtk.org`.

[84] Eastridge Technology. Jshrink, 1997. URL: `http://www.e-t.com/jshrink.html`.

[85] Scott Thibault, Charles Consel, and Gilles Muller. Safe and efficient active network programming. In *Symposium on Reliable Distributed Systems*, pages 135–143, 1998.

[86] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programs, Languages and Systems*, 34(10):292–305, 1999.

[87] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Computational Complexity*, pages 18–34, 2000.

[88] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[89] Todd L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 13–18, 1999.

[90] Jim Waldo and Ken Arnold, editors. *The Jini Specifications*. Pearson Education, 2 edition, December 2000.

[91] T. Wilkinson. Kaffe v0.8.3 — a free virtual machine to run Java code, March 1997. URL: `http://www.kaffe.org`.

[92] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik R. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *IEEE PACT*, pages 128–138, 1999.

Vita

Lars Reder Clausen was born in Århus, Denmark, in 1969, and grew up there. He received his B.Sc. in Computer Science and Mathematics from University of Aarhus in June, 1993, and his M.Sc. in Computer Science at the same place in June, 1998. Lars now lives in Århus again, where he works as an IT developer at the State and University Library.