A SUIF Java Compiler

Holger M. Kienle
`kienle@cs.uscb.edu`

Technical Report TRCS98–18
August, 1998

Abstract

To compete with optimized C, object-oriented languages need classical optimizations as well as specific object-oriented optimizations. Therefore, a compiler infrastructure that enables object-oriented compiler research is of great interest for the research community. The SUIF compiler system with the OSUIF extension offers such a research environment. SUIF defines an intermediate representation for procedural languages and offers standard optimization passes. OSUIF extends SUIF with an intermediate representation tailored towards object-oriented languages, hence allowing specific object-oriented optimizations.

To enable Java-related compiler research in (O)SUIF, we have implemented `j2s`, a Java bytecode front end for the OSUIF/SUIF 2.0 compiler system. Our compiler uses off-line compilation, so that the whole program is known at compile time. Thus, static whole-program optimizations such as class hierarchy analysis and type inference can be implemented with (O)SUIF and applied during the compilation.

`j2s` supports the complete Java Virtual Machine specification, except for threads and exception handling, which are currently not supported by the runtime system and (O)SUIF. Because we implemented an off-line compiler, the dynamic loading of classes is also not supported.

The compiler is divided into the (O)SUIF front end and the runtime system. The main part of the thesis describes the compilation passes, which consist of Java class loading, resolution of symbolic references in class files, control flow graph generation, data flow analysis to recover lost type information in class files, and finally the actual (O)SUIF code generation. The compiler generates either plain SUIF 2.0 or OSUIF code.

Java, in contrast to C, requires a sophisticated runtime system. This thesis contains a detailed description of the interface between `j2s` and the runtime system; the runtime system itself has been written by Andrew Rutz as part of his Master's thesis.

The current version of `j2s` compiles Java applications that require neither threads nor exception handling during program execution. Despite these restrictions a large number of Java applications compiles and executes. For example, our compiler can handle the transitive closure of the JDK1.2beta2 `javac` compiler, which consists of 538 Java classes. The `javac` executable generated with our compiler runs about twice as fast as the JDK1.2beta2 Virtual Machine.

Acknowledgments

I was fortunate enough to spend my last two years as a Visiting Researcher of the OOCSB group at the University of California Santa Barbara. I am very grateful to Prof. Urs Hölzle for giving me this wonderful opportunity in the first place and for supervising my external Diploma Thesis during that time. He is a great advisor—fun to work with—who pointed me into the right direction and raised the right questions.

I am very much indebted to Prof. Plödereder for supporting my Diploma Thesis and for navigating me through the bureaucratic minefield of an external thesis.

Chris Wilson patiently answered many questions when I started with the SUIF 2.0 system. I had many interesting discussions with the core OSUIF team, namely Andrew Duncan and Costin Iancu.

I would especially like to thank the proofreader, which for no apparent reason spent their time reading part of this thesis instead of doing something fun: Sylvie Dieckmann, Andrew Duncan, Chris Wilson, Ralph Keller, Andrew Rutz, and Karel Driesen.

Thanks to Lara and Alicia, who (quite unknowingly) managed to keep my spirits high during the writing of this thesis. Last but no least, I wish to thank my parents for their love and support. I love you, folks.

Contents

1 Introduction **1**

2 Intermediate Representations **4**
2.1 Motivation 4
2.2 Discussion of existing IRs 5
2.3 Classification of IRs 7

3 The SUIF 1.0 Compiler System **9**
3.1 Motivation 9
3.2 The SUIF 1.0 Kernel 9
3.2.1 File Level 10
3.2.2 Procedure Level and ASTs 11
3.2.3 Instructions 11
3.2.4 Symbol Tables 12
3.2.5 Types 12
3.2.6 Symbols 13
3.2.7 Variable Definitions 13
3.2.8 Annotations 13
3.3 The SUIF 1.0 Toolkit 13
3.4 Assessment of SUIF 1.0 14

4 The SUIF 2.0 Compiler System **16**
4.1 Overview 16
4.2 The *sty* Library 17
4.3 The *zot* Library 18
4.4 The *pyg* Library 21
4.5 The *suif* Library 25
4.6 The *suifpasses* Library 31
4.7 The *core_dialect* Library 31
4.8 Assessment of SUIF 2.0 32

5 OSUIF **34**
5.1 Motivation 34
5.2 Introduction 35
5.3 Overview of OSUIF Extensions 35
5.4 Classes in OSUIF 37
5.4.1 Symbol Table 38

5.4.2 Representation of Inheritance . . . 39
5.5 Instance Fields in OSUIF . . . 39
5.6 Instance Methods in OSUIF . . . 40
5.7 Static Members in OSUIF . . . 40
5.8 New Instructions . . . 40
5.8.1 Method Call Instruction . . . 40
5.8.2 New and Delete Instruction . . . 41
5.9 Exception Handling in OSUIF . . . 41
5.10 Enhanced Type System . . . 42
5.10.1 Type sets . . . 43
5.11 Lowering Passes . . . 43
5.12 Raising Passes . . . 44
5.13 Assessment of OSUIF . . . 44

6 The Java Virtual Machine 46
6.1 Virtual Machines . . . 46
6.2 Overview of the JVM . . . 47
6.3 Type System . . . 48
6.4 Runtime Layout . . . 48
6.5 The Class File Format . . . 49
6.6 Verification of Class Files . . . 50
6.7 Constant Pool Resolution . . . 51
6.8 Instruction Set . . . 51
6.9 Exceptions . . . 52

7 Overview of the `j2s` Compiler 53
7.1 Supported Java Subset . . . 55
7.2 Compilation Techniques . . . 55
7.3 Application Domain . . . 56
7.4 Integration into (O)SUIF . . . 57
7.5 Compilation Passes . . . 58

8 The `j2s` Front End 60
8.1 Class Loader . . . 60
8.1.1 Implementation . . . 61
8.2 Resolver . . . 61
8.2.1 Representation of Types . . . 62
8.2.2 Bytecode Iterator and Visitor . . . 63
8.2.3 Lazy resolution . . . 64
8.2.4 Iterator/Visitor Example . . . 64
8.3 Control Flow Graph Generation . . . 65
8.3.1 Control Flow Graph Example . . . 66
8.3.2 `jsr` and `ret` Bytecodes . . . 66
8.3.3 Exception Handling Control Flow . . . 70
8.3.4 Implementation . . . 74
8.4 Data Flow Analysis . . . 75
8.4.1 Type Recovery . . . 75
8.4.1.1 Symbolic Execution . . . 76
8.4.1.2 Merging . . . 77

8.4.1.3 Putting it all Together . . . 78
8.4.2 Computation of `ret` Targets . . . 79
8.4.3 Dead Basic Block Elimination . . . 79
8.4.4 Reducibility of Control Flow . . . 80
8.4.5 Implementation . . . 82

9 The `j2s` Back End **84**
9.1 Compilation Models . . . 84
9.1.1 Separate Compilation . . . 85
9.1.2 Full Compilation . . . 86
9.2 (O)SUIF Code Generation . . . 86
9.2.1 Classes . . . 87
9.2.1.1 Forward Declarations . . . 88
9.2.2 Arrays . . . 88
9.2.3 Methods . . . 88
9.2.3.1 Method Types . . . 89
9.2.3.2 Declarations vs. Definitions . . . 89
9.2.3.3 Native Methods . . . 89
9.2.4 Static Fields . . . 90
9.2.4.1 Declarations vs. Definitions . . . 91
9.2.5 Instance Fields . . . 91
9.2.5.1 Object Layout . . . 91
9.2.5.2 Declarations vs. Definitions . . . 93
9.2.6 Object Allocation . . . 93
9.2.7 Field Access . . . 94
9.2.7.1 Instance Fields . . . 94
9.2.7.2 Static Fields . . . 95
9.2.7.3 Array Fields . . . 95
9.2.8 Method Calls . . . 96
9.2.8.1 Parameter Passing . . . 97
9.2.8.2 Statically Bound Calls . . . 97
9.2.8.3 Dynamically Bound Calls . . . 97
9.2.8.3.1 Class Dispatch . . . 97
9.2.8.3.2 Interface Dispatch . . . 98
9.2.8.4 Implementation . . . 103
9.2.9 Operand Stack and Local Variable Array . . . 104
9.2.9.1 Local Variables . . . 104
9.2.9.2 Operand Stack . . . 105
9.2.9.3 Stack Spilling . . . 108
9.2.10 Load and Store Bytecodes . . . 110
9.2.11 Arithmetic Bytecodes . . . 111
9.2.12 Control Transfer Bytecodes . . . 112
9.2.12.1 `jsr/ret` Bytecodes . . . 113
9.2.13 Comparison Bytecodes . . . 115
9.2.14 Type Inclusion Tests . . . 115
9.2.14.1 Subclass Test . . . 115
9.2.14.2 Interface Test . . . 117
9.2.14.3 Array Test . . . 117
9.2.14.4 Implementation . . . 117

9.2.15 Exception Handling . . . 118
9.2.16 Runtime Checks . . . 119
9.2.17 Class and Interface Initialization . . . 120
9.2.18 Implementation . . . 121
9.3 Meta-Data Structure . . . 122
9.3.1 Class Meta-Data . . . 123
9.3.2 Method Meta-Data . . . 124
9.3.3 Field Meta-Data . . . 125
9.3.4 Global Meta-Data . . . 125
9.3.4.1 Transitive Closure of Classes . . . 125
9.3.4.2 Array Meta-Data . . . 126
9.3.4.3 Size of Data Structures . . . 126
9.4 Runtime System . . . 127
9.4.1 Runtime Meta-Data . . . 127
9.4.2 Interface . . . 127
9.5 System Startup . . . 129

10 Testing **130**
10.1 Testing of `javac` . . . 130
10.2 The *Jasmin* Assembler . . . 131

11 Performance Evaluation **133**
11.1 Benchmark Programs . . . 133
11.2 Results . . . 133

12 Related Work **137**
12.1 Other Java Compilers . . . 137
12.2 Decompilers . . . 138

13 Evaluation **140**
13.1 Conclusions . . . 142

A Performance Data **143**

Bibliography **144**

List of Figures

2.1 Typical compilation process . . . 4

3.1 SUIF 1.0 class hierarchy . . . 10
3.2 Symbol table hierarchy . . . 12
3.3 The SUIF 1.0 framework . . . 14

4.1 The zot interface . . . 19
4.2 Simplified zot library class hierarchy . . . 20
4.3 The pyg library class hierarchy (generic classes omitted) . . . 22
4.4 SUIF 2.0 class hierarchy (type-related classes) . . . 25
4.5 SUIF 2.0 instructions class hierarchy . . . 26
4.6 SUIF 2.0 control flow graph nodes class hierarchy . . . 28
4.7 SUIF 2.0 statements class hierarchy . . . 29
4.8 SUIF 2.0 value blocks class hierarchy . . . 29
4.9 SUIF 2.0 ownership graph example [Wil97c] . . . 30

5.1 OSUIF class hierarchy (generic classes omitted) . . . 36
5.2 OSUIF class symbol table interface . . . 38

7.1 The `j2s` compiler . . . 53
7.2 Integration of `j2s` into the SUIF framework . . . 57
7.3 `j2s` translation process . . . 59

8.1 Low-level implementation of the `ClassFile` table . . . 61
8.2 Java constant pool entries . . . 62
8.3 High-level implementation of the `ClassFile` table . . . 62
8.4 Java type hierarchy . . . 63
8.5 Internal representation of type `[[Ljava/lang/Object;` . . . 63
8.6 Source code of Fibonacci class . . . 67
8.7 Control flow of Fibonacci methods . . . 68
8.8 Jasmin assembly example for `jsr/ret` bytecodes . . . 69
8.9 Control flow for example in figure 8.8 . . . 70
8.10 Java source that contains exception handling control flow . . . 71
8.11 Two CFGs (with and without EHCF) for Java code in figure 8.10 . . . 72
8.12 Merging Example . . . 77
8.13 Computation of the `ret` target set . . . 79
8.14 DFST for Java code in figure 8.6 . . . 81
8.15 Irreducible control flow (Jasmine source and DFST) . . . 81

9.1 Object Layout . . . 92
9.2 Object layout for `Integer[3][2]` . . . 93
9.3 SUIF code to get/set fields . . . 95
9.4 SUIF code to load/store array elements . . . 96
9.5 Table-based dispatch . . . 98
9.6 Pseudo Java class hierarchy . . . 99
9.7 Vtables for class hierarchy in figure 9.6 . . . 100
9.8 Interface dispatch for class hierarchy in figure 9.6 . . . 101
9.9 Vtable duplication for interfaces . . . 102
9.10 Local variable translation . . . 104
9.11 Stack mapping with C local variables . . . 105
9.12 Stack mapping with expression trees for `iadd` . . . 106
9.13 Example Java source and bytecodes . . . 107
9.14 Stack translation . . . 107
9.15 Example code for stack spilling . . . 108
9.16 CFG of code in figure 9.15 (stack spilling required) . . . 108
9.17 SSA form vs. stack spilling . . . 110
9.18 SUIF code for `if_icmp<cond>` bytecodes . . . 112
9.19 Relative numbering algorithm . . . 116
9.20 Bytecode visitors for code generation . . . 121
9.21 Runtime meta-data (with vtable) . . . 127

10.1 `javac` test case . . . 131
10.2 Bootstrap of `javac` . . . 132

11.1 Speedups . . . 134
11.2 Execution profile of `javac` executable . . . 135

List of Tables

2.1 Classification of some IRs . . . 8

8.1 Bytecode Visitors . . . 74

9.1 Statistics for stack spills . . . 110
9.2 Statistics for type checks . . . 117

11.1 Benchmarks . . . 133
11.2 Impact of `structure` optimization pass on `javac` . . . 135

Chapter 1

Introduction

This thesis describes the design and implementation of a Java bytecode to SUIF 2.0 compiler, called `j2s` in the following.

The *Java Virtual Machine* (JVM) [LY96] is a platform-independent bytecode representation designed to support the Java programming language [GJS96]. In this scheme, every Java class or interface is compiled to a separate class file that contains the JVM representation.

The *SUIF compiler system*, which has been developed at Stanford University, supports experimental research on new compiler techniques and consists of various tools and standard optimization passes, which operate upon a common intermediate representation. There are two major revisions of the SUIF compiler system: SUIF 1.0 [WFW$^+$94] [SCG94b] and the recently introduced SUIF 2.0. SUIF 2.0 shares the basic principles and functionalities with SUIF 1.0, but the underlying design and implementation is completely different. Moreover, SUIF 2.0 has a cleaner design and a more extensible intermediate representation.

Targeting a compiler system, such as SUIF, has several advantages. For example, our compiler can take full advantage of the available SUIF optimization passes, native code back ends, and debugging tools (e.g., the C back end). Furthermore, new optimization passes can be easily written and incorporated into the existing system.

Unfortunately, the intermediate representations of both SUIF 1.0 and SUIF 2.0 support only imperative languages (such as Fortran and C); they lack constructs to represent object-oriented or functional features. Object-oriented languages, for instance, require additional—or completely different—high-level information than imperative languages. For this reason, an additional layer on top of SUIF 2.0, called *Object SUIF* (OSUIF), is in development, which extends the SUIF 2.0 intermediate representation with object-oriented constructs. OSUIF is developed at the University of California Santa Barbara. Both SUIF 2.0 and OSUIF are part of the *National Compiler Infrastructure*[1] project.

The `j2s` compiler supports the complete JVM specification, except for threads, locks, and exception-handling. Accurate garbage collection, which is not part of the JVM specification, is not supported, but we incorporated the Boehm–Demers–Weiser conservation garbage collector.[2] `j2s` is an off-line compiler and hence does not allow dynamic loading and linking of class files. Our compiler optionally targets SUIF 2.0 or OSUIF.

`j2s` implements a Java class loader, constant pool resolution (during compile time), a large subset of the bytecode verifier, a check for irreducible control flow, and type recovery of the (untyped) Java operand stack and local variables.

Among the contributions of our work are:

---

[1] Homepage: `http://suif.stanford.edu/suif/NCI`.

[2] Homepage `http://reality.sgi.com/boehm/gc.html`.

- Enabling Java-related compiler research for SUIF. We hope that our Java front end will prove useful to develop static optimization techniques for Java and object-oriented languages in general.
- Proof-of-concept for the design of the SUIF 2.0 system. The `j2s` compiler was the first larger project that used SUIF 2.0. As of this writing it is the first stable SUIF 2.0 front end.
- Design input and proof-of-concept for the OSUIF system. The `j2s` compiler was the first front end that targeted OSUIF.

The `j2s` compiler is complete enough to compile large programs, such as the `javac` compiler of the JDK1.2beta2 release, which consists 538 Java classes.

Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2 discusses intermediate representations (IRs). We describe design goals for IRs and briefly describe and classify several existing IRs.

Chapter 3 and **chapter 4** introduce SUIF 1.0 and SUIF 2.0, respectively. Although SUIF 1.0 has been rendered obsolete by its successor we discuss it here because it is well documented, still widely used, and many familiar concepts from SUIF 1.0 have been incorporated in SUIF 2.0. **Chapter 5** describes OSUIF.

Chapter 6 briefly describes the Java Virtual Machine. We included only the parts of the JVM that are relevant for the understanding of this thesis.

The next three chapters describe the `j2s` compiler in detail. **Chapter 7** gives a first overview of the compiler and describes its features, application domain, and integration into the (O)SUIF compiler system. **Chapter 8** introduces the front end passes and **chapter 9** describes the back end of the compiler. The largest part of chapter 9 is dedicated to the (O)SUIF code generation.

Chapter 10 describes how the compiler was tested and **chapter 11** gives a brief performance evaluation of the compiler.

Chapter 12 contains related work. We give an overview of currently available Java compilers and briefly compare them to `j2s`. This chapter also introduces decompilers in general and lists several Java decompilers.

Chapter 13 describes the author's experiences while implementing the compiler. This includes a short description of the development process, (unexpected) problems that showed up, and how easy it was to use the—at that point completely undocumented—SUIF 2.0 compiler system.

Task Description

In the following, we give the task description for the Diploma thesis as agreed upon with Prof. Plödereder, University of Stuttgart, Germany. This thesis has been executed as an "externe Diplomarbeit" for the University of Stuttgart during the author's stay as a Post Graduate Researcher at the University of California Santa Barbara.

> The thesis will implement a Java bytecode to SUIF compiler translating Java bytecode to the SUIF intermediate representation.
>
> The source language of the compiler is the Java class file format as described in the Java Virtual Machine (JVM) Specification. The target language of the compiler is the intermediate

representation (IR) of the SUIF 2.0 compilation system. The compiler will be implemented in C++ (because SUIF is implemented in C++).

The compiler will support the full JVM Specification, except for certain restrictions regarding the following features:

Threads: The compiler will assume the absence of threads.

Locks and synchronization: The compiler will interpret the respective bytecodes as `noops`.

Exception handling: The compiler will generate code that causes program abortion upon the occurrence of an exception at run-time. Related run-time checks will be translated correctly, so that any program not raising an exception at run-time will execute correctly.

Garbage collection: Any garbage collection provided by the run-time system will not require specific support by the compiler.

The compiler will implement a subset of the constraint checks that the JVM Specification imposes on a Java class file. These checks include the implementation of a control-flow based verification of the Java bytecodes. A significant part of the byte code verification is the recovery of type information from the Java bytecode representation. Local variables and the operand stack are untyped in the JVM. Hence, type recovery is necessary in order to generate a type-safe intermediate representation in SUIF.

The compiler will use the code generator provided by the SUIF system and a Java-specific run-time system developed concurrently at UCSB. Part of the task of the Diploma thesis is the coordination and description of an appropriate interface to this run-time system, but not the implementation of the run-time system.

The compiler will support, at a minimum, the standard classes specified in `java.lang`, `java.util` and `java.io`.

Acceptance testing of the Java-to-SUIF compiler will comprise compilation of the `javac` compiler and execution of the resulting compiler on a substantive test set of Java programs, in particular a bootstrap of the compiler. Also, a sufficient set of white-box test programs will be produced and used in testing of the compiler.

The thesis will be written in English. It will be delivered in printed and electronic form.

The thesis work will be supervised by Prof. Hölzle, University of California Santa Barbara, U.S.A.

Chapter 2

Intermediate Representations

This chapter introduces intermediate representations, gives examples of existing intermediate representations, and finally introduces a classification scheme for them.

2.1 Motivation

An important aspect of every compiler is its *intermediate representation*[1] (IR). The *front end* of the compiler translates a program from the source language into its IR (*analysis* part of the compiler). The *back end* generates the target language from the IR (*synthesis*) [ASU86]. Optionally, a set of optimization passes, which perform tranformation on the IR, can be run. Figure 2.1 illustrates the compilation process.

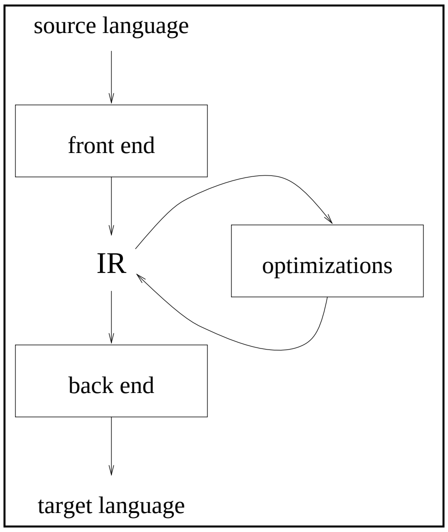

Figure 2.1: Typical compilation process

The benefits of employing an IR—in an ideal scenario—are flexibility and reusability:

[1]Some authors draw a distinction between an intermediate representation an a intermediate language. This thesis uses these terms interchangeably.

- It allows porting of a compiler to a new architecture while enabling the developer to re-use the front ends.
- Creating a front end for a new target language is also made easier, because the back ends can be reused.
- Machine-independent optimization passes (that operate on the IR) can be easily written.

Choosing the right IR is an important decision in the design of a compiler system because its structure remains the same throughout the entire compilation process, and thus affects all parts of the compiler. Ideally, an IR

- should have a clean and readily apparent semantics.
- should be economic because most optimizing compilers perform many passes over it.
- should be sufficiently general to represent important aspects of multiple front end languages.
- should be sufficiently general to support efficient code generation and/or interpretation for multiple back end architectures.
- should allow easy implementation of machine-independent program optimizations. For example, it should be easy to inspect operators/operands and to transform the IR.

2.2 Discussion of existing IRs

There is a wide range of different IRs, reflecting the different requirements and design goals of various compiler systems. For example, IRs of traditional compilers usually use (low-level) instructions that resemble a CISC or RISC instruction set. Other compilers have IRs that use higher-level instructions similar to the constructs of a high-level programming language (e.g., to model structured control-flow or to make array accesses explicit) and/or augment the IR with specific constructs to model object-oriented or distributed/parallel behavior. In the following, we give examples of several interesting IRs (refer also to table 2.1).

GNU CC [Sta95] from the Free Software Foundation is a highly portable compiler system. GNU CC expects the front end to generate an *abstract syntax tree* (AST) representation of the input program. As a first step, the compiler transforms the AST into *register transfer list* (RTL) [DF84] representation. RTL represents operations by atomic operations of a very primitive hypothetical machine. RTL is target-independent as long as all atomic operations can be implemented on every target machine. GNU CC has a set of standard scalar optimization passes that operate on the RTL. Finally, a specific back end translates RTL into machine code. The back end has to provide information of the target machine with a machine description, which gives an algebraic formula for each of the machine's instructions. Typically a single target machine instruction combines multiple RTL atomic operations.

`lcc`[2] [FH91] is a retargetable compiler for ANSI C. The interface between the target-independent front end and the back end consists of shared data structures, functions, and a *directed acyclic graph* (DAG) language. The ANSI C front end and a specific back end are tightly coupled in a single program (*monolithic compiler*). They are clients of one another and manipulate the shared data structures. (Both front end and back end also have their own private data structures.) While this approach means a tight coupling (in fact, the front end contains a few target-specific operations), it results in efficient, compact compilers. The authors claim that sharing data structures is manageable because there are few such structures. The DAG language realizes the IR of the compiler. It has 36 operators and therefor is

[2]Homepage: `http://www.cs.princeton.edu/software/lcc/`.

smaller than most other IRs. A procedure body is represented as an ordered forest of DAGs. Note that the ordering of the DAG yields an execution order for the code generation. The code generation first traverses the forest and linearizes each DAG[3] and then performs register allocation. (This order simplifies the insertion of spills and reloads.) The interface is designed to support only code generation. For example, there is no direct support to represent control flow. Hence, optimizations that are based on control flow information cannot be realized easily.

ILOC '93 [Bri96] is an IR employed by the *Massively Scalar Compiler Project*[4] at Rice University. It is very simple and low-level and has been designed for traditional, scalar optimizations, such as common subexpression elimination, strength reduction, and copy propagation. The IR supports may-alias information of variables (but not pointer-based aliasing). Registers are untyped and the number of registers is unlimited. ILOC instructions are typed. It is possible to pack several operations into a single instruction. These operations are (conceptually) executed in parallel. This makes it possible to model, for example, branch-and-execute instructions and auto-increment addressing modes. The IR is powerful enough to represent Fortran 77 code; however, some extension are needed to support C. These extensions are currently added.

The *Architecture-Neutral Distribution Format*[5] (ANDF) [OSF91] was specifically designed as a language- and machine-independent distribution format. The scope of ANDF is certainly broader than that of an IR, addressing issues like protection from reverse engineering, distribution of applications/libraries, etc. The IR itself is tree-structured and has a complete and unambiguous definition. ANDF can be used as a high-level or low-level IR. All operations are completely specified for both normal and exceptional cases. Thus optimizations can be expressed by ANDF-to-ANDF transformations. The design goal of ANDF was to cover a wide spectrum of languages, from C to Ada, Common Lisp, and ML.

The *Vortex compiler infrastructure*[6] [DDG+96] of the University of Washington is a language-independent optimizing compiler for object-oriented languages. Currently front ends for Cecil, C++, Java, and Modula-3 exist. The target language is C or native code for the SPARC architecture. The Vortex IR is essentially three-address code (including the usual complement of arithmetic, logic, branch, and procedure-call–related instructions) augmented with higher-level object-oriented instructions, such as message sends, instance variable accesses, object allocations, run-time class tests, and type assertions. As part of the optimization phases, high-level operations are lowered (when no longer needed) by expanding them into sequences of simpler instructions. Vortex implements traditional optimizations, selective recompilation, an iterative data flow analysis framework, and exceptions. It appears that Vortex and OSUIF (refer to chapter 5) have similar goals. We compare these two systems in section 5.13.

The *FLINT system*[7] [Sha97b] is organized around a strongly typed intermediate format that allows to model the semantics and interactions of various programming languages. The system focuses on support for HOT languages[8] [Wad97], such as Java, ML, Haskell, and Scheme. The FLINT IR is based on polymorphic lambda calculus and has a rich set of primitive types and functions. FLINT's author claims that because all runtime representations are expressed in FLINT IR types, different source languages can share the same system-wide garbage collector and foreign call interface (e.g., the C calling convention). Optimization passes are implemented as type-preserving FLINT IR transformations. Conventional data flow and loop optimizations, type inference, and λ-calculus-based contractions and reductions are supported. Apart from polymorphism and higher-order functions, the resulting code should be very close to a typical machine language. After the optimizations, the back end lowers high-level constructs, does standard register allocation and instruction scheduling, and finally generates machine code. The FLINT

[3]The generation of code for a basic block in DAG representation is discussed in [ASU86, Section 9.10].
[4]Homepage: `http://softlib.rice.edu/MSCP/`.
[5]Homepage: `http://www.osf.org/andf/`.
[6]Cecil/Vortex homepage: `http://www.cs.washington.edu/research/projects/cecil/`.
[7]Homepage: `http://flint.cs.yale.edu/`.
[8]HOT is an acronym for "Higher-Order and Typed." There is not clear-cut definition for this term, but HOT usually implies a safe, sound type system and a high level of abstraction of the programming model of the language.

runtime system provides support for system-wide garbage collection, foreign-function call interface, and connections to lower-level OS services. As of this writing, only a front end for SML '97 exists, which has been incorporated into the Standard ML New Jersey compiler [Sha97a]. Work on Java, Haskell, and "Safe C" are underway.

The *Illinois Concert System*[9] [Ple96] [CDG+97a] is an optimizing implementation for a concurrent object-oriented programming model. The system has a retargetable front end. Front ends for ICC++ and CA exist. The compiler uses several IRs. Every front end targets a simple core language. It is a "bare bones" concurrent object-oriented language (which uses Lisp syntax). Method sends (dynamic binding) are supported; procedure calls (static binding) are not explicitly supported. Control flow is restricted to `if` and `while` constructs to simplify graph construction. Concurrency is expressed with a `conc`-block (i.e., a list of statements that can be executed in parallel) and a `conc-while`. The core language is transformed to a CFG in SSA form, which in turn is transformed to a program dependency graph (PDG) in SSA. Optimizations use the PDG representation for their transformations. The code generation first rebuilds the CFG and then generates RTL. Registers are allocated over the RTL which is finally written out as a set of macros. These macros are converted into C++ which is used as a slow but portable assembler.

The *Fractal Intermediate Language* (FRIL) [CS93] is a simple IR that has *fractal* characteristics, i.e., program transformations and optimizations can occur at different abstraction levels. With a fractal IR, an operation is initially specified at a high-level. Subsequently, the high-level operation is lowered by the front end to potentially enable further optimizations. Note that the front end does not vanish from the compilation scene after the construction of the initial high-level IR.

The *Slim Binary Format* [Fra94] [FK96] is a portable and exceptionally dense code distribution format, which is currently used in the *Juice* project[10] [KF96] and for the *MacOberon* distribution. A slim binary is essentially a tree-based IR, which describes the actions of the original program similar to a parse tree. The block structure of a program is preserved. Furthermore, every node in the tree is strongly typed by a reference to a symbol table. The tree is represented with a semantic dictionary. This dictionary can contain template entries for code sequences that occur more than once in a program.

The IRs of the SUIF 1.0 and SUIF 2.0 compiler systems are discussed in detail in chapter 3 and chapter 4, respectively.

2.3 Classification of IRs

Brandis [Bra95] characterizes IRs with the three following orthogonal attributes:

1. Abstraction of operations. The abstraction level can range from using operators of the source language directly, through using machine instructions of the target machine, to using a sub-machine language such as RTL. Every abstraction level has its strength and weaknesses. For example, a high level of abstraction is easy to generate, easy to understand, and portable, but does not allow the compiler to optimize the actual machine instructions.

2. Abstraction of control flow. At a high level, control flow is expressed by using the constructs of the source language directly, leading to an AST representation. ASTs represent control flow implicitly. In contrast, CFGs represent control flow explicitly by exposing the branch structure of a program.

3. Abstraction of data flow. An IR can allow multiple assignments to the same variable (*multi-assignment*), use *static single-assignment* (SSA) form, or use *dynamic single-assignment* (DSA) form. Three-address code keeps data flow implicit and hard to recover. DAGs make data flow

[9] Homepage: `http://www-csag.cs.uiuc.edu/projects/concert.html`.

[10] Homepage: `http://www.ics.uci.edu/~juice/`.

IR	operations	control flow	data flow
SUIF 1.0	low-level (RISC-style) operations and selected high-level operations (e.g., array access)	AST three-address code	multi-assignment
SUIF 2.0	ditto	AST three-address code CFG	multi-assignment SSA
ANDF	low-level and high-level	AST three-address code	multi-assignment
GNU CC	RTL	AST CFG(?)	multi-assignment
Vortex	traditional low-level operators and selected high-level operations for OO features	three-address code	multi-assignment
`lcc`	low-level operations	forest of DAGs	DAG with multi-assignment(?)
ILOC '93	low-level operations	three-address code	multi-assignment SSA(?)

Table 2.1: Classification of some IRs[11]

more explicit because they show how a computed value in a basic block is subsequently used in the block.

Table 2.1 classifies some IRs that have been discussed in section 2.2.

The IR of the SUIF system is closest to Vortex and ANDF. The SUIF IR has been designed to be language- and machine-independent. ANDF has this property as well, whereas Vortex is tailored towards object-oriented languages. Standard SUIF supports only imperative languages. Other language paradigms (e.g., functional or object-oriented) can be realized by extending the IR. This distinguishes SUIF from ANDF and Vortex, that define a fixed IR. SUIF's IR provides a (high-level) AST, a (low-level) instruction list, and—for SUIF 2.0 only—a control flow graph representation. SUIF passes can convert between these representations. A SUIF pass that expects a certain representation can run a conversion pass first if necessary. In this sense, SUIF's scheme is more flexible than that of ANDF and Vortex.

Having a single representation can be desirable—especially for a production compiler—because optimization passes can be applied selectively (or even iteratively [CC95]) without the need for additional conversion passes. On the other hand, by having more then one representation a pass can use the most suitable representation to render its transformations. Furthermore, a single representation might not be powerful enough: A high-level representation allows optimizations that would be more difficult or even impossible in a low-level representation and vice versa. Let's consider the representation of control flow. High-level control flow constructs are necessary for many optimizations, such as loop parallelization, elimination of induction variables, and hoisting of code to an outer scope. On the other hand, these constructs have the disadvantage that branching is not explicitly represented and therefore disallow certain transformations. (See [Bra95, Section 5.2] for an example of such a transformation.)

[11] A (?) indicates that we could not determine whether the IR has indeed this property.

Chapter 3

The SUIF 1.0 Compiler System

This chapter explains how compiler researchers benefit from the SUIF compiler system, and discusses the SUIF 1.0 compiler infrastructure and intermediate representation.

3.1 Motivation

Research in the area of compiler technology usually requires the implementation of a complete compiler. This includes all the important standard program analyses and optimizations that manipulate the intermediate representation (IR) of the compiler [ASU86]. These standard passes are well-known, but because different compilers have different IRs they need to be re-implemented for every one of them. Implementation of these standard optimizations is often necessary because more sophisticated optimizations rely on them. Because independent development of standard program analyses and optimizations is costly, compiler researchers would benefit greatly from an infrastructure that allows them to share investments.

The *SUIF compiler system*[1] provides such an infrastructure for compiler-related research [WFW$^+$94]. It is modular, flexible, and well documented. Because the system is meant to be a research tool rather than a production compiler, it often trades efficiency for flexibility. The Stanford Compiler Group has successfully used SUIF 1.0 to perform research on topics including scalar optimization, array data dependence analysis, loop transformations and instruction scheduling. (These passes are part of the SUIF 1.0 compiler toolkit.)

The SUIF 1.0 compiler consists of the *kernel* and the *toolkit*. They are discussed in section 3.2 and 3.3, respectively.

3.2 The SUIF 1.0 Kernel

This section discusses the kernel of the SUIF 1.0 system, which provides the following functionality:

- It defines the intermediate representation on which compilation passes operate. The IR is similar to an AST and is implemented as a hierarchy of data structures in an object-oriented class library [SCG94b]. This library defines a C++ class for each element of the IR (see figure 3.1), providing an interface that hides the details of the underlying data structures.

  The intermediate format has a *high-level* and a *low-level* representation. The former includes language-independent high-level constructs for loops, conditional statements, and array access op-

[1] SUIF is an acronym for **S**tanford **U**niversity **I**ntermediate **F**ormat.

erations. These constructs capture high-level information that is useful for certain optimizations. They can be easily transformed/mapped to the equivalent low-level representation.

- It defines the interface between compilation passes. SUIF passes communicate via files. A *pass* typically performs a single analysis or transformation of the IR and then writes the results out to a file. This is inefficient but provides the flexibility to reorder passes by simply running them in a different order (provided that the passes themselves do not depend on a certain order). SUIF passes can generate user-defined annotations, which can be optionally written to the output file.

  Compiler researchers can assemble their experimental compiler by simply combining available SUIF passes with their own experimental passes, which can easily be made available for reuse.

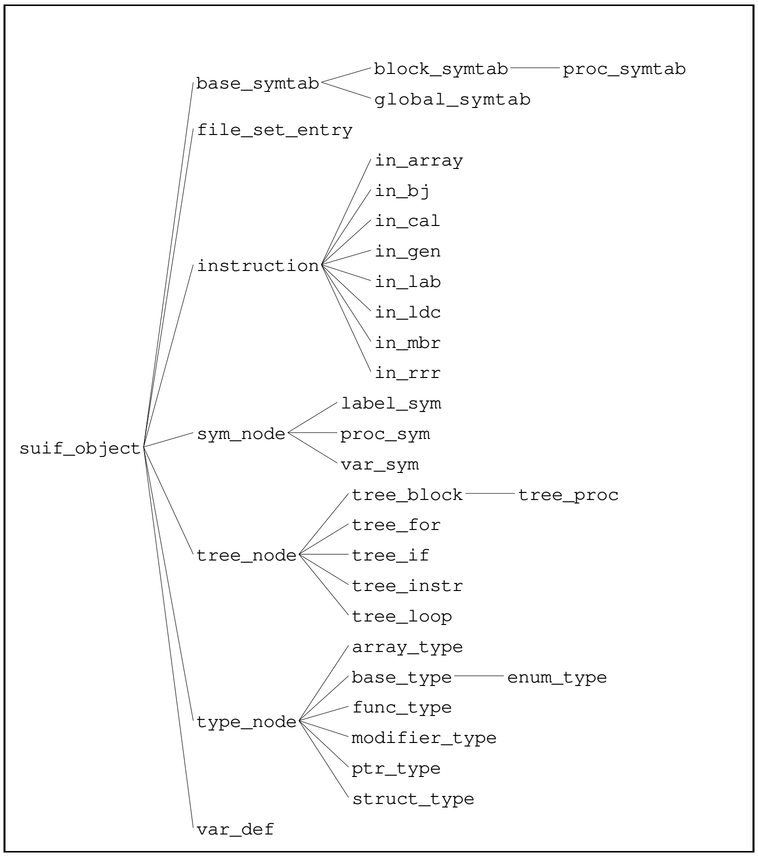

Figure 3.1: SUIF 1.0 class hierarchy

3.2.1 File Level

The SUIF IR is organized as hierarchical structure. At the root of the hierarchy is a *file set*, containing a list of the source files that represent the program. Each source file is associated with a *file set entry*, which contains the input and output streams of the file.

The file set contains a *global symbol table* for symbols that are visible across all the source files (e.g., procedures and `typedefs` in C). Every file set entry has its own *file symbol table* for symbols that are only visible within the source file (e.g., procedures and global variables with static linkage in C). Symbol tables are discussed in more detail in section 3.2.4.

3.2.2 Procedure Level and ASTs

The representation of procedure bodies contains both high-level and low-level structures, called *high-SUIF* and *low-SUIF*, respectively.

High-SUIF corresponds to an AST representation with tree nodes containing high-level control structures of the source program. Nodes of the AST are instances of classes derived from the abstract *tree node* class. For example, *block nodes* are used to model nested scopes, conditional structures may be represented by *if nodes*, and loops by *loop nodes* or *for nodes*. Leaf nodes of an AST are *instruction nodes*. These nodes act as simple wrappers, which contain a single instruction (refer to section 3.2.3). The root node of an AST has to be a *procedure node*. Procedure nodes are not allowed anywhere else in the AST. (This implies that SUIF 1.0 has no support for nested procedures.)

When the code is lowered to the low-SUIF form, these high-level constructs are translated to a sequential list of instructions. This representation corresponds to *quadruple code*, which is often used by traditional compilers [ASU86].

Because a typical SUIF pass needs to visit all nodes in an AST in order to perform operations, the SUIF 1.0 library provides a mechanism for it. It is possible to traverse the tree preorder or postorder with a user-defined *map function* that is automatically applied to every tree node during traversal.

Associated with every procedure is a *procedure symbol table* that stores symbols (e.g., formal parameters and local variables), that are within the scope of the procedure.

3.2.3 Instructions

SUIF instructions are attached at the leaf nodes of ASTs. Most SUIF instructions have the behavior of a typical RISC-style instruction set. More complex instructions for procedure calls (`cal`), multi-way branch (`mbr`), and array indexing (`array`) are also available. These instructions provide a higher level of abstraction.

SUIF 1.0 supports both *expression trees* and *flat lists* of instructions. In the flat list form, each SUIF instruction is atomic (i.e., each operand is either a symbol or a null operand), whereas instructions in expression trees can take other instructions as their operands. It is possible to convert expression trees to flat lists and vice versa.

The instructions use a quadruple format with one destination operand and two source operands. Operands either refer directly to a variable symbol (refer to section 3.2.6) or to another SUIF instruction. (The latter is used to implement expression trees.)

SUIF 1.0 uses special pseudo-instructions to mark the position of *labels* in the IR. Note that this means that labels are not attached to instructions, but rather stand on their own.

SUIF 1.0 provides a *generic instruction* class to support special-purpose extensions in SUIF. (For example, Machine SUIF [Smi96] uses this class to extend the SUIF instruction set.) Generic instructions contain an arbitrarily number of source operands and a user-defined name to identify them. They are not part of standard SUIF 1.0 and standard optimization passe ignore them.[2]

[2] More precisely, existing SUIF passes can usually be reused, but since they do not understand a generic instruction they must be conservative in any analysis that relates to this instruction.

3.2.4 Symbol Tables

Figure 3.2 gives an overview of the different kinds of symbol tables and shows how they are conceptually associated with the SUIF 1.0 hierarchy. Every entity on the left side of the diagram is associated with exactly one symbol table on the right side.

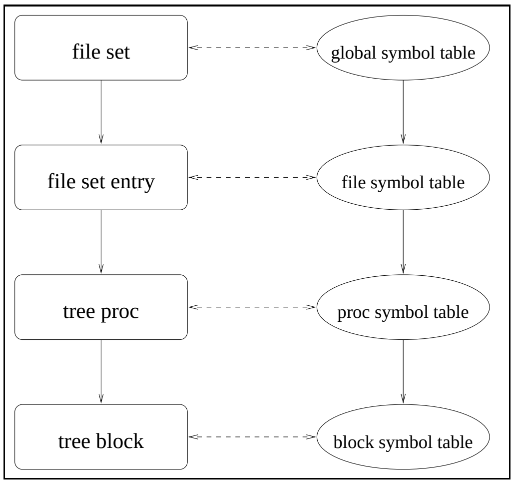

Figure 3.2: Symbol table hierarchy

A file set contains file set entries and has the global symbol table attached to it. In contrast, a file set entry does not contain procedures as one might expect. The procedures of a file are accessible through procedure symbols (refer to section 3.2.6), which are kept in the file symbol table that is associated with the file set entry. Tree blocks are nodes in the AST of a procedure (and hence are not directly accessible). Symbol tables contain three different kinds of entries:

- *types* (refer to section 3.2.5)
- *symbols* (refer to section 3.2.6)
- *variable definitions* (refer to section 3.2.7)

3.2.5 Types

SUIF types are used to describe the result-types of instructions and the types of variables. Types are stored in symbol tables. They are visible only in the scope of the symbol table they reside in. Each type knows its storage size (in number of bits).

SUIF 1.0 provides a set of classes to represent types. Objects of these classes are used as *type nodes*. *Type graphs* can be constructed with *type operators*. SUIF 1.0 provides operators for base types (integer, float, `void`, `enum`), arrays, pointers, function types (for SUIF procedures), `struct`s, and `union`s. Structured types contain references to other type nodes, whereas the leaves of a type graph are base types that contain no references. Furthermore, SUIF 1.0 uses *modifier types* to add attributes (e.g., `constant` and `volatile`) to other types.

SUIF 1.0 predefines a number of (numerical and pointer) types, which reside in the global symbol table.

3.2.6 Symbols

SUIF 1.0 symbols are subdivided into the following categories:

- *procedure symbols* contain a reference to their type and a procedure node that represents the procedure's body. Because nested procedures are not supported, a procedure symbol can be entered only into the global symbol table and file symbol tables.

- *label symbols* are used within SUIF code to specify targets of branch and jump instructions. (The position of a label in the SUIF code is indicated by a label instruction.) Label symbols may only be entered into procedure and block symbol tables.

- *variable symbols* represent data objects, i.e., local or global variables. (SUIF 1.0 does not distinguish constants at the symbol level.) They contain a reference to the type of the variable. Variable symbols may be used in any symbol table.

  Symbols for variables that are not allocated on the stack (e.g., global variables and `static` local variables in C) must have a separate variable definition (refer to section 3.2.7) in order to allocate storage for them. If the variable definition is missing, it is assumed that it is a declaration of an *external symbol*.

  Variable symbols have flags to indicate whether a variable is stored in a register, whether the symbol represent a formal parameter of the procedure, and whether it has its address taken (and hence could be *aliased*). The latter is especially useful for compiler optimizations that can apply transformation to non-aliased variables.

3.2.7 Variable Definitions

Variable definitions are used to allocate storage for variables that are not allocated on the stack. The symbol table that holds the variable definition is the place where the variable is actually defined. Usually the variable definition will reside in the same symbol table as the variable symbol itself. It is possible to specify the initial data for the allocated storage.

3.2.8 Annotations

Annotations are structures that contain arbitrary user-defined data. Every class derived from the SUIF 1.0 root base class can have annotations attached to it. New annotations can be declared for a SUIF pass and used to record whatever information is needed within that pass. *Registered annotations* can be written out to SUIF output files, providing a mechanism to pass user-defined data to another SUIF pass.

Annotations are implemented by the class `annote`. Each annotation contains its name and a data pointer. The data field is implemented with a `void*` pointer that has to be cast to the user-defined data structure.

3.3 The SUIF 1.0 Toolkit

The SUIF 1.0 compiler toolkit consists of

- a C and Fortran front end that generate SUIF files.

- various SUIF passes that implement program analyses and optimizations.

- back ends for several architectures.

- various compiler development tools to aid in the compiler development process, such as passes that generate textual, postscript or C output from a SUIF file. The textual output directly corresponds to the intermediate format of a SUIF file and the postscript output shows the corresponding AST. The C code output is useful for high-level debugging. The *checksuif library* is a debugging tool that checks the consistency of the IR of a SUIF file. The *builder library* [SCG94a], which sits on top of the SUIF library, provides a convenient, high-level interface to build SUIF code.

Figure 3.3 illustrates how the elements of the toolkit are combined into the SUIF 1.0 framework.

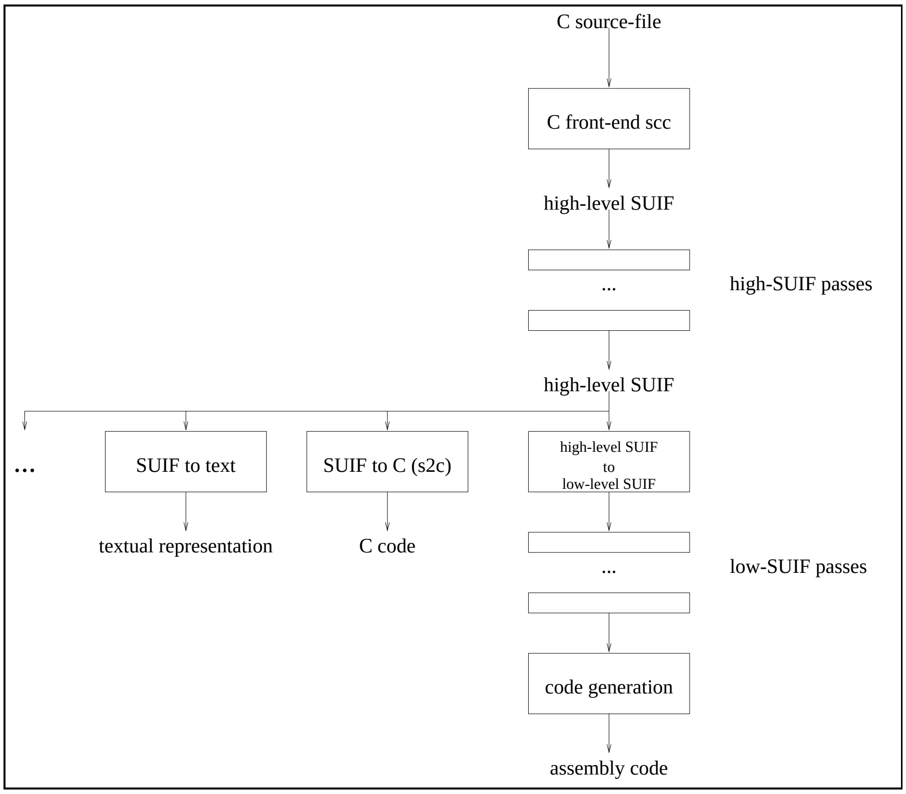

Figure 3.3: The SUIF 1.0 framework

3.4 Assessment of SUIF 1.0

The design of the SUIF 1.0 kernel is object-oriented and implemented in C++, but the implementation often makes use of C features that obscure the design and hamper extensibility. The implementation uses global variables, function pointers for iterators over SUIF structures, `define` preprocessor directives to mimic templates,[3] and global `enums` for runtime type identification.

The design of the kernel also reflects the fact that it was written to represent translated C and Fortran code in the first place. Nested procedures are not supported in SUIF 1.0. The SUIF type system

[3]When the SUIF kernel was implemented "C++ templates seem[ed] to be plagued with bugs in the GNU compiler."

supports `volatile`, incomplete function prototypes, and functions with a variable number of arguments as they are found in C.

Machine SUIF [Smi96] extends the SUIF 1.0 system with abstractions necessary for machine-dependent optimizations such as global instruction scheduling. One of the goals was to extend the SUIF 1.0 system without the need to duplicate functionality, by reusing the SUIF infrastructure whenever possible. Therefore Machine SUIF provides some valuable insights into the extensibility of the SUIF 1.0 system. Machine SUIF fits almost seamlessly into the SUIF 1.0 system, testifying that SUIF 1.0 has been designed with a certain degree of extensibility in mind. For example, the Machine SUIF library derives a new set of instructions from the generic instruction class `in_gen` (refer to section 3.2.3 and figure 3.1). Unfortunately, this is the only mechanism—besides annotations—that SUIF 1.0 provides if a client wishes to extend functionality. It is not possible to extend other parts of the SUIF IR (e.g., statements, symbols, or types) in a seamless way. Machine SUIF had to define its own wrapper methods in order to use the I/O functionality provided by SUIF 1.0. Furthermore, methods had to be modified and new methods had to be added in the SUIF `instruction` and `operand` class.

A SUIF 1.0 experience paper [OQL97] from Hewlett-Packard Labs lists a number of other shortcomings. For example, the paper points out that there is no easy way to save data structures that contain pointers as annotations.

Annotations provide a very flexible and powerful mechanism to extend SUIF 1.0, but they have limitations. Annotations do not provide type checking and have a rather cumbersome interface. This is a potential source for programming bugs that are very hard to find. Moreover, annotations do not fit in seamlessly because they have a different interface compared to the standard SUIF constructs. Finally, trying to use annotations to give a different meaning to (predefined) SUIF constructs often invalidates existing code. For example, assume an extension to SUIF 1.0 that models a C++ `try`-block. An intuitive approach is to use a block node with annotations attached to it. But this means that existing optimization passes would still see a block node and wrongly assume that they understand its semantic meaning (e.g., its control flow), probably resulting in the generation of wrong code.

To overcome the various shortcoming stated above, the Stanford Compiler Group decided to redesign SUIF 1.0. This new version, SUIF 2.0, will be discussed in chapter 4.

As stated before, the high-SUIF representation keeps high-level information for control flow constructs, array access, and retains scope information. These high-level informations are usually sufficient for imperative languages. Other language paradigms, for example object-oriented or functional programming languages, require additional (or completely different) high-level information for program analysis and optimization.

SUIF 2.0 does not extend the functionality of SUIF 1.0 towards non-imperative languages, but provides far better support for implementing those extensions on top of SUIF 2.0. OSUIF, an extension for object-oriented languages is discussed in chapter 5.

Chapter 4

The SUIF 2.0 Compiler System

To overcome several shortcomings of SUIF 1.0, the Stanford Compiler Group developed a new version of SUIF, called *SUIF 2.0* in this thesis. The basic principles and functionalities stay the same, but the old SUIF kernel (refer to section 3.2)has been completely redesigned. This means that SUIF 2.0 is not compatible with SUIF 1.0.

For a smooth migration, the SUIF 2.0 system provides a compatibility library that supports SUIF 1.0. Optimization passes of both versions can coexist. Conversion passes translate from the SUIF 2.0 to the SUIF 1.0 file format and vice versa, thus allowing to use most parts of the old SUIF toolkit in the new system.

4.1 Overview

The new design of the SUIF kernel introduces five layers of abstraction:[1]

1. The *sty* library is the lowest layer of SUIF 2.0. It defines generic data structures (e.g., lists, arrays, and hash tables), string classes, a lexicon class, a command line parser, classes that can hold arbitrary precise numerical values. Higher layers make extensive use of these classes. Relevant parts of the library are explained in more detail in section 4.2.

2. The *wallow* library provides architecture independent input/output routines. Data can be exchanged between two different architectures, even if they have different byte ordering and different sizes for various integral and floating point types. This library is only used internally by the zot library; a user of the system does not need to know anything about it.

3. The *zot* library defines the base class of all SUIF objects. This design makes it possible to provide a uniform low-level view of SUIF objects. Operations on SUIF objects that do not care about the actual type of the object (e.g., reading and writing SUIF objects to a file) can use this interface

[1] The author feels compelled to give a short explanation of the motivation that lead to the SUIF 2.0 library names—especially for non-native speakers [Wil97a]. *pyg* is pronounced like pig. Now, a pig—or rather pyg—*wallow*s in its *sty*; get it? *zot* has no meaning in the English language. It is a (recursively defined) abbreviation for "**z**ot **o**bject **t**ype." This rather awkward name has been chosen because the name for the *abstract* third layer of the SUIF 2.0 system does not want to raise any unwanted associations by using a word that has a defined semantical meaning (at least in the English language; the author is not sure if this holds for Vogon as well [Ada89])—simply because this association would probably turn out to be wrong or misleading. Certainly, everybody will agree that zot sounds a whole lot better than "thingy." The last library name, *suif*, turns out to stand for precisely what its name suggests, namely the topmost library of the SUIF 2.0 system—pretty disappointing, huh?

to handle SUIF objects in a uniform way. This library lays the groundwork for the extensibility of the system; it is explained in more detail in section 4.3.

4. The *pyg* library defines the abstract framework of the SUIF IR, for example, (abstract) base classes for types, symbol tables, instructions, statements, etc. This library is explained in more detail in section 4.4.

5. The *suif* library defines concrete SUIF IR constructs, for example, classes for integer types, arithmetic instructions, control flow statements, etc. This library is explained in more detail in section 4.5.

To use the existing functionality of SUIF, only the first (sty), fourth (pyg), and fifth (suif) layer have to be used. Every SUIF class contains a fixed set of methods that realize the interface of the extensibility mechanism. If an application does not need to extend SUIF (a common case) it can simply ignore these methods. Extending SUIF with, for example, a new instruction or data type, requires an additional understanding of the extensibility mechanism (provided by the zot library).

It is tempting to include a detailed description of the SUIF 2.0 libraries with example code in this thesis, because, as of this writing, no documentation is available. But because of time and space constraints provide instead a high-level description of the concepts. In the following, the actual C++ class name is given (in `typewriter` font) when a SUIF construct is explained, thus making it easier to find the corresponding source code.

4.2 The *sty* Library

This library implements a wide variety of conventional, linear data structures and provides a uniform interface to them. The abstract base class `ro_tos` (read-only totally ordered set) defines the uniform interface for read-only access. It is possible to access elements by index and ask for the number of elements (array-style) or to iterate over the set by getting the first/last element and asking for the next/previous one (list-style). `tos` is derived from `ro_tos` and extends the interface with methods that can modify the data structure (list-, array-, and stack-style). Concrete classes that are derived from `tos` implement

- arrays (`array_tos`)
- singly-linked list (`slist_tos`)
- doubly-linked list (`dlist_tos`)
- cached doubly-linked list (`cdlist_tos`)
- array-style doubly-linked list (`adlist_tos`)

The implementation of `adlist_tos` is such that if only list-style operations are performed, it is as efficient as a linked list, and if only array-style operations are performed it is as efficient as a simple array.

Other SUIF libraries make extensive use of these data structures. Because they have a uniform interface it is easy to substitute different implementation strategies. An iterator class (`tos_iter`) can be used to iterate over an `ro_tos`.

Arbitrary size integers are supported by the `i_integer` class. It has the same operators as a C++ `int`. An `i_integer` is internally represented as a `long` if it fits, otherwise a string is used. Strings are supported with the `string` and `lstring` class. A `lstring` is a subclass of `string` that is registered in a lexicon. These string classes have unique identification numbers, and hence string comparison can be performed efficiently by pointer comparison.

The abstract base class `ion` (input/output noter) can be used to create human-readable output. It is used, for example, to generate error/warning messages and output for the `print_full()` method defined in the zot class.

Error and warning reporting functionality are provided by global functions. They allow SUIF passes to optionally specify file and line number information and automatically put in line breaks to keep the output below 80 characters per line. Using this interface yields uniform output for SUIF passes and executables. Output text is generated with the `ion` interface and can contain SUIF-specific structures.

The class `therapist` provides an alternate, more flexible interface for error reporting. The test condition is passed as an argument to a `want()` or `need()` method along with an optional diagnostic message. These methods are used like assertions but can be specialized by subclassing from therapist. The `want()` method is used to indicate an assertion failure that is non-fatal and allows the program to proceed, whereas the `need()` method is used for assertion failures from which the program cannot recover. This behavior is useful for verification methods that check the state of an object because the verification need not to be aborted after the first assertion failure that is encountered.[2]

Finally, a powerful command line parser (`command_line_parser`) is available, which can handle all standard usages of boolean, integer, and string arguments.

The sty library is compiler-independent and designed for interfaces between modules. It makes extensive use of C++ templates. The question why SUIF 2.0 does not make use of the C++ *Standard Template Library* (STL) [Str97] comes naturally to mind. At design time, STL was not widely available. Furthermore, STL uses exceptions, but many C++ compilers have trouble supporting them. There are other—more subjective—reasons. For example, STL makes extensive use of standard operator overloading, whereas SUIF tends to avoid this language feature [Wil97c].[3]

4.3 The *zot* Library

This library is used in the SUIF system to model a flexible extensibility scheme to put new kinds of objects into the preexisting hierarchy of SUIF constructs. For example, this scheme allows one to add new kinds of symbols, types, symbol tables (e.g., to model classes for object-oriented programming languages), statements (e.g., to model exception handling or concurrency constructs), etc.

The abstract base class `zot` provides the interface that gives a single, logical view to all SUIF constructs that inherit from this base class. Because every object supports this interface, each object can be seen as a black box (even if the particular semantics of its components is not known).

The logical view of a `zot` object consists of (1) a string tag for introspection, and (2) a finite sequence of *bricks*.

The string tag tells the position of the object in the SUIF hierarchy. For example, the SUIF instruction "add" has the following tag:

`"azot/szot/instruction/binary_arithmetic_instruction/add"`.

Note that the SUIF hierarchy does not necessarily reflect the C++ class hierarchy of the implementation.[4] A tag tells how to interpret a specific object. Even if a pass does not understand all of the tag, it still might understand a certain prefix.

A brick, implemented in class `brick`, is one of the following:

[2] Surprisingly often, not the first assertion failure, but a subsequent one, indicates the actual problem that caused the inconsistent object state.

[3] The usefulness of overloading standard operators is questionable (e.g., extensive and unintuitive usage can degrade program readability).

[4] For example, this is the case for all concrete SUIF arithmetic instructions, which are represented by either `binary_arithmetic_instruction` or `unary_arithmetic_instruction` in the C++ class hierarchy.

- A reference to another object. IR objects form a directed graph, i.e., each object has some number of references to other objects. For example, an `add` instruction refers to the result type, the two source operands, and the destination operand.
- An *ownership link* to an owned object. Each SUIF object has exactly one parent, which can be accessed with the `parent()` method. This structure yields an "ownership tree" of the IR objects that is conceptually embedded into the directed graph structure of the object references. For example, an instruction in an expression tree is owned by another instruction and a statement is owned by its enclosing statement. Having a distinguished owner for every object has several useful applications, for instance, iterating over a node, reading/writing of a node, or cloning a node and everything that it owns. Note that leaf objects in the ownership tree do not have an ownership link brick. A variable symbol, for example, never has an ownership link brick among its components.
- An integer, a string, or a sequence of bits.

Each object's data can be represented with a sequence of bricks.

In summary, tags represent the semantic information and bricks store the raw data of an object. Figure 4.1 shows the zot interface.

```
class zot {
private:
  zot *_parent;
  ...
public:
  virtual lstring tag(void) const = 0;

  virtual s_count_t num_components(void) const = 0;
  virtual brick component(s_count_t component_num) const = 0;
  virtual void set_component(s_count_t component_num, brick the_brick) = 0;

  virtual void verify_invariants(therapist *doc) const;
  virtual void apply_zot_visitor(zot_visitor *the_visitor);
  ...
};
```

Figure 4.1: The zot interface

File I/O at the zot level is abstracted in terms of tags and bricks. Usually a class X registers its tag and an instance of its *builder class* X`_builder` with the zot library. When X's tag is found during reading in from a file, the correct in-memory representation of the object can be constructed by calling the associated *builder method*, namely X`_builder::default_builder()`. If X's tag has not been registered (i.e., no specific builder method is available) an object of class `generic_`C is created, where C is the "closest" subclass to X in the class hierarchy that has a registered builder method. The builder method calls a constructor that takes a list of bricks as a parameter. This functionality is realized with the `zot_builder` class.

Every class X that is (transitively) derived from `nfzot` (see below) is required to implement a `generic_`X class as well. This makes it possible to represent objects of classes even when the class is unknown to the system (and hence no builder class for them exists). For example, a user might want to build a new library on top of SUIF and subclass Y from a SUIF class X. When class Y is read in, but the system cannot find the specific library of the user, SUIF needs a mechanism to represent the addition information (i.e., bricks) of class Y that are not in class X. The class `generic_`X provides this functionality.

When reading in from a file we don't always want to read in the whole file and create the whole in-memory representation of it, but would rather like to do I/O on subtrees of the ownership tree of zot objects. For example, a pass that does intra-procedural analysis needs only one procedure body at a time. Thus, it is not necessary to have all procedure bodies in memory simultaneously. To handle such cases, a flexible scheme is provided that groups sub-trees of zot objects into I/O units. A group of I/O units is represented with the `zio` class. The user can read, write and deallocate zot objects in units of zios. For example, a file set block contains a list of procedure bodies. By grouping every procedure body subtree into a separate zio we can read in one procedure at a time. The zio functionality is used for several different memory models that the suifpasses library provides (refer to section 4.6).

It can happen that a zot object in memory contains a reference to another object that is currently not in memory. For example, a interprocedural analysis might store information that points from a zot object in one procedure to an object in another procedure. If a subsequent intra-procedural pass reads in one procedure at a time it is very likely that references are out-of-memory. These cases are handled by the library with a "forwarder" mechanism. A `forwarder` is a special zot class, serving as a proxy for something that is not in memory. If a class is read in from a file that contains a reference to an object that is out-of-memory and in a different zio, a reference to a forwarder object is created.[5] The forwarder allows the user to transparently see the object the forwarder is referring to.

The non-forwarder classes are distinguished from the forwarder class by inheriting from the abstract class `nfzot` (non-forwarder zot). All SUIF constructs from the pyg and suif library inherit from this class. Figure 4.2 shows a simplified class hierarchy of the zot library. Note that no `generic_zot` class exists because the only legal subclasses of zot are forwarder and nfzot. Hence, every zot object can be represented as either a forwarder or a nfzot. The `generic_forwarder` class has not been implemented because there seems to be no application that requires to subclass from `forwarder` [Wil97b]. Typically, a user will extend azot or a class that inherits from azot.

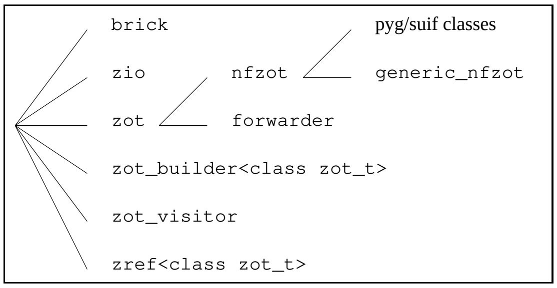

Figure 4.2: Simplified zot library class hierarchy

The `zref` class is a more convenient wrapper for an object reference that can either be a forwarder or a "normal" zot object. A zref always return the forwarded zot object (going along the chain of forwarders if necessary) or nil if the forwarded object is not in memory.

The zot level also introduces the interface for the visitor pattern [GHJV94] (see figure 4.1). The zot visitor is implemented with the class `zot_visitor`. A visitor class contains methods to visit all SUIF IR constructs that the library defines. The default implementation is to do nothing. A visitor can be customized by subclassing from it. Every library L that builds on top of zot extends the visitor

[5]References to objects that are out-of-memory and in the same zio are first initialized to nil and backpatched as soon as the object is in memory.

framework by defining `apply_`L`_visitor()` methods and the L`_visitor` class. The default behavior for a visitor method that implements a new IR construct is to call the method for the closest underlying IR construct.

Furthermore, an interface to clone zot objects is provided. Every class X that inherits from the zot hierarchy must implement the method `clone_tree_`X`()`, which performs a partial deep clone, and the method `clone_local_`X`()`, which performs a shallow clone. To get a complete deep clone, the method `clone()`, which is implemented by the zot class, is called. A deep clone replicates all SUIF objects that it owns, but no references to objects that do not live in the subtree that is about to be cloned. The clone method first uses the `clone_tree_*()` methods to get a deep clone with unresolved references and then backpatches the unresolved references. The last step of the cloning process calls `close_clone()` for every object of the cloned tree. This method allows the implementation of class-specific cloning semantics.[6]

Finally, the library provides interfaces for useful functionality, which should be implemented by every class that inherits form the zot hierarchy, namely, printing and verification. A new class will usually redefine the `print_full()` method to provide a specific printout of its state. (Printing can be seen as a debugging-friendly, high-level representation of bricks.) Because classes are not forced to redefine the print method, it is possible that a print method, after pretty-printing the state of its own class, has bricks left it knows nothing about (from the chain of subclasses that did not redefine the print method). If this happens, the remaining bricks (i.e., the state of the subclasses) are printed with the `print_brick()` global function. The print methods use the ion class to generate string output and the `zot_print_helper` class. The helper class manages indentation and prints a zot (by invoking `print_full()` on the zot) or just a reference to a zot (usually represented by a number).

Verification of the state of an object can be done by invoking the `verify_invariants()` method. Only the local state of the object is verified (e.g., if references to other objects have the correct type); the verify method is not intended to walk a tree and verify everything in the tree. To verify the whole (sub-)tree, the user must iterate over it. This means that if the verified object contains references to other zot objects these must be verified in turn. A similar situation to the print method arises when this verify method encounters state in the form of bricks. In this case, the verify method can look for bricks that represent a reference to another zot object and invoke the verify method on them. The verify method uses the `therapist` class (see section 4.2) to issue diagnosis messages.

The zot framework is nothing IR-specific; the same design could be used for other graph structures. The zot interface is orthogonal to the high-level interface of a SUIF construct. A typical user will only use the high-level interface and ignore the low-level zot interface.

4.4 The *pyg* Library

The pyg library is the first compiler-specific library of the system. The (abstract) constructs of the SUIF IR are implemented in this library. For example, the pyg library defines an abstract base class `instruction` to model the concept of an instruction. Classes that are subclasses of zot, but not leafs in the inheritance hierarchy are called *intermediate classes*. Intermediate classes have intermediate interfaces. For example, `instruction` defines the interface for an arbitrary number of source and destination operands, and result types. All concrete instructions have these components. Every subclass gives a more concrete interface. For example, `binary_arithmetic_instruction` restricts the number of source operand to two. Usually intermediate classes are defined in pyg, and leaf classes are defined in the suif library. The pyg class hierarchy (except for the generic classes) is shown in figure 4.3.

The `azot` (annotatable zot) class inherits from zot. All SUIF IR constructs inherit from azot. An *annotation* consist of (1) a string to identify it, and (2) data. The abstract base class `annote` implements

[6]For example, whenever a `label_location_instruction` is cloned, the corresponding code label symbol needs to be explicitly cloned as well if the symbol table where the label lives has not been cloned.

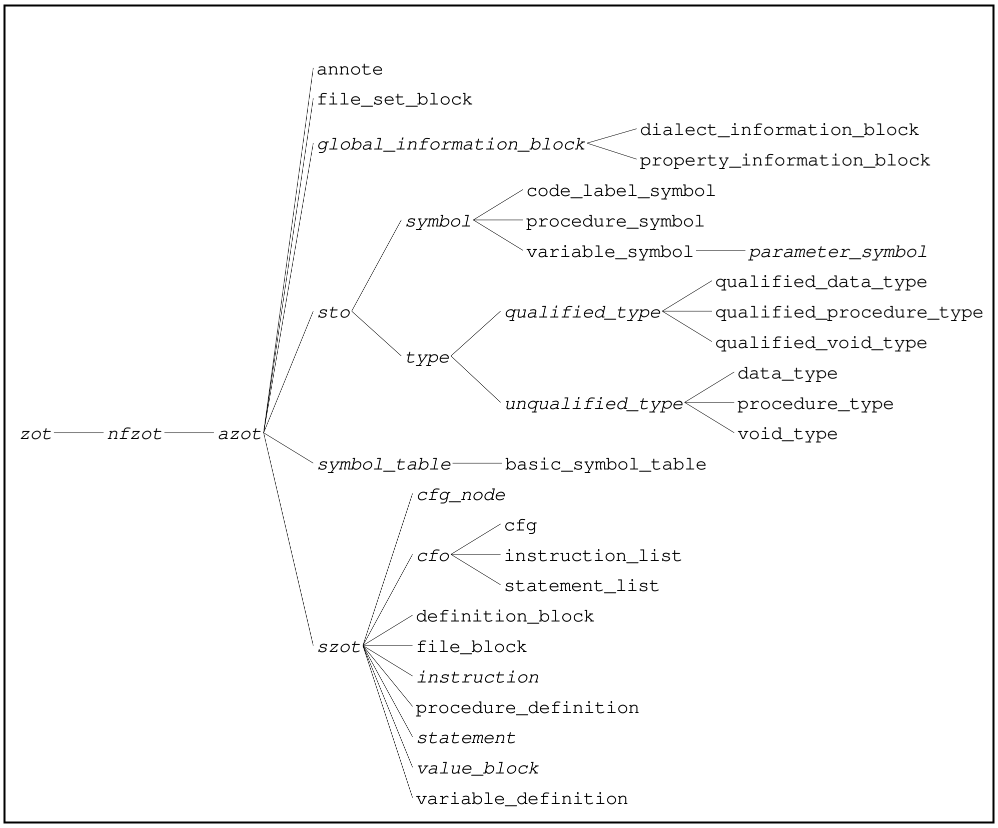

Figure 4.3: The pyg library class hierarchy (generic classes omitted)

annotations. The `generic_annote` class keeps the annotation data as bricks. If the user wishes a high-level view of the annotation data he/she can derive a custom subclass from annote. The annotation name is the last part of the tag.[7] The annote class itself is a subclass of azot, which makes it possible to annotate annotations. An arbitrary number of annotations can be attached to an azot.

The `sto` (symbol table object) class defines objects that can be stored in symbol tables. The sto interface allows symbol tables to treat their contents uniformly. The pyg library has two kinds of symbol table objects: (1) symbols, and (2) types.

A `type` can be one of the following:

- A data type (`data_type`), which gives structural and semantical interpretation to otherwise "raw" data.
- A function type, which is expressed with the class `procedure_type`. A SUIF procedure type can have multiple return types. (A SUIF procedure can return multiple return values.)
- The void type (`void_type`).

[7]This subtle design decision makes it possible that annotations that are instances of the `generic_annote` class will be "magically" restored to instances of this class when read back from a file.

Types can be qualified or unqualified. A *qualified type* is a unqualified type with additional *qualifications* about storage (not data or values).[8] For example, expressions always return unqualified types because they give a value; there is no storage location involved. But a pointer type refers to a qualified type because we want to be able to represent information about the storage that the pointer points to.[9] SUIF supports qualifications such as `volatile` and `constant`. The user is allowed to add other qualifications, but they should follow the above conventions. For example, there could be qualifications `local memory` and `remote memory` to distinguish memory with different physical access characteristics. This information could be used to optimize code based on the knowledge that some memory accesses are cheaper than others.[10] The type hierarchy for qualified types mirrors the type hierarchy for unqualified types. The class `qualified_`X is implemented as a wrapper that refers to the underlying unqualified class X.

All `symbols` can be—unlike certain types—conveniently identified by their name.[11] Every symbol can be asked if its address has been taken. This is useful for alias analysis. There are four kinds of symbols:

- Variable symbols (`variable_symbol`) represent data storage. They are typed and hence must contain a reference to a SUIF type. Variable symbols that represent non-stack data have a reference to a variable definition (see below). Typically a variable symbol is used as an instruction operand.
- Procedure symbols (`procedure_symbol`) have a reference to a qualified procedure type and if the procedure is not external a reference to a procedure definition (see below).
- Code label symbols (`code_label_symbol`) denote a specific program point in the IR. Usually a label is used as a jump target for control transfer instructions (e.g., `jump_instruction`).
- The formal arguments of a procedure definition are represented by the `parameter_symbol` class. It is derived from a variable symbol and contains an additional reference to its procedure definition.

A `symbol_table` is a container for symbol table objects. Its interface defines methods to retrieve, remove, and insert sto objects.

The abstract base class `cfo` (*control flow objects*) represents a chunk of executable code, such as a procedure body or the `then`-part of an `if` instruction. There are three different choices to represent code, namely,

- as a control flow graph (`cfg`). A SUIF CFG is a directed graph. Vertices are represented with the `cfg_node` class. Outgoing edges of a node are (conceptually) modeled as references to other cfg nodes. Every cfg node contains a cfo object. Every CFG must have a `start node` and can have an `exit node`. The exit node is used to model nested CFGs, but it can be used for flat CFGs (i.e., the nodes of the CFG contain a list of instructions) as well. (An explicit exit node often simplifies CFG algorithms.)
- as a list of instructions (`instruction_list`). SUIF instructions (`instruction`) are used for low-level, explicit control flow. They are atomic in a sense that they cannot contain other cfo objects.
- as a list of statements (`statement_list`). Statements (`statement`) are used for high-level structuring of the control flow. They are similar to ASTs (e.g., control flow is implicit). SUIF statements are very similar to statements as they can be found in high-level programming languages. For

[8] This scheme has been adapted from the ANSI C standard.

[9] One might wonder why qualifications are not attached to symbols. Pointer variables are one of the reasons, because sometimes storage is not directly associated with a variable, but can be referenced indirectly through a pointer.

[10] Note that this information is of interest for all storage accesses, direct uses of variables as well as indirect references through pointers. That is why it is appropriate to use these qualifications on types.

[11] Names are not required to be unique in a symbol table. Lookup by name can retrieve multiple symbols.

example, a `while` statement has an operand that represents the condition and a cfo object that represents the loop body.

The representations mentioned above can be freely mixed. For example, it is legal to have a cfg whose nodes contain sub-`cfg`s or statement lists. SUIF provides conversion passes that transform between any of the cfo representations.

The class `constant` is used to represent a compile-time constant. For instance, it is used as the operand of the `load_constant_instruction` and for the `case`-labels of the `multi_way_branch_instruction`. A constant can be an integer value, a bit-block, or a string. String constants are used to encode floating-point values. (This is machine-independent and allows the encoding of "NaN.") Furthermore, it can be used for application-specific constants. For example, a front end might wish to represent a nil reference with a string constant that is lowered to a concrete representation by a subsequent machine-specific pass.

Operands in SUIF are represented with the abstract base class `operand`. Source and destination operands of instructions are distinguished with instances of the classes `source_op` and `destination_op`, respectively. An operand holds one of the following: (1) a variable symbol or (2) a reference to an instruction and the number of a source or destination operand that is part of this instruction. More precisely, if it is a source operand the number denotes the destination operand of the instruction; if it is a destination operand the number denotes the number of the source operand of the instruction.

This last features makes it possible to build *expression trees*. They do not require temporary variables and are often straightforward to build. For example, it is possible to build the expression $x + 42$ without introducing temporary variables when an expression tree is used. The `add` instruction expects two source operands. The first operand holds a variable symbol (which represents x), the second operand holds a `load_constant_instruction` and the number zero, which denotes the first (and only) destination operand of the load constant instruction (which holds the value 42). This tree, which represents a (sub-)expression, could be combined with another expression tree or assigned to an l-value.

Note that constants and operands are not strictly part of the IR because they are not derived from zot and hence not explicitly written out to a file. Constants are written out as bricks, and operands are wrappers that can refer to different IR constructs.

The `variable_definition` class is used to represent statically allocated variables (i.e., variables whose addresses are known at compile time). The storage represented by a variable definition can be explicitly initialized with a `value_block`. A variable symbol can have at most one variable definition.

Similarly, a procedure is defined with a `procedure_definition`. It contains a symbol table (which represents the procedure scope), the procedure body (which is a cfo object), a list of formal parameters (which are represented with the `parameter_symbol` class), and a `definition_block` that holds statically allocated locals (e.g., `static auto` variables in C/C++).

At the highest level of the IR hierarchy resides the `file_set_block`. It contains

- a list of file blocks (`file_block`). A file block represents a file scope (i.e., a single source file). Note that SUIF does not support namespaces, which can be orthogonal to file scopes. Furthermore, a file set block has a symbol table and a definition block, which holds a list of variable and procedure definitions. If the source language has a flat namespace (i.e., a file scope does not introduce a new namespace), the file set block's symbol table is empty. If the source language does not support multiple source files (e.g., original Pascal and Simula-67) the file set block contains a single file block.

- a file set symbol table (for objects that are visible for all file scopes).

- an external symbol table (for objects with external linkage).

- information blocks (see below).

Information that is invariant for the entire file set is kept in an *information block* system. Any of the following blocks, which are derived from `global_information_block`, can be part of a file set block:

- The `dialect_information_block` contains a list of *dialect properties* (see section 4.7) for a particular file set. This information is used, for example, by the suifpasses library (refer to section 4.6) to ensure that passes are (1) run in the right order and (2) make valid assumptions about the input that they receive from a previous pass.
- The `property_information_block` keeps a set of properties, which state information about SUIF IR constructs. For example, for an instruction the property "has no side effects" and for a binary arithmetic instruction the property "is associative" could be given. A property is denoted by a string-tag and contains an annotation to keep additional information. Properties are useful for optimization passes that do not know the precise semantics of a (user-defined) IR construct.

To summarize, the pyg library provides the conceptual framework of the IR. The suif library, discussed in the next section, implements a concrete IR on top of this conceptual framework.[12]

4.5 The *suif* Library

The suif library, which is the top-level library of the SUIF system, contains all the IR constructs that are needed to represent imperative languages, such as C and Fortran. To support these languages, the suif library extends the pyg library with (concrete) types, instructions, statements, and control flow nodes. Furthermore, subclasses of `value_block` and additional information blocks are defined.

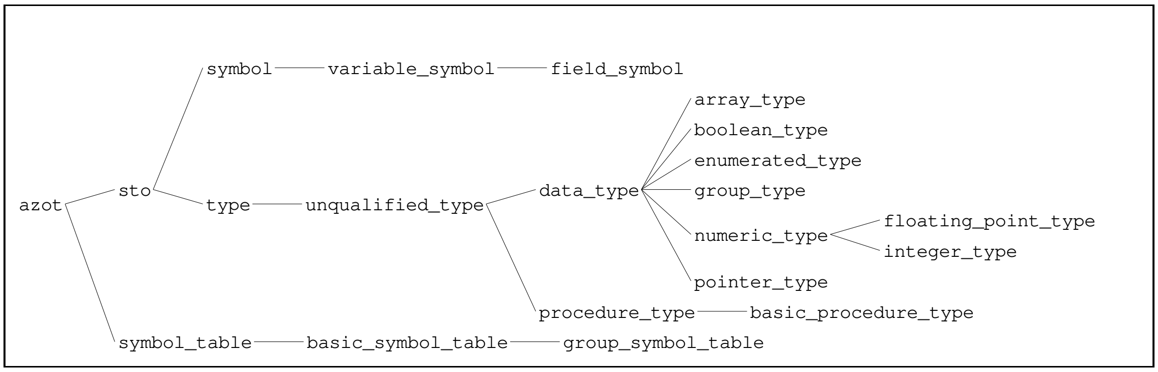

Figure 4.4: SUIF 2.0 class hierarchy (type-related classes)

Figure 4.4 shows the unqualified data types implemented by the library. This figure omits the class hierarchy for qualified types because it is identical to the unqualified type hierarchy. (See section 4.4 and figure 4.3 for a more detailed explanation.) Most types are self-explanatory. Integer and floating-point types can be defined with arbitrary precision and alignment restrictions at the bit level.

A `group_type` corresponds to a C-style `struct` or `union`[13]. A boolean flag indicates if the type is *incomplete*, i.e., not all fields of the group type have been specified. Standard SUIF requires an incomplete

[12] This distinction is not always intuitive and somewhat in the eye of the beholder.
[13] Fields in a group type can overlap arbitrarily.

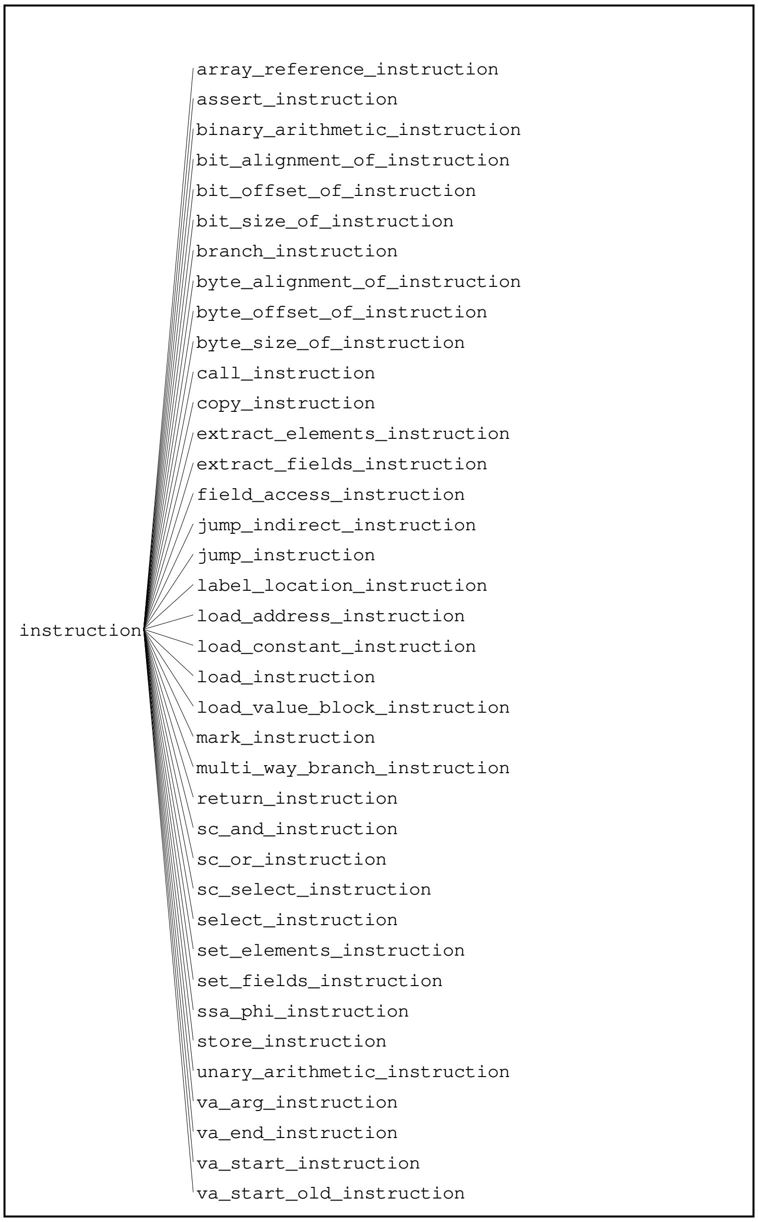

Figure 4.5: SUIF 2.0 instructions class hierarchy

group type to be empty (i.e., it does not contain any fields).[14] Incomplete group types in Standard SUIF correspond to *forward declarations* in C++. The *SUIF linker*, which takes an arbitrary number of SUIF files as input, resolves incomplete types by replacing them with the actual type. A group type contains a `group_symbol_table`. This symbol table subclasses from `basic_procedure_table` and can hold only `field_symbols`. Field symbols are used to represent fields in a group type. In addition to a variable symbol they contain a field offset.

The `basic_procedure_type` class, which is used to repesent the type of a SUIF procedure, implements a procedure type that is similar to a C function type in the sense that it can have a variable number of arguments and need not have its argument types specified.[15] Unlike a C function, it can have multiple return values. All types have an alignment restriction (specified in multiples of bits).

It is not required to specify a type completely. For example, one can leave the alignment restriction for a new type unspecified. In this case, a subsequent (target-specific) pass will fill in this information before code is generated. Similarly, the upper and lower bounds for an array type[16] and the offset for a field in a group type can be omitted.

SUIF distinguishes between structural equivalence and name equivalence of types [ASU86]. Name equivalence is used for group types[17] and enumerations (`enumerated_type`). These types have names (like symbols). All other types use structural equivalence and hence are unnamed. Every type X that uses structural equivalence defines a global function `get_`X`()`, which checks whether the type has been already entered into the global symbol table. If so, the existing type is returned; otherwise, a new type is constructed.

The suif library defines concrete instructions (see figure 4.5), which take any number of source operands, constants, etc. as input and have zero or more destination operands[18] for output. Thus, an instruction can be seen as a function with multiple return values. We give here a short explanation of the instructions whose semantics might not be obvious:

- The `label_location_instruction` defines a label. A `branch_instruction/jump_instruction` takes such a label as its branch/jump target. Every label instruction must have a unique `code-_label_symbol`.

- The `copy_instruction` is used to assign a different source operand to one or more destination operands. It must not be confused with the `store_instruction`, which stores a source operand into a storage location (denoted by a source operand[19]).

- The `field_access_instruction` takes the base address of a variable symbol, which represents a group type, and a field symbol. The address of the field is returned. To store a value into a field, the store instruction is used subsequently; to access the field value, the `load_address_instruction` is used subsequently. The `select_fields_instruction` and `set_fields_instruction` can be used to conveniently access the fields in a group type without using addresses.

  Instructions for arrays are almost analogous to group types, with the only difference that an index must be given instead of a field symbol.

[14] Note that on the other hand an empty group type does not imply that the type is incomplete (as opposed to SUIF 1.0).

[15] Such a type corresponds to a function prototype with empty parameter list in K&R C.

[16] This does not mean that SUIF supports *dynamic arrays* (like Ada and Java) or *flexible arrays* (like APL and CLU) [GJ87]. A subsequent pass must specify at least the lower bound before code can be generated for an array.

[17] Note that SUIF's notion of type equivalence for group types resembles Pascal `records`, but differs from C `structs`.

[18] The Standard SUIF dialects restrict the number of destination operands to zero or one.

[19] Note that a source operand is used because the store instruction does not return a value. The actual store is just a "side effect" of the instruction.

- The `ssa_phi_instruction` represents the SSA ϕ operator [CFRW91]. This instruction is legal only in code that uses the control flow graph representation. (SUIF will offer passes that transform code from non-SSA form to SSA form and vice versa.)
- The `sc_*` instructions are used for short-cut boolean operators. (Standard boolean operators are defined as `binary_arithmetic_instructions`.)
- The `mark_instruction` does not perform any operation. It can be used to attach annotations (e.g., for line number information).

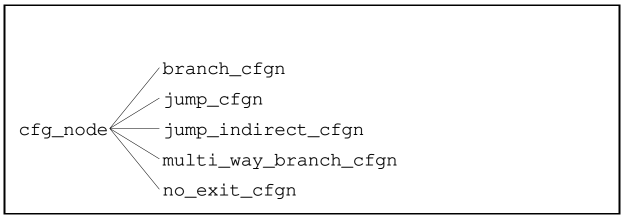

Figure 4.6: SUIF 2.0 control flow graph nodes class hierarchy

In the CFG representation, SUIF distinguishes between the following types of control flow graph nodes, all of which are subclasses of `cfg_node` (see figure 4.6):

- A `branch_cfgn` is used for conditional jumps (i.e., nodes with two outgoing edges).
- A `jump_cfgn` is used for unconditional jumps (i.e., nodes with exactly one outgoing edge). This node is also used for control flow that "falls through" to another node.[20]
- A `jump_indirect_cfgn` is used for an indirect jump to a code label symbol. (A `load_address-_instruction` can be used to get the address of a code label symbol.)
- A `no_exit_cfgn` has no out edges. Typically, this block transfers control flow back to a caller (e.g., with a `return_instruction`) or marks the end of the program.
- A `multi_way_branch_cfgn` corresponds to a `switch` statement in C.

SUIF statements (see figure 4.7) are used in SUIF to express control flow at a high level, as opposed to the low-level representation that instructions and CFGs offer. Statements are used only to structure the control flow; instructions must be used to perform computations. The `eval_statement`, which holds a sequence of instructions, is used to wrap instructions into a statement.[21] Conceptually, all expressions are evaluated first before any value is assigned.

Concrete subclasses of `value_block` (see figure 4.8) are used to initialize symbols with constants (e.g., a numerical value), or expressions (e.g., a numerical computation that uses the value or address of another symbol). A `multi_value_block` wraps a sequence of arbitrary value blocks. It is used to initialize group and array types. The `undefined_value_block` explicitly specifies that a symbol is uninitialized, but this is not mandatory.[22]

[20] Often a separate fall-through node is used to distinguish a basic block that ends with a non-transfer instruction from a block that ends with an unconditional jump, but this is not strictly necessary if the original code sequence need not be recovered.

[21] Usually a single expression tree per eval statement is used.

[22] This value block is not strictly needed for symbols with scalar types, but is necessary for partial initialization of symbols with array or group types.

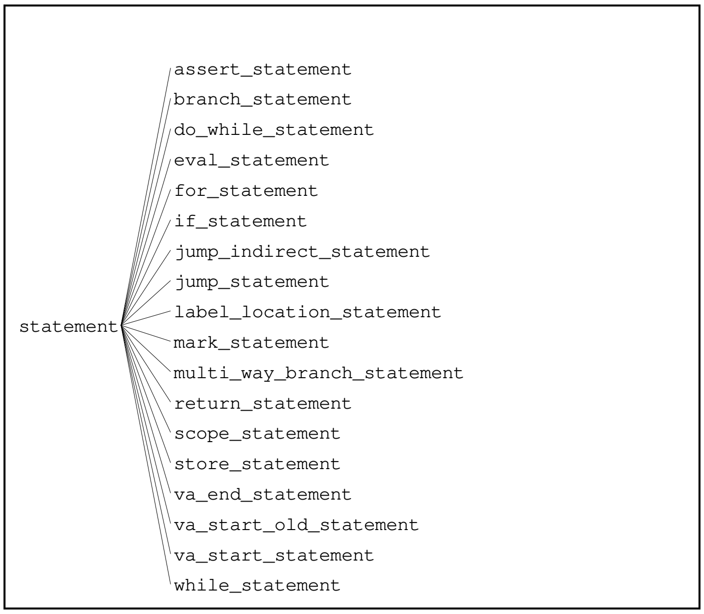

Figure 4.7: SUIF 2.0 statements class hierarchy

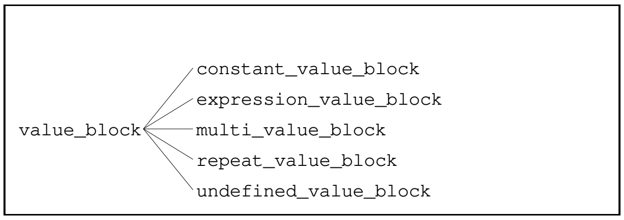

Figure 4.8: SUIF 2.0 value blocks class hierarchy

Finally, the library defines two additional information blocks:

- The `c_information_block` contains predefined C data types, which are useful for calls to the C library (e.g., `printf()`), or to external C code.
- The `target_machine_block` contains machine specific information (e.g., big or little endian and alignment restrictions).

It should be noted that the discussed libraries provide only basic functionality. SUIF's philosophy is it to keep the interface and the implementation of the five core libraries as simple as possible and to provide a more user-friendly interface with auxiliary utility libraries on top of the core libraries.

Last but not least—to wrap up the SUIF libraries—figure 4.9 gives an (incomplete) example of an IR ownership graph.

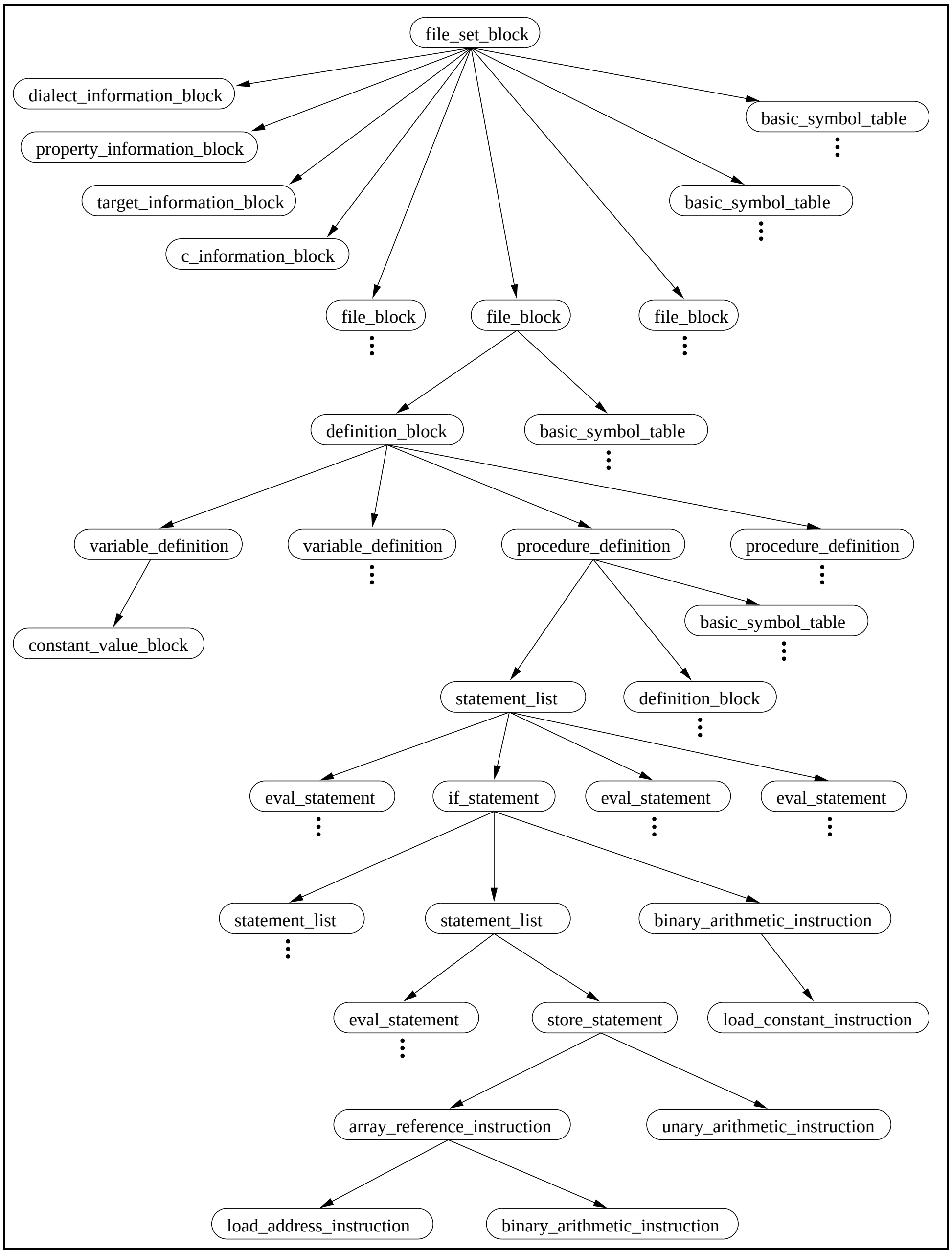

Figure 4.9: SUIF 2.0 ownership graph example [Wil97c]

4.6 The *suifpasses* Library

SUIF passes are usually independent units, which operate on or create SUIF code. They can be composed in memory or through files. A generic driver offers both behaviors for all passes that use the suifpasses library.

The suifpasses library distinguishes between the following types of passes (which are derived from the abstract base class `suif_pass`):

- A front-end pass (`suif_front_end_pass`) is used to represent passes that produce SUIF code as output but do not take SUIF code as input. (The `j2s` compiler is implemented as a front-end pass.)
- A SUIF-to-SUIF pass (`suif_to_suif_pass`) is used to represent SUIF-to-SUIF transformations of the SUIF code. This is typically an optimization pass. Several standard memory models are supported. For example, the "one definition at a time" model is used by passes that only want to see one procedure or variable definition at a time (such as an intra-procedural optimization pass). On the other extreme, the "all in memory" model reads in the complete SUIF file before the pass is run.
- A back-end pass (`suif_back_end_pass`) takes SUIF as input but does not produce any SUIF code as output. For instance, a pass that prints a textual representation of the SUIF code would be implemented as a back-end pass.

Each pass is registered by name and specifies its dialects (see section 4.7). The transformations on the SUIF code that are made by the pass and the properties of the SUIF code that the pass expects are described in terms of these dialects. The core_dialect library has a collection of useful standard dialects, whose properties can be checked with a corresponding checker. A checker performs static checks by applying the rules imposed on the SUIF code by the corresponding dialect. Each pass specifies the dialect properties that it

- expects from the input (as prerequisites).
- preserves (if already present).
- destroyes (by the transformations that the pass does).
- generates (by the transformations that the pass does).

This framework makes it easy to describe the effects that a pass has on the SUIF code. Furthermore, it can be used to verify that passes are run in correct order, i.e., that they make correct assumptions about the input they get from a previous pass.

To summarize, the suifpasses library organizes algorithms in an object-oriented way that allows modularity, extensibility, and reuse of passes.

4.7 The *core_dialect* Library

SUIF 2.0 allows users to specify certain (semantic) properties that a specific SUIF IR fulfills. These properties are called *dialects* in SUIF terminology. A dialect is expressed either as an atomic constraint that is imposed upon an IR construct or as the intersection of other dialects.

For example, the "Standard SUIF" dialect is defined as the intersection of several other dialects. Among these dialects are

"Standard Completeness" This dialect states which IR components must be complete to represent a meaningful IR tree. Incomplete components contain, for example, `NULL` pointers, empty strings, or empty operands. For instance, the load address instruction must have a valid reference to a symbol and a result type ("Standard Load Address Instruction Completeness").

"Standard Typing" This dialect specifies the type system, i.e., the constraints that are imposed on the SUIF types of IR constructs. The type system allows incomplete types, which essentially means that complete, valid types can be "chopped off" by `NULL` references.

For example, the return type of a load address instruction must be a `NULL` reference or an unqualified pointer type. The referenced type by the pointer type is constrained by the symbol (if the symbol is specified, i.e., not a `NULL` reference). The complete description is given in "Standard Load Address Instruction Typing."

"Standard Relationships" These dialects define local constraints for an IR construct. For example, the referenced symbol of a load address instruction cannot be a group or register symbol (see "Good Load Addresses").

All of the above dialects are composed of atomic constraints. These atomic constraints are (informally) defined for each IR construct.

A SUIF file specifies a list of dialects in its dialect information block (see section 4.4). A certain dialect can be enforced by registering a *checker* for it. Checkers can be run to assure that a certain SUIF file indeed fulfills the constraints of the dialects it claims to have.

4.8 Assessment of SUIF 2.0

SUIF 2.0 has several benefits over SUIF 1.0:

- It is more extensible and highly orthogonal. The most important part of the new design is the zot library, which models an extensible class hierarchy.
- It uses introspection with string-tags. Because they encode the complete SUIF hierarchy of the construct, string-tags convey more semantic information than the SUIF 1.0 implementation based on `enmums`.
- It supports nested procedures. (A nested procedure is kept in the `definition_block` of the enclosing procedure.)
- It is implemented in a more object-oriented programming style. (Section 3.4 describes where SUIF 1.0 tends towards a rather procedural programming style.[23])
- It allows a more object-oriented style of programming. SUIF passes are now encapsulated into classes, making it easier to write variations on existing passes. Visitors can be used to iterate over SUIF structures (e.g., the code of a procedure).
- It provides a new mechanism to manage extensions to the IR. Derivations from the base IR are called dialects. The well-defined "Standard SUIF" is the central dialect. It is possible to register new dialects and variations of existing dialects and to keep track of the dialects that a certain pass can handle.
- It additionally offers a CFG representation, which is very useful to describe unstructured control-flow.

[23] Unfortunately, C++ encourages (or at least does not hamper) procedural style in object-oriented programming.

- It has a more expressive instructions set (e.g., instructions for short-cut boolean expressions and SSA form).
- It has a more flexible type scheme. The specification of a type's memory layout can be deferred until it is actual needed. For example, a front end can define a group type without specifying any alignment or padding information for the fields. A back end later fills in the missing information in order to generate code for a specific machine architecture. In contrast, SUIF 1.0 assumes that the field offsets are known up-front.
- Its *interpretation framework* allows any piece of SUIF code to be evaluated by providing an interpreter class for each dialect. This functionality might be useful for constant folding and *partial evaluation* [CD93]. Furthermore, the interpreter code serves as the exact definition of what a dialect means.[24]
- SUIF 2.0 passes can be composed in memory. In contrast, SUIF 1.0 requires file I/O to sequentially execute passes. It is still possible in SUIF 2.0 to write out the result of a pass into a file, which is useful during development and debugging. However, no intervening file I/O during passes greatly improves the overall speed of compilation and moves SUIF closer to the domain of a production compiler.
- SUIF 1.0 passes have to be very conservative when encountering an instruction that they do not understand because no information about the effect of the instruction is available. SUIF 2.0 allows the definition of *semantic properties* for user defined instructions. Such properties specify, for example, whether the instruction has a side effect, or whether it is commutative. Thus, passes that do not know about the exact semantics of an instruction are still able to extract useful information for optimizations.
- Development experiences with SUIF 1.0 uncovered a number of design decisions that turned out to be inconvenient—SUIF 2.0 remedies these. For example, `if` statements in SUIF 1.0 contain a header part that tests for the condition. This header contains an explicit branch instruction to an implicitly defined `else` label. This turned out to be awkward to handle for optimization passes. SUIF 2.0 defines the condition as an expression (like C), thus making the control flow implicit.
- The use of annotations is often no longer necessary. The same effect can usually be achieved in a more consistent and intuitive way by extending SUIF 2.0 constructs (e.g., types, symbols or statements) directly. For example, one can introduce a new statement to model a C++ try block (presumably derived from the `scope_statement` class). SUIF's goal is to provide extensibility in a way that allows existing code to continue to work. For example, even if old passes do not understand all about a new construct, they should be able to be conservative about the semantic of the new construct. One way to achieve this is to use semantic properties.

Note that SUIF 2.0 does not extend the functionality of SUIF 1.0 towards non-imperative languages, but provides far better support for implementing those extensions. OSUIF (refer to chapter 5) is such an extension for object-oriented languages.

[24] Using the interpreter code to define the semantics of a SUIF construct is certainly questionable because bugs in the interpreter code become features of the semantics of a SUIF construct, but the practical advantages overshadow this rather theoretical concern.

Chapter 5

OSUIF

Object SUIF (OSUIF) [DCI+97][1] is an extension to SUIF 2.0, which provides support for the representation of object-oriented languages. It can be viewed as the sixth layer of the SUIF 2.0 system.

5.1 Motivation

SUIF is well suited to capture high-level information for code that has been translated from an imperative programming language, but it lacks support for object-oriented languages.

Object-oriented languages require additional—or completely different—high-level information than imperative languages. OSUIF defines a small number of specific object-oriented constructs (e.g., support for dynamic dispatch) that retain the high-level information of an object-oriented source language as long as necessary. Once this information is no longer needed (i.e., all optimization passes for object-oriented features have been run), OSUIF code is lowered to Standard SUIF code so that all generic SUIF passes can be used for the remaining compilation process.

One might wonder why a compiler framework specifically tailored to object-oriented languages is desirable. First, there has been a major shift towards the object-oriented paradigm—almost all important languages support it (e.g., C++, Java, Eiffel, Ada-95, and OO-Cobol). Hence, a generic compiler framework must offer support for object-oriented features in addition to imperative features.

Second, object-oriented languages are in more need of good compiler optimizations due to their different runtime behavior compared to imperative languages. This difference is caused both by programming style (e.g., smaller methods and consequently more frequent method calls) and by language features (e.g., late binding). Besides, most object-oriented languages require a sophisticated runtime system to support exception handling, threads, and/or garbage collection. Thus, using generic components to build (parts of) the runtime system can save a lot of work.

Finally, while it is certainly possible to write a SUIF front end for an object-oriented language, such a compiler would not serve as a good vehicle for object-oriented compiler research. For example, if one wants to express a dynamic method call in SUIF, a concrete implementation technique has to be chosen and implemented with the available SUIF instructions. Thus, high-level information is lost that could have been used for code optimization (e.g., class hierarchy analysis [DGC95] [DMM96] or type inference [CU90] [PC94] [AH95]). OSUIF overcomes this problem by introducing a new, high-level IR constructs that model dynamic method calls, classes, class hierarchies, etc. This approach is analogous to the high-level array instructions that SUIF already offers to make array accesses explicit.

[1] This paper reflects the OSUIF design at a rather early design stage; parts of it are no longer accurate.

5.2 Introduction

OSUIF's goal is to extend SUIF with object-oriented features, thus allowing to express object-oriented behavior in SUIF and to provide a framework that makes it possible to share passes that are specific to object-oriented languages. To give a first impression, we briefly list the features provided by OSUIF:

- classes with static and instance members.
- dynamic method calls.
- object creation and destruction.
- relationships between classes.
- exception handling.
- generic lowering and object-oriented optimization passes.
- accurate garbage collection (planned).
- type sets (planned).

Unfortunately, object-oriented languages are very diverse and hence it is hard to design a consistent, common framework that encompasses all languages. For example, different object-oriented languages usually have a very different notion of inheritance, subtyping, subclassing, and dynamic binding. In contrast, imperative languages have a fairly uniform semantic and type system. For example, SUIF accommodates both C and Fortran in a single type system (and support for Pascal and Modula-2 could probably be realized without any extensions to the existing type system). Thus, owing to the diverse nature of object-oriented languages compared to imperative ones, OSUIF takes a different approach.

Instead of trying to contain the union of all object-oriented languages, OSUIF provides an extensible framework, which abstracts away the details of a particular language. Each front end must extend the framework in order to represent its source language. A Java compiler, for example, would subclass from (abstract) OSUIF classes in order to implement front-end–specific functionality. The front end can also make use of certain functionality that OSUIF already provides, such as a generic pass that builds virtual function tables for single inheritance.

To summarize, OSUIF clients have to extend the framework in order to model the semantics of a specific source language, and then build a front end that depends on these extensions.

5.3 Overview of OSUIF Extensions

Figure 5.1 shows the OSUIF classes that subclass from SUIF. (For every OSUIF class C, a generic version `generic_`C and sometimes an argument builder C_args for the I/O code is implemented. These classes are not shown.) All leaf classes in the class hierarchy graph are OSUIF classes.

In particular, OSUIF extends the SUIF 2.0 class hierarchy in the following areas:

- `types`: to express types for classes, methods, and fields.
- `symbols`: to express symbols for fields and methods.
- `symbol_table`: to model class-specific behavior for member lookup.
- `instructions`: to express exception handling and object-oriented instructions.
- `statements` and `cfg_nodes`: to model exception handling.

The following sections discuss these extensions in more detail.

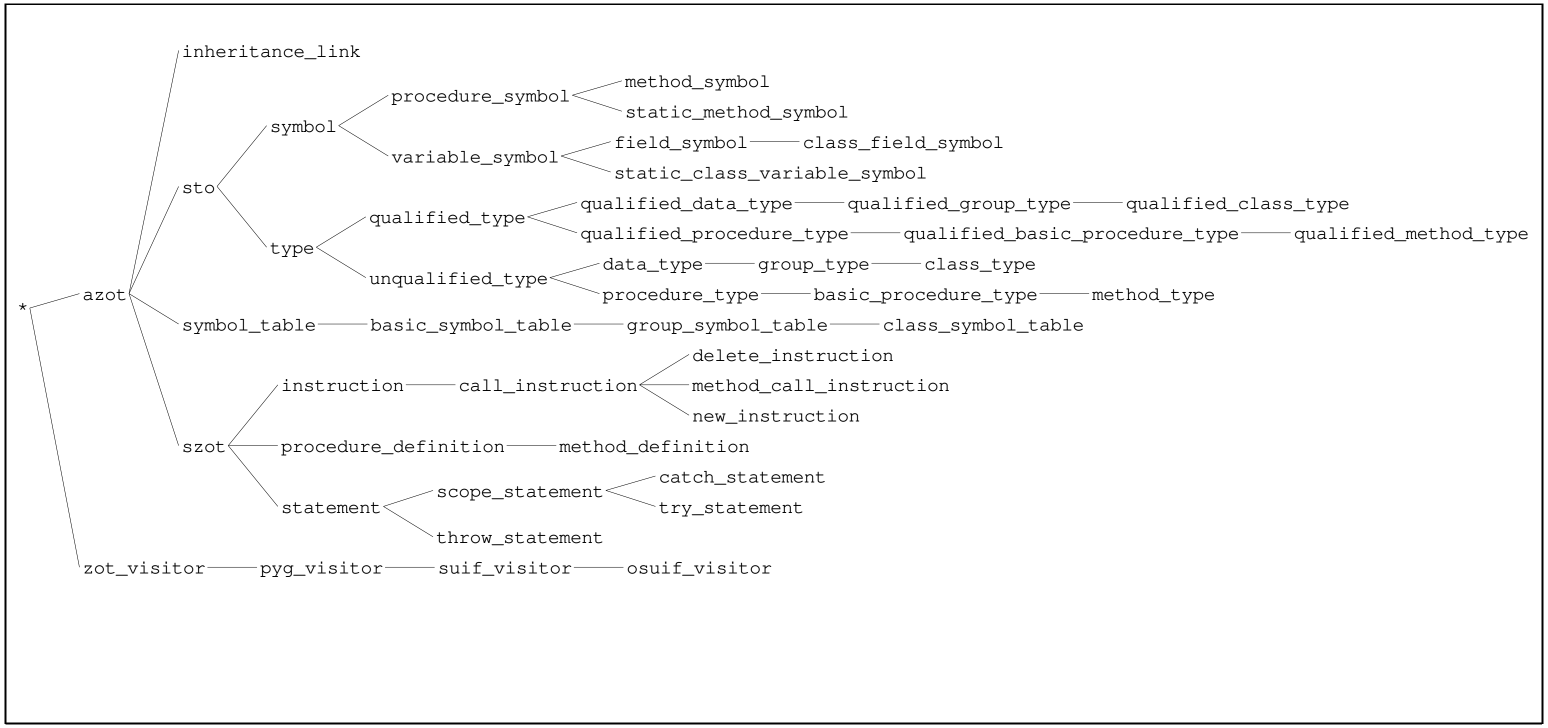

Figure 5.1: OSUIF class hierarchy (generic classes omitted)

5.4 Classes in OSUIF

To model object-oriented features, OSUIF introduces the (abstract) concept of classes, methods, and fields. An OSUIF class can have

- static (i.e., per-class) *members* (i.e., static fields and methods).
- instance members (i.e., instance fields and methods).[2]
- semantic attributes.
- links to its ancestor and descendant classes.

Moreover, a class introduces a new scope for its members. These properties seem general enough to apply to all object-oriented languages.

The class `class_type`, which is derived from `group_type` (see section 4.5), represents a class in OSUIF. A class type has two symbol tables. One holds per-class members, the other one holds per-instance members. Furthermore, the class can contain `inheritance_links` (see below), which are used to represent direct ancestor and descendant classes.

Note that members have no specific semantics, because OSUIF does not know anything about the source language. For example, the front end must support/enforce language-specific behavior (e.g., access restrictions), and language-specific passes must be written in order to exploit language-specific features (e.g., `static final` fields in Java).

A class type has a list of *attributes*. An attribute is a simple string[3] that can be used to attach front-end–specific information to the class type. For example, a Java front end can use attributes to distinguish between a Java `class` and `interface`.

A class type keeps its instance members in the symbol table it inherits from group type. This design decision yields the most intuitive interface because the per-instance part of a class type can be viewed as a group type that has methods in addition to fields. One important difference is that a class type usually holds only the instance fields that it defines (as opposed to the instance fields it can access via inherited fields), but this is merely an implementation choice.[4] Furthermore, a variable symbol with a class type represents only the instance fields of the class type, and a variable definition reserves storage only for these fields. (In fact, the memory layout of the instance fields of a class type is identical to a group type that represents a `struct` and contains the same fields.) Hence, declaring a variable and defining storage for a class type works the same way as for a group type.

Note that a front end can subclass from `class_type` if it needs more specific behavior for a certain source language.

subsectionIncomplete Class Types

Since classes are modules with well-defined interfaces, a compiler can easily generate separate code for a single class. We assume *separate compilation* of classes in the following.

Generated code for a certain class usually contains method calls and field accesses to other classes. In the following, we call the class we currently want to compile the *main class* and all other classes that this class references its *auxiliary classes.*

In order to generate code for method calls and field accesses to auxiliary classes, the class types of these classes must be included as part of the code. This is even the case if just a reference to an auxiliary class is used and can result in a large amount of superfluous information in the symbol table, because

[2] This thesis uses Java's terminology to talk about (generic) object-oriented concepts.

[3] SUIF has a similar mechanism, which allows to add qualifications to types. The important difference is that SUIF qualifications describe behavior of storage and are associated with types, whereas attributes describe arbitrary semantic properties and are associated with OSUIF symbols and the class type.

[4] An OSUIF lowering pass can be run to copy fields down the inheritance hierarchy.

usually only a small fraction of the fields and methods of an auxiliary class are actually referenced in the generated code. For example, the private part of an auxiliary class is never visible to the main class and hence can always be omitted.

OSUIF allows to specify an *incomplete class type*, which is a subset of the complete class type. The given subset typically is only the interface that is actually referenced by the main class. A class type has two boolean flags to mark separately if instance fields or instance methods are incomplete.[5] This scheme is more flexible than the "forward declaration" that Standard SUIF offers.

For a typical compilation, the class type of the main class is complete, but the class types of the auxiliary classes are mainly incomplete. OSUIF will provide a linker that merges (incomplete) class types.

5.4.1 Symbol Table

As mentioned before, a class type uses separate symbol tables for static and instance members. Static members are kept in a `basic_symbol_table`, which can be accessed with the `get_per_class_symbol_table()` method.

```
class class_symbol_table : public group_symbol_table {
private:
  adlist_tos<method_symbol*> _methods;
  ...
public:
  // lookup for methods
  virtual array_tos<method_symbol *> *
    lookup_methods(string name) = 0;
  virtual array_tos<method_symbol *> *
    lookup_local_methods(string name);
  ...
  // lookup for fields
  virtual array_tos<field_symbol *> *
    lookup_fields(string name) = 0;
  virtual array_tos<class_field_symbol *> *
    lookup_local_fields(string name);
  ...

  boolean is_subclass_of(class_type*);
  boolean is_direct_subclass_of(class_type*)
  boolean is_subtype_of(qualified_class_type*);
  ...

  virtual boolean is_dispatched(method_symbol*) = 0;
  virtual method_symbol* redefines(method_symbol*) = 0;
};
```

Figure 5.2: OSUIF class symbol table interface

A first, naïve approach for instance members would probably reuse SUIF's `group_symbol_table` (which class type inherits from group type). OSUIF takes a different approach because it wants to

[5] The boolean flag that specifies if fields are complete is actually inherited from group type.

offer a convenient and uniform interface for field and method lookup. This interface is realized by the abstract class `class_symbol_table` (see figure 5.2). The class's interface defines two different strategies for member lookup:

- *Local lookup* searches only the local symbol table. This lookup corresponds to the functionality that SUIF already offers for fields.
- *Transitive lookup* typically searches up the inheritance chain to find a matching member. This lookup algorithm is not implemented by OSUIF. Instead, it must be implemented by the front end, which captures language-specific semantics of the lookup, such as method overriding, method renaming, and method hiding rules.

Furthermore, class types contain the interface for inquiring about specific object-oriented properties. For example

- `is_dispatched()` is true for method symbols that are dispatched. Typically, only non-constructor, non-private, virtual methods are dispatched (e.g., C++ and Java).[6]
- `redefines()` returns the method symbol of the "closest" superclass that is overridden by the method symbol given as argument. This provides a generic interface for constructing a concrete dispatch mechanism.

By redefining these two methods, a front end decides which methods are dispatched and how. OSUIF provides a default implementation for single inheritance. OSUIF also offers a generic virtual function table pass for single inheritance that makes use of the above methods.

A typical front end needs to subclass only from the class symbol table.[7] OSUIF already provides a (concrete) class symbol table for single inheritance, which redefines the abstract methods of `class_symbol_table` with default implementations.

5.4.2 Representation of Inheritance

A class hierarchy (i.e., subtype and/or subclass relationships between classes) can be modeled with inheritance links (`inheritance_link`). A class hierarchy is essentially a directed acyclic graph (DAG). The nodes of the DAG are classes and the edges are inheritance links, which describe relations between classes.

An inheritance link has a list of attributes. An attribute is a simple string that can be used to express the meaning of inheritance. In C++, for instance, it could be a `public` vs. `private` and `virtual` vs. `non-virtual` inheritance link. Of course, one can also subclass from `inheritance_link` to model specific behavior. For instance, as an alternative a C++ front end might want to subclass inheritance link with a `virtual_link` and a `non_virtual_link` instead of using attributes.

Note that all OSUIF classes that subclass from `symbol` implement such an attribute list, to represent front-end–specific information.

5.5 Instance Fields in OSUIF

An instance field is represented with a `class_field_symbol`. It extends a `field_symbol` with a list of attributes. An attribute is used to describe additional, source-language–specific information. For

[6]Class hierarchy analysis can be used to obtain a more accurate result of the methods that actually require dispatch.

[7]This is a useful property since subclassing from (O)SUIF involves a lot of work due to the standard functionality that SUIF requires a class to offer.

instance, a Java front end could model access restrictions with `public`, `protected`, `private`, and `package` attributes. An instance field behaves exactly like a group field.

An instance field access is represented with the SUIF `field_access_instruction`. This instruction takes a field symbol and an address (which is typically the base address of a variable symbol that represents a group or class type) as its operands. In SUIF, this instruction is usually used for field access of a group type, but because `class_field_symbol` is derived from `field_symbol` the same instruction is applicable in OSUIF.

Note that a class field symbol (as well as a field symbol) cannot directly be used in a variable definition to reserve storage. It is possible, however, to reserve storage for all instance fields of a class type (see to section 5.4).

5.6 Instance Methods in OSUIF

An instance method (`method_symbol`) subclasses from a procedure symbol and allows the user to distinguish the receiver. Furthermore, it associates the method with its class type and provides an attribute list.

By convention, the first formal parameter of the method is its receiver (e.g., the `this` reference in Java and C++). However, the user is still responsible to explicitly define the formal parameter and type that represents the receiver. Typically, the type of the receiver is a pointer to its defining class type. The *defining class* of a method or field is the class that first introduces the method or field, respectively.

The type of a method is represented with the class `method_type`, which is derived from `procedure_type`. The only extension that the method type provides is a more convenient and type-safe interface for the receiver type.

SUIF's `procedure_definition` is used to define SUIF procedures as well as OSUIF methods because both have the same behavior (and thus can share their implementation).

5.7 Static Members in OSUIF

Static fields and methods are represented with `static_class_variable_symbol` and `static_method_symbol`, respectively. Both extend the interface with attributes and have a reference to their defining class. This reference is useful if the symbol is moved out of its class scope. The standard OSUIF lowering, for example, moves static members from the per-class symbol table to the global symbol table. This reference makes it also possible to undo the lowering by moving a symbol back to its defining class scope.

Static field symbols (as opposed to `class_field_symbols`) represent only a single storage location (if any). Hence every static field symbol is implicitly associated with a unique storage location. For this reason, all SUIF instructions that work for non-group types are applicable to static fields. For example, `load_address_instruction` can be used to obtain the address of a static field.

5.8 New Instructions

OSUIF provides additional constructs to model specific object-oriented behavior at the instruction level.

5.8.1 Method Call Instruction

The most distinctive feature of object-oriented methodology is the dynamic binding of method. A compiler may have considerable latitude in how to implement it [DHV95]. For this reason, OSUIF provides an instruction to express message sends at a high level. The `method_call_instruction`, which is derived from `call_instruction`, keeps information about the dispatched method (`m`), the receiver(`r`), and the actual parameters of the call:

```
r.m(...)
```

The method `m` is represented by a method symbol, which states its name, signature and defining static class. A lowering pass resolves the method dispatch, by replacing `m` with code that calls the actual method. A statically bound call typically replaces `m` with the address of `m` or one of its redefinitions. A dynamically bound call could generate a virtual function table access through `m`'s slot number.

The receiver `r` can be, for example, a variable symbol whose type is a pointer to a class type (that is a subtype of `m`'s class type). Note that OSUIF poses no restriction on the receiver type. It is planned that these restrictions are introduced by OSUIF dialects (see section 4.7).

Subsequent passes have access to the method call's actual parameters through the inherited interface (which shows the receiver as the first parameter) and direct access to the receiver through the new interface. Note that OSUIF only supports a single receiver (i.e., no multiple dispatch).

5.8.2 New and Delete Instruction

The `new_instruction` (derived from `call_instruction`) creates a new class instance. Creating a new instance means by default to allocate memory and to make a constructor call to initialize the new instance. This behavior corresponds to the usual semantics of object-oriented languages, such as in Java, C++, and Eiffel. Alternatively, the new instruction can be used to only allocate (and initialize) memory, which correspond to the semantics of the Java Virtual Machine `new` bytecode. The new instruction is also useful for type inference because the precise object type is known at these program points.[8] OSUIF also provides a construct to express the explicit destruction of an object (`delete_instruction`).

5.9 Exception Handling in OSUIF

Although, strictly speaking, not part of the object-oriented paradigm, many modern object-oriented languages support exception handling. Since SUIF does not support it, OSUIF must provide it to cover the full semantics of languages such as Java, C++, Eiffel, and Ada-95.

The semantics of exceptions vary considerably across popular languages. For example, some languages support the termination model (e.g., Java, C++, Modula-3, and Ada-95), whereas other languages use the resumption model (e.g., Eiffel, PL/1, and Cedar Mesa). Exception objects (e.g., Java and C++) or simple named entities (e.g., Ada) transfer information from the throw-point to the exception hander. Java and Eiffel add runtime code to keep track of the call chain (which is useful to debug uncaught exceptions); C++ does not.

Exception handling can be implemented with a number of different techniques. For instance, a portable albeit slow technique is based on the use of the C Library `setjump()` and `longjump()` functions [Rob89]. More efficient, PC-based techniques use in-code markers or PC range tables [Cha95].

In keeping with the design philosophy outlined before, OSUIF aims to provide a framework for expressing the essential and common behavior of exceptions.[9] Clients of OSUIF may use standard OSUIF lowering passes or implement their own passes that lower these constructs to OSUIF or SUIF. Such passes may investigate, for example, the trade-offs between optimizing the speed of exception dispatch (i.e., transferring control flow to the appropriate handler) vs. reducing the penalty that an exception handling implementation causes even if no exceptions are thrown.

[8] Alternatively, a method call, which calls the constructor, could be used instead of the new instruction. In this case OSUIF's type system (refer to section 5.10) could be used to indicate that this program point has a precise type.

[9] OSUIF (and this section) use the terminology of C++ for exceptions.

OSUIF will provide additional constructs to model exception handling control flow for SUIF instruction, statement, and CFG representation. The representation of exceptions at the statement level is straightforward. SUIF already provides a `scope_statement` class, which represents a new scope. OSUIF represents a `try`-block by deriving from this class. In addition to the `scope_statement`'s body, a list of `catch` statements is attached to it. A `catch_statement` introduces a new scope, states the type of exception it catches, and contains a pointer to its belonging try-block. To express the actual throwing of an exception, OSUIF provides a `throw_statement`.

As of this writing, exceptions at the instruction and CFG level have not been fully designed. The differences between exception handling for instructions/CFGs and statements is essentially equivalent to the differences of the exception handling model for the Java language and the Java Virtual Machine (see section 6.9). For example, exceptions at the instruction level are more "fine-grained" because an arbitrary section of code can constitute a try-block. The try-block code section can be specified with code labels. For the CFG representation the section can be represented with a list of CFG nodes. Exception handling control flow is expressed with an `exception_test_instruction`. This instruction behaves like a `branch` instruction and hence is derived from it. The branch is taken if the exception condition is true. This mechanism makes exception handling control flow explicit (and allows to distinguish between "normal" and exception handling control flow). The CFG representation has a corresponding `exception_test_cfgn` (which is derived from `branch_cfgn`).

Note that a back end (or OSUIF lowering pass) does not necessarily generate branch instructions for `exception_test_instruction` and `exception_test_cfgn`—both constructs are merely used in OSUIF to provide a high-level abstraction for exception handling control flow. Typically, a back end would use a more efficient scheme that preserves the control flow, but does not introduce exception handling code that is executed during "normal" control flow [Cha95] [Cha96].

5.10 Enhanced Type System

OSUIF provides type equivalence tests for (qualified) method types and method symbols with the binary method `is_equivalent()`. The standard implementation of *equivalence* is defined as follows:

Definition 5.1 (Type equivalence) *If not otherwise specified for a particular type, two types are equivalent if they are represented by the same type object (i.e., they are pointer-equivalent).*

Definition 5.2 (Method type equivalence) *Two method types are equivalent if they have the same number of formal arguments, the same number of return types, type equivalence for all corresponding formal arguments except for the "this" reference (which is the first formal argument by convention), and type equivalence for all corresponding return types. No restriction is imposed upon the "this" reference.*

This definition corresponds to non-variance of formal argument types and return types, such as specified by Java and C++.[10] OSUIF does not restrict the type of the "this" reference, but certain OSUIF dialects will impose restrictions on it (e.g., it must be a `void*` or a pointer to a class type).

Definition 5.3 (Qualified method type equivalence) *Two qualified method types are equivalent if their unqualified types are equivalent (i.e., if they are method type equivalent).*[11]

Qualifications are ignored for the equivalence test because they depend on the source language. In C++, these qualifications yield new types.[12] Java does not have qualifications on method types.

[10] A more general scheme would allow contravariance of formal argument types and covariance of return types, but the current definition is simple and sufficient for Java, C++, and Ada-95.

[11] SUIF introduces the qualifications `volatile` and `const` for types.

[12] Actually, this is only true for method overriding, but not for `catch` handler matching.

Definition 5.4 (Method symbol equivalence) *Two methods symbols are equivalent if they have identical names and if their types are method type equivalent.*

Note that attributes, such as `public`, `final`, and `abstract` are not part of the equivalence check. This seems to be a good default behavior because these attribute usually do not introduce a new type. In Java, a `native` method can be redefined with a non-`native` method (and vice versa) and a non-`final` method can be redefined with a `final` method (and vice versa). In C++, a `protected` method can be made `public` when it is redefined. The exceptions that a method can raise are ignored for the equivalence check. This means that both the `throw()` exception specification in C++ and the `throw`-clause in Java are not part of the method's type.

OSUIF also provides an interface for expressing subclass and subtype relationships; by implementing them a front end provides the meaning. Subsequent generic passes can, for example, carry out analyses that need to know type compatibility rules without having to know details about the source language.

5.10.1 Type sets

OSUIF provides functionality to express *type sets*. A type set contains class types as its elements. The following type set constructors are defined:

Constructor	Meaning
Unknown()	Set of all classes
Class(C)	$\{C\}$
Cone(C)	$\{x \mid x\ \textit{is subclass of}\ C\}$

The operators $\cap$, $\cup$, $\setminus$, and $\in$ are applicable for type sets.

Type sets are needed for concrete type inference because this analysis must (conceptually) maintain a mapping from variables to types at each program point. Usually, the type sets contain the potential types (i.e., types that the variable symbol might hold). The set of real types (i.e., the types that a variable symbol actually holds during a program execution) is a subset of the set of potential types. OSUIF's type system is similar to the ones that have been designed for type inference in Vortex [DDG$^+$96], Self-90 [CU90], and Typed Smalltalk [Joh96] [JGZ88].

5.11 Lowering Passes

Let's consider a SUIF pass that translates a high-level SUIF construct (e.g., `switch` or `for` statement) to a semantically equivalent sequence of low-level statements (or instructions).[13] This transformation is usually straightforward and does not allow a broad variety of different implementation strategies. Hence, a standard SUIF pass is usually sufficient and the user seldom needs to customize it.

In contrast, the translation of a (high-level) OSUIF construct to a sequence of SUIF statements (or instructions) can have much more diverse implementation strategies, each of them having totally different space and time complexity for translation and run time. For example—without going into detail—the encoding of an OSUIF `method_call_instruction` in SUIF can be realized with various different implementation techniques, ranging from virtual function tables to polymorphic inline caches [DHV95].

Part of OSUIF are several *lowering passes*, which takes care of this problem. A lowering pass typically translates a certain OSUIF construct either directly to SUIF or some other, more low-level form, of OSUIF. (Therefore, the actual lowering to SUIF can be performed by several cascading lowering phases.) Passes are designed to be extensible. This ensures that different implementation techniques for an OSUIF

[13] These kind of "dismantling" passes are implemented by the SUIF 2.0 *porky* library.

construct can be easily exchanged. Thus, the user can experiment with different lowering passes in the compilation process and compare the static and dynamic properties of the resulting code.

It is also possible to run certain SUIF passes even before the lowering. Code optimization is more flexible if we allow both OSUIF and (certain) SUIF passes to run interchangeably on the OSUIF representation. This is achieved by changing the behavior of the inherited interface as little as possible. For example, if a class type is only accessed through the interface defined by group type, it actually behaves like a group type. This means, of course, that only the instance field of a class type are visible to a SUIF pass and that these instance field are interpreted as group type fields.[14] This behavior is often appropriate, for example, for a scalar optimization pass.

OSUIF implements a lowering pass for most of its constructs. Even if a generic pass cannot be used by a certain front end, it should serve as a good starting point for a more specific implementation. Some examples are:

- Lowering of static class members to plain SUIF by moving them to the global symbol table.
- Propagation of instance fields down the inheritance graph in order to get the complete object layout for a class.[15]
- Construction of virtual function tables for single inheritance.
- Construction of dispatch code (for a method call instruction) for virtual function tables.
- Unwinding of the call stack and implementation of the dispatcher code for exception handling (planned).
- Generation of stack tracing code associated with exceptions (planned).

5.12 Raising Passes

It is planned to provide passes that reverse a certain lowering pass, i.e., these passes recover the original OSUIF code as it was before the lowering. This is only possible if sufficient information is present in the lowered code. An example for such a raising pass is the moving of static fields and method that have been put into the global symbol table back to the per-class symbol table of the defining class type.

5.13 Assessment of OSUIF

OSUIF is still under active design. A lot of the planned functionality is missing. Nevertheless, a first assessment of OSUIF itself and of the SUIF extensibility mechanism can be given.

OSUIF is the first system that subclasses from SUIF 2.0. It is a proof-of-concept for the SUIF extensibility scheme. Indeed, no changes to SUIF 2.0 were necessary to accomplish OSUIF's implementation.

On the downside, SUIF's extensibility scheme poses a serious burden on the programmer whenever a class needs to be derived from (O)SUIF. A generic class, a builder class, and methods for tags, the zot interface, and cloning must be implemented to provide the full functionality that SUIF demands from all of its classes. None of these extensions are trivial to implement.[16]

OSUIF and Vortex (see section 2.2) share the same goals. Both are systems for object-oriented compiler research. Both systems define an IR, which incorporates object-oriented constructs, and provide (generic) optimization passes. Vortex implements object-oriented and traditional optimizations, selective

[14] We assume that non-local instance fields have already been propagated down the inheritance graph.
[15] Recall that before the lowering a class type usually contains only the instance fields that the class defines.
[16] Work is in progress to simplify parts of the extensibility mechanism.

recompilation, and exceptions. OSUIF will provide object-oriented optimizations and exceptions; SUIF 2.0 will provide standard optimizations, separate compilation, and several back ends. OSUIF's IR is more flexible and extensible (due to the mechanism introduced by SUIF). On the other hand, OSUIF currently lacks many of the front ends and object-oriented optimization passes that Vortex offers.

OSUIF's usefulness as a research platform certainly depends on the functionality and genericity of its passes and on the question if its generic passes can be truly shared among various front ends.

Chapter 6

The Java Virtual Machine

The first section gives the definition and concepts of a virtual machine and some examples of virtual machines. The rest of this chapter shortly discusses the Java Virtual Machine.

6.1 Virtual Machines

An *abstract*, or *virtual machine* (VM) is a fictitious computer.[1] Like a real computing machine, it has an instruction set architecture and a memory model. A VM realizes an abstract layer that sits on top of a real machine architecture. Hence, VMs are architecture neutral and highly portable.[2] Programs that "run" on a VM can be either interpreted by a real machine or compiled to machine code. A compiler that targets a VM translates a source program into a sequence of VM instructions.

Another alternative to achieve portability is to compile to a (high-level) programming language that is available for many architectures. ANSI C is the primary choice, but recently C++ and Java have been targeted as well. For example, the NESL programming system translates its IR (called VCODE) into Java source [HS96][3]. The authors state they used "Java as a portable dialect of C with garbage collection and a good collection of preexisting data structures." Java is attractive as a target language for several reasons. First, it is strongly typed, making debugging of generated code easier. Second, it provides garbage collection, thereby removing a large source of potential memory-management bugs in the generated code, and eliminating the implementation of a garbage collector itself if the source language requires one. Third, the generated code benefits directly from improvements in Java compiler technology.

Often VMs have been designed with a specific source language in mind. P-code [ANJN75] [NAJ$^+$81] is a stack-based VM for Pascal compilers (that can be either interpreted or compiled to native code). Fcode is a machine-independent binary representation of Forth. Smalltalk [GR83] and Lisp systems often use a compact bytecode representation that can be interpreted more quickly than source code and can be exchanged between different machines.

VMs are either register-based or stack-based. Using a stack-based architecture for a virtual machine has the following benefits:

- They are conceptually simple. For instance, the concept of l-values and r-values is not needed, the evaluation order of expressions is implicit, and the effect of an instruction can be easily expressed

[1] A VM should not be confused with an intermediate representation, even though a VM is sometimes an IR (e.g., three-address code).

[2] Usually the (re-)implementation of a (simple) interpreter is sufficient to port the VM to another system.

[3] In this paper the authors refer to Java as an IR; however, we prefer to think of it as a target language.

with a stack effect notation (refer to section 6.8).

- Code generation is straightforward for most target languages. This is most obvious for arithmetic expressions. Furthermore, no register allocation needs to be performed.[4]
- They can be efficiently implemented as interpreters (e.g., with *(in)direct threading* [Kli81]).
- They impose fewer arbitrary limits (e.g., no limited number of registers).
- The program representation is very compact,[5] which is beneficial for storage and transfer over a network.

On the downside, stack-based VMs are not well suited as an intermediate representation for optimization passes. Furthermore, generationing of efficient code for register machines is complicated. Ertl [Ert96] summarizes the following implementation techniques for translating stack-based languages to register machines:

Interpretation: The major advantage of direct interpretation is simplicity. A straightforward translation consists of a designated memory area that contains the stack entries and a register that holds the address of the top-of-stack entry.

Native code compilation: This technique requires a full (native) compiler. Typically, the compiler applies special compilation techniques for stack-based source. For instance, Ertl mentions static single assignment form [CFRW91], treatment of unknown stack effects, and dynamic stack manipulation.

Translation to C: This translation can take full advantage of the C language as a portable target language. This approach is very similar to the translation to the SUIF intermediate representation, (which has been designed to represent imperative languages such as C).

6.2 Overview of the JVM

The *Java Virtual Machine* (JVM) [LY96] [MD97] is stack-based. JVM instructions (or *bytecodes*) are typed; data (e.g. local variables) is untyped. The instruction set shows low-level as well as high-level characteristics. For example, it contains bytecodes for high-level object-oriented features (which are specifically tailored towards the Java programming language). On the other hand, control flow instructions are low-level and can only express unstructured control flow.

Many JVM features[6] directly reflect the design goals of the Java language [GM95]. In fact, the JVM and the Java language are tightly coupled. For example, the type system and runtime type checks in the JVM match the Java type system, dispatch bytecodes provide exactly the functionality that is needed for Java, the exception models are almost identical, etc. This coupling is tight, but not seamless. For example, object creation in the JVM is done in two steps. First, the `new` bytecode is executed, which returns (initialized) storage for the object. Second, the constructor is called with the `invokespecial` bytecode. Furthermore, the JVM does not require exception handlers to be properly nested and control flow is expressed differently. Finally, lexical scopings and types/names of local variables are lost.

[4] Actually, it is possible to perform *stack allocation* to reduce the number of load instructions [ME98].

[5] This is also true for JVM bytecodes, but the effect is usually foiled by the space-consuming representation of the constant pool.

[6] The buzzwords robust, secure, network-oriented, and portable come to ones mind

6.3 Type System

The JVM has the following data types:

- *primitive types.* A primitive type is an integral type (`byte`, `char`, `short`, `int`, `long`), or a floating-point type (`float`/`double`). The bit-representation of all primitive types is specified (as opposed to other VMs). For example, an `int` value is a 32-bit signed two's-complement integer. This scheme imposes addition constraints on a JVM implementation, but increases the portability of programs.
- `returnAddress`. This type is used for values that are indexes to the opcode of a bytecode (refer to section 6.8). It is used internally by the JVM; there is no corresponding type in the Java language.
- *reference types.* A reference to an object has the type `reference`. Values of this type can be thought of pointers to objects, or as the "object-handle". Objects are never addressed directly; they are always specified indirectly with values of type reference. This design provides more flexibility for a particular JVM implementation because the JVM specification [LY96] needs not to specify the layout of an object. For example, the specification gives no concrete representation for a `null` reference. Instead, it provides bytecodes that create one (`aconst_null`) and test for it (`ifnull`). This scheme makes it possible for every JVM to use its own internal representation that fits the underlying architecture best. There are three kinds of reference types: (1) class types, (2) interface types, and (3) array types. The value of a reference is either a reference to an object or `null`.

An object is either a *class instance* or an array. Objects live only on the heap. The type system for reference types is almost identical to that of the Java language.[7] Java's integral and floating-point types are identical to the ones defined by the JVM. Note that no boolean type exists. The JVM instruction set has no support for a boolean type, but it can be efficiently represented with an integral type.[8]

6.4 Runtime Layout

The JVM does not define storage sizes for the data types that have been introduced in the previous section. Instead, the specification defines a platform-specific size called a *word*. A double-word must be big enough to hold values of `long` and `double`. The values of all other types must fit into a single word.[9] The (conceptual) execution model consist of the following components:

- A (per thread) *Java stack* that holds *Java frames*. A new frame is pushed when a method is invoked and popped when a method completes. A Java frame corresponds to the *activation record* of an imperative language. Each frame holds the *local variable array* and *operand stack* of the invoked method. Both a local variable slot and an operand stack entry are a single word wide. Values of types that require a double-word are stored in two consecutive slots/entries. Note that local variable slots and operand stack entries never hold values of the JVM types `byte`, `char`, or `short`. These types are converted to `int`s first. Local variable slots are untyped (i.e., a slot can hold variables of different types at different program points[10]) and they are addressed by word-offset (starting with zero).

[7] Subtle differences exist, for example, the Java language has a *null type*.

[8] The JDK `javac` compiler uses `int` to represent a boolean (`0` is `false` and `1` is `true`). Other representations are perfectly reasonable too, especially considering that the Java language specification forbids the conversion of a boolean to an integer type and vice versa.

[9] A word has usually the size of a pointer on the host platform.

[10] A slot cannot hold different types at the same program point at different times because the Java bytecode verifier forbids this.

- A garbage collected *heap* that holds objects. All objects are allocated on the heap and hence all object references point to the heap.
- A *method area* that stores per-class structures such as the constant pool, method code, and static fields. Note that the constant pool need not exist (in its original form) and that method code can be represented in various ways (e.g., as JVM bytecode or as native code).
- The *execution engine* which is typically an interpreter for the JVM instruction set (implemented in software and/or hardware). Other techniques, such as off-line compilation or dynamic compilation, are briefly discussed in section 7.2. Conceptually, each thread runs on a separate execution engine.
- The *class loader* to load, link, and initialize Java class files during runtime.
- Runtime system support for exception handling, threads, dynamic linking of native methods, etc.

6.5 The Class File Format

The Java class file format keeps all the information about a Java class or interface[11] that is necessary for the JVM to (securely) execute bytecodes. A class file is the equivalent of an object file, but it contains much more semantic context than, for example, a C/C++ object file. This additional semantic information allows type checks, dynamic linking, verification of bytecode, etc.

The class file is organized as a set of nested tables. The JVM specification gives the complete definition; we only want to give the reader an idea of the kind of information that is stored in the tables by describing important entries of some tables. The top-level table contains

- the *constant pool* of the class. It it essentially a symbol table, holding symbols that represent one of the following:
  - class/interface (`CONSTANT_Class`)
  - field (`CONSTANT_Fieldref`)
  - class method (`CONSTANT_Methodref`)
  - interface method (`CONSTANT_InterfaceMethodref`)
  - name or type descriptor (`CONSTANT_Utf8`)
  - `String` literal (`CONSTANT_String`)
  - integral or floating point constant
- a flag to distinguish between a `class` or `interface`.
- a symbolic reference to the superclass (if any).
- an interface table that hold a list of symbolic references to the implemented/extended interfaces.
- a method and field table of all methods and fields that the class defines. The method table contains, besides other entries, the *bytecode array* (i.e., method body) and the exception table (see section 6.9).
- etc.

[11] Note that arrays are fixed, implicit constructs in Java (similar to Ada, unlike to Eiffel) and hence do not need a class file representation.

Type descriptors for methods and fields are represented as strings in the JVM. Note that descriptor strings only encode the type, not the name of the method or field.[12] For primitive types a single character denotes the type (e.g. `I` for `int`). A class X is encoded with the string `L`X`;` and an array of X with the string `[`X. For example, the method descriptor for the method

```
void foobar(int i, Object[] os, Thread t)
```

is

```
(I[Ljava/lang/Object;[Ljava/lang/Thread;)V
```

Note that fully qualified names for classes/interfaces are used in descriptor strings. Type strings can be easy parsed because they can be expressed with a *Simple LL(1)* grammar [Plö94].

6.6 Verification of Class Files

Java bytecode is designed to be transported across networks (from a server site) so that it can be executed by a client site (*executable content*). Hence, secure execution of untrusted programs is of a major concern [Yel95] [DFW96]. In the simplest scenario, untrusted code could wreak havoc by using illegal opcodes or operands. To ensure security, untrusted class files are subjected to the *Java verifier*, which checks all constrains that are imposed on a class file by the JVM specification. The verification of a class file (conceptually) operates in four passes:

Pass 1 is performed when the class file is loaded by the JVM. It ensures that the basic structure of the class file is valid.

Pass 2 verifies properties of the class file that can be checked without looking at the code attribute. This includes the *static constraints* of the constant pool and well-formedness of names and type descriptors.

Pass 3 checks the code array of every method. It is ensured that at any program point, no matter which code path is taken to reach that bytecode,[13]

- the *configuration* (refer to definition 8.4) of the bytecode is always the same.
- the configuration of the bytecode is type safe for the execution of that bytecode, i.e., all operations on the operand stack and local variable array are legal.
- methods calls and field accesses are type safe.

Furthermore, lots of other *structural constrains* are checked. For example, the verifier poses restrictions on the bytecodes that make it easy to ensure that no uninitialized object is accessed, or that the execution of `jsr`/`ret` bytecodes yield a proper call chain. These verifications can only be implemented with a *data flow analyzer*. (A description of such an analyzer that has been implemented for the `j2s` compiler is given in section 8.4.)

Pass 4 does all the checks that require constant pool resolution (refer to section 6.7). For example, classes that are symbolically referenced in the constant pool (with a `CONSTANT_Class` entry) must be loaded if they are not already present. For interpreters and JIT-compilers, this resolution is usually postponed until runtime and triggerd on-demand by the execution of certain bytecodes.

The JVM specification gives a complete and more elaborate description of the verification process.

[12] In contrast, the Java language specification defines *method signatures* as the name and list of parameter types of the method (but not its return value).

[13] [AD97] calls these rules the *Gosling property*. The paper explains why a `jsr` call violates this property.

6.7 Constant Pool Resolution

As mentioned before, the constant pool of a class holds symbolic references that uniquely identify a class, interface, field, or method. References into the constant pool are initially symbolic. The process that replaces symbolic references that point into the constant pool with concrete references is called *constant pool resolution*. Concrete references have a JVM specific representation as opposed to symbolic references.

Usually constant pool resolution in triggered by the execution of a bytecode whose operands refer to the constant pool. Constant pool entries can refer to other constant pool entries. Hence, the resolution of one entry can trigger the resolution of another. Because the resolution can involve external references to other classes, these classes must be loaded, linked, and initialized dynamically if this has not been done already.

6.8 Instruction Set

As mentioned before, the JVM is stack-based and strictly typed. In order to show the effect that the execution of a bytecode has on the stack, we informally introduce a *typed stack effect notation* based on [Ert96]. The stack contents is specified as a sequence of typed stack entries, with the top-most entry to the right. Only the stack entries that the bytecode manipulates are specified. The stack effect of a bytecode is expressed by giving the stack contents before and after its execution. For instance, the bytecode `iadd` pops two `int`s from the stack, computes the integer sum and pushes the result back on the stack. `iadd` has the following stack effect:

$$a_{\mathrm{int}} b_{\mathrm{int}} \rightarrow c_{\mathrm{int}}, \text{ where } \quad c := a + b.$$

Both the operand stack and local variable array are untyped (i.e., data is not tagged), but the opcode of every bytecode specifies the types it expects to find when accessing data and which types the values have that are written. This information is sufficient for the verifier to guarantee type safe execution. For example, the verifier can easily infer if the execution of an `iadd` bytecode violates its specified stack effect. Some bytecodes, like `dup` seem to be untyped, but because they do not operate on data it is not necessary to introduce a separate bytecode for every data type. The stack effect of `dup` is

$$a_{\mathrm{T}} \rightarrow a_{\mathrm{T}} a_{\mathrm{T}}, \text{ where } \quad T \in \{\texttt{int}, \texttt{float}, \texttt{reference}, \texttt{returnAddress}\}.$$

The JVM offers the following *control transfer instructions* [Cif94], i.e., bytecodes that can transfer control flow to a program point different from the next bytecode:

Unconditional jump: `goto`, `jsr`, `ret`. The last two bytecodes are used to compile the Java `finally` construct. Note that all jump bytecodes must have jump targets in the same bytecode array.

Conditional jump: `if`{`eq`, `lt`, ...} (for integer comparisons with zero) and `if_`*T*`cmp`{`eq`, `lt`, ...}.

Indexed jump: `tableswitch` and `lookupswitch`.

Method call: `invokevirtual` for class dispatch, `invokeinterface` for interface dispatch, `invoke-special` for statically bound calls of instance methods, and `invokestatic` for calls to class methods.

Method return: *T*`return` for methods that return a value and `return` for methods with the return type `void`.

Explicit exception: `athrow`.

Non-transfer instructions are bytecodes that always transfer control to the next bytecode, namely:

Comparison: `lcmp`, `fcmpl`, etc. The integer result of a comparison is 1, 0, or −1, depending if the topmost stack entry is greater, equal, or less than the second stack entry, respectively.

Arithmetic bytecodes: T`add`, T`mul`, T`neg`, `iinc`, ...

Load and Store: T`load`, T`store`, `ldc`, `iconst_1`, `aconst_null`, ...

Type conversion: The widening conversions (`i2l`, `i2f`, `i2d`, `l2f`, `l2d`, `f2d`) do not lose information "about the overall magnitude of a numeric value." A narrowing conversion can result in "a value of different sign, or of different order of magnitude, or both." Conversions never throw a runtime exception.

Object creation and manipulation: `new` and {`new`, `anew`, `multianew`}`array` for the creation of class and array instances, respectively. `getfield/putfield` for manipulations of instance variables, and `getstatic/putstatic` for manipulations of class variables. T`aload`, T`astore`, and `arraylength` are used to manipulate array elements.

Type check: `instanceof` and `checkcast`.

Stack manipulation: `pop`, `dup`, `swap`, ...

Synchronization: `monitorenter` and `monitorexit`.

Note that the instruction set is not orthogonal (i.e., bytecodes that are provided for one type are not necessarily provided for other types too).

6.9 Exceptions

A method *completes normally* if it does not cause any exception to be thrown or the thrown exception is handled within the method; otherwise it *completes abnormally*. The Java language and the JVM use the *termination model* [BW90] for exception handling [Goo75]. Exceptions can carry exception information with *exception objects*. Appropriate handlers are selected based on the type of the thrown exception object. An exception can be thrown implicitly by the JVM (e.g., if the value of the divisor for a T`div` bytecode is zero, an `ArithmeticException` is thrown) or explicitly with the `athrow` bytecode.

Exceptions for a particular method are represented in the JVM with an exception table that contains a separate entry for every handler. Every table entry contains

- an index into the code array that is the start of the handler.
- two indexes that indicate the range of the code array in which the handler is active (i.e., the associated `try`-block of the handler).
- a reference to the constant pool that denotes the `catch` type of the handler.

Exceptions in Java always have properly nested exception handlers [Yel95]. This restriction is not enforced by the JVM specification because it does not pose a security threat.

Chapter 7

Overview of the j2s Compiler

The `j2s` compiler expects a valid *Java class file* [LY96], analyzes the class file and outputs either plain SUIF 2.0 or OSUIF code (see figure 7.1).

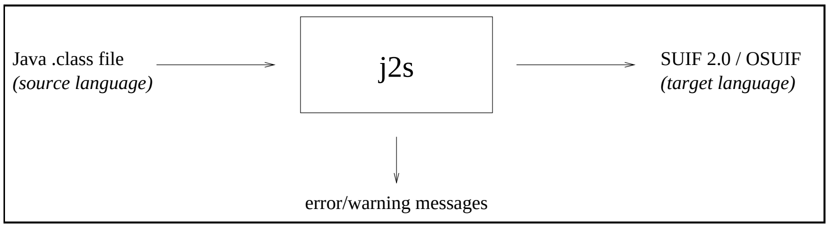

Figure 7.1: The `j2s` compiler

`j2s` issues error messages if it encounters a situation it cannot recover from, which is usually due to wrong (or unexpected) information in the class file. We implemented a large subset of the bytecode verifier and the static and structural constraints that the JVM poses on bytecodes.[1] Warning messages are generated for unsupported bytecodes (see section 7.1) or class files that pass the verifier but contain suspicious information (e.g., not properly nested exception handlers or irreducible control flow graphs).

We chose to use Java bytecodes—as opposed to Java source—as the source language for several reasons:

- The scanning of a Java class file is extremely easy; no parsing (e.g., construction of an AST) has to be done. Thus, a simple class loader is sufficient. In contrast, Java source requires the implementation of a fully-fledged front end. We chose the first approach because this allowed us to concentrate on the actual task, namely building a compiler that supports a meaningful subset of the Java specification, instead of spending most of the limited time at the rather straightforward implementation of the front end.

- It is the natural way to obtain the interface of an (external) class during compilation.[2]

- It is expected that the class file format will not change in the future. The Java language, however, might still be subject to significant changes. For example, various proposals to extend Java with

[1] Besides additional robustness of the compiler, this turned out to be quite useful for debugging.
[2] In this respect, a class file is similar to a pre-compiled C++ header file.

parametric polymorphism [OW97] [MBL97] [AFM97] [BOSW98] have already been made and it is planned that this feature will be finally incorporated into the language. Note that changes in the Java programming language do not necessarily imply changes of the JVM specification. For example, the introduction of *inner classes* [JIC97] and *reflection* [JCR97] in Java version 1.1 did not have any impact on the class file format. Similarly, all of the proposals on parametric types describe how they can be realized without changing the JVM.

- A class file contains sufficient high-level information (as opposed to traditional object files; refer to section 6.5) for the compilation process. In fact, from the perspective of the compiler, a class file contains almost the same information as Java source.

  For example, object-oriented constructs, such as object creation and method invocation, are preserved (and need not to be recovered). The only exception is structured control flow, which is lost, but can usually be recovered (see below).

- We can take advantage of specific Java source-to-source optimizations [BG97b] and bytecode-to-bytecode optimizations [BK97] [BG97a], which are just starting to evolve. These optimizations can be run before the actual compilation takes place and would not interfere with successive SUIF optimization passes.

- Java bytecodes have already been used as a target language for various languages other than Java: Ada-95 [Taf96] [CDG97b], Prolog, Cobol, a Scheme environment called Kawa,[3] an Eiffel implementation called SmallEiffel,[4] etc. This means that we get compilation from these languages to SUIF "for free." (However, it is questionable if the generated bytecode of these compilers—and hence the resulting SUIF code generated by our compiler—is suitable for compiler research.)

- Many commercial libraries are distributed only as class files. By using class files we have access to the whole program even if the sources are not available.

We chose SUIF 2.0 (see section 4) as the target language for several reasons:

- SUIF provides various tools and standard optimization passes that are helpful in the development process.

- We expect that the SUIF system will become a fairly popular platform for compiler research. Thus, `j2s` might be of interest for the research community. We also expect to benefit from contribution from other researchers.

- The SUIF 2.0 system has an extension, OSUIF, which implements specific IR constructs for object-oriented languages. These IR constructs are essential for object-oriented optimizations.

- No Java front end is available for this compiler system so far.

To our knowledge, no other compiler system unifies all of the benefits above.

The `j2s` compiler enables Java-related compiler research for SUIF, specifically the development of Java-specific, static optimization techniques.

[3] URL: `http://www.cygnus.com/~bothner/kawa.html`.
[4] URL: `ftp://ftp.loria.fr/pub/loria/genielog/SmallEiffel/`.

7.1 Supported Java Subset

`j2s` supports a meaningful subset of the Java language. For example, it is possible to compile the JDK1.2beta2 `javac` compiler, which consists of more than 500 Java classes (see section 10.1 for more details).

Currently, our compiler does not support:

exception handling: Whenever an exception would be thrown during program execution, the program is aborted and the name of the exception object is printed along with a full stack trace. (The stack trace is generated by the runtime system.) See section 9.2.15 for further details.

accurate garbage collection: No accurate garbage collection has been implemented. However, we use the freely available Boehm–Demers–Weiser conservative garbage collector[5] [BW88].

thread support: The compiler issues a warning message for code that instantiates a `java.lang.Thread` object. Compilation is not aborted because compiled thread code is not necessarily executed.

synchronization: The compiler simply ignores synchronized methods[6] and `monitorenter`/`monitor-exit` bytecodes. These constructs occur frequently in Java because all classes of the standard library are thread-safe; they can be safely ignored, however, for single-threaded code.

dynamic loading: Dynamic loading and linking of class files is not possible.

Code generation assumes that the underlaying hardware supports IEEE 754 floating-point arithmetic as required by Java. This restriction allows us to directly generate floating-point instructions, rather than calling floating-point library routines.

7.2 Compilation Techniques

The basic execution model of Java class files is bytecode interpretation. The downside of this approach is a huge performance penalty compared to native code execution.[7] So far, several compilation techniques have been proposed by the research community, mainly with the goal to remedy the inefficiency of interpreted execution and to achieve performance comparable to optimized C code:

- *Off-line compilation* generates all target code prior to execution. The target code is linked into a (static) executable. Hence, dynamic loading of unknown code is not possible. The target code can be native code, a (high-level) programming language, or an intermediate representation.

- *Dynamic compilation* [DS84] [USCH92] compiles source code to native form on demand. This means that there is no separate compilation phase. Instead, execution and (re-)compilation are interleaved. For example, the Self-91 system uses a scheme called *adaptive compilation* [HCU91]: A simple compiler generates unoptimized code when a method is first invoked. If this unoptimized code proves to be a bottleneck, a more sophisticated compiler is called to recompile and optimize the "hot spot." The Sun HotSpot implementation for Java [HBG$^+$97] [Arm98] uses the same approach, but has a bytecode interpreter as opposed to a simple compiler. Recompilation of hot spots is based on profiling data.

[5] Homepage `http://reality.sgi.com/boehm/gc.html`.

[6] Synchronized methods have the `synchronized` attribute set (rather than being wrapped with `monitorenter`/`monitor-exit` bytecodes).

[7] For example, the native code generated by the Harissa compiler [MMBC97] achieves speedup of up to 140 times (50 times on average) compared to the JDK 1.0.2 interpreter.

Just-in-Time (JIT) compilation is a specific type of dynamic compilation. The basic scheme is to compile a method to native code when it is called for the first time, pausing execution while doing so. The generated code is not modified afterwards (i.e., no recompilation or decompilation is applied to it). Smart JIT systems for Java can either interpret bytecodes or execute native code (which has been generated by a JIT compiler). A similar idea, called *continuous compilation*, overlaps compilation with program interpretation or native code execution [PC97].

- *Hybrid models* combine off-line compilation with dynamic compilation or interpretation. Harissa provides an off-line bytecode compiler and an interpreter integrated in the runtime library, thus permitting to either execute statically compiled code or interpret bytecodes [MMBC97]. We are not aware of any system that combines off-line compilation with JIT compilation.

The `j2s` compiler supports off-line compilation, which is suitable for traditional compiler research, such as static optimizations. It is the only compilation model that the SUIF system offers. Off-line compilation

- is (theoretically) not constrained by an upper limit for compile and/or response time as it is required, for example, by an interactive environment.
- sees the whole program at compile time (i.e., no dynamic loading of executable code must be taken into account).[8] Thus, interprocedural optimizations, which need to see the entire program, can be applied [CDG96].

As stated above, our compiler does not support dynamic linking of unknown classes, which would be necessary to support the full Java functionality. For example, the method `java.lang.Class.forName()` takes the name of a class and tries to load and link it into the execution environment. This call fails in `j2s` if the class is not already present in the executable. If the user knows at compile time which classes will be loaded dynamically by the system, `j2s` can be instructed to include these classes into the transitive closure of classes for the compilation process. Hence these classes appear to be dynamically loaded and linked, but in fact are already present in the executable.

In principle, dynamic linking of a class can be supported if the underlying operating system provides the functionality to dynamically link object files to an executing process. In this case, the runtime system could spawn a process that runs `j2s` on-the-fly to generate the object code for the class (including its transitive closure). Then, this code could be dynamically linked to the executable. Unfortunately, this approach would be painfully slow. Engler and Proebsting state in [EP94] that a dynamic compilation system that uses this technique requires approximately 30,000 cycles to generate a single machine instruction. Our system would probably be another magnitude slower. Furthermore, the runtime system would have to update its internal runtime data structures to reflect the new class hierarchy. For now, we don't want to place this additional burden on the compiler and runtime system, especially since the compiler focuses on a different application domain (see next section).

7.3 Application Domain

Because the `j2s` compiler uses off-line compilation and focuses on research for static optimizations, we are not interested in Java as a portable language for web-based applications, but merely as an alternative to C++ for (high-performance) object-oriented programming. Examples of such application are

- platform-independent software that uses Java as a portable distribution format.
- platform-dedicated software that uses Java because of its language features.

[8] Library code is usually not visible. For example, this is the case for `native` methods in Java.

Our compiler cannot be used for programs that depend on dynamic loading of classes and is of limited use for programs whose code does not need to be optimized. For example, this is the case for programs that have a short runtime and/or are loaded interactively over the network.

7.4 Integration into (O)SUIF

Figure 7.2 shows how `j2s` is integrated into the SUIF framework. As stated before, `j2s` generates SUIF 2.0 code—optionally enhanced with OSUIF constructs. If OSUIF code is generated, various (O)SUIF passes can be run before the OSUIF code is finally lowered to plain SUIF 2.0. The generated code can now take full advantage of the SUIF 2.0 compiler system or can be converted to SUIF 1.0 code. As of this writing, the SUIF 2.0 system does not offer any back ends besides translation of a limited subset of the Standard SUIF dialect to SUIF 1.0. Hence, we translate SUIF 2.0 to SUIF 1.0 and then use `s2c` to generate C code. (Figure 3.3 gives an overview of the SUIF 1.0 framework with `s2c`, which translates SUIF 1.0 to C code.)

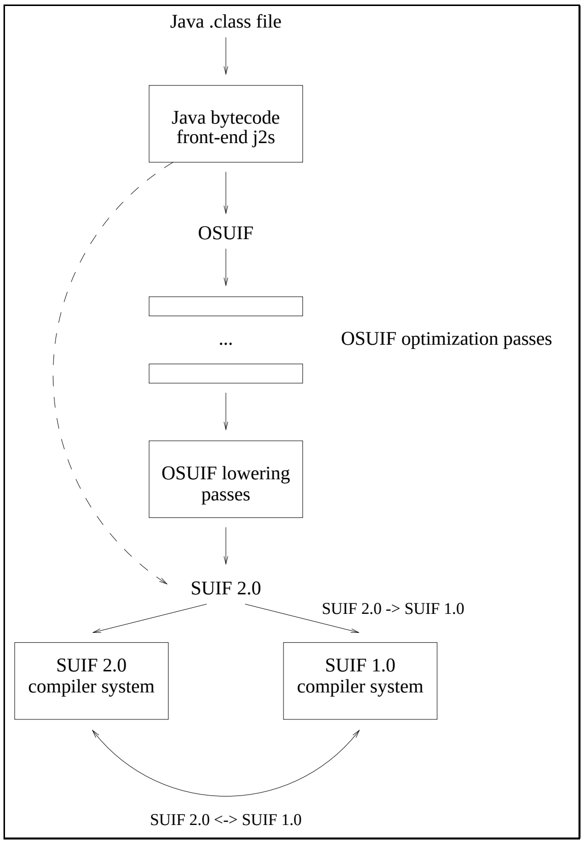

Figure 7.2: Integration of `j2s` into the SUIF framework

`j2s` is tightly integrated into the SUIF framework. It is implemented as a SUIF 2.0 front-end pass (refer to section 4.6) and uses the SUIF error and warning reporting interface (refer to section 4.2).

7.5 Compilation Passes

Our compiler implements the following passes (see figure 7.3):

1. Class loading: The classloader reads in a Java class file and creates an equivalent representation in memory. This internal representation preserves the original structure of the class file. In the following, we call this format the *low-level representation* of the class file.

2. Resolution: This pass uses the low-level class file representation to construct a *high-level representation* of the class file. This representation provides a more abstract view of the class file structure by resolving the constant pool.

3. Control flow analysis: This pass builds a control flow graph, which represents both "normal" and exception handling control flow, for each non-`native` method.

4. Data flow analysis: This pass uses the control flow graph constructed in the previous pass to extract type information from the bytecodes.

5. Code generation: The code generation pass translates Java bytecodes to semantically equivalent SUIF 2.0 or OSUIF code (by using the control flow graph representation of the previous pass). Code generation is done in a single pass over the bytecode array and translates each bytecode isolated (i.e., without taking the bytecodes's context into account). The compiler generates either SUIF instruction list from or CFG form (see section 4.4).

6. High-level control structure recovery: Structured programs are usually better compiled and easier to understand. The high-level control structure recovery pass tries to recover high-level control flow constructs, such as loops and conditional statements, from unstructured control flow (see also section 12.2). Reconstruction is always possible if the control flow graph is reducible [ASU86]. For irreducible graphs, a partial recovery is usually possible [Hav97]. This pass (partially) translates the code generated by the previous pass into SUIF statement list form. It has not been implemented as part of the thesis.

The first four passes constitute the analysis part of the compiler and are further explained in chapter 8; the remaining passes constitute the synthesis part, which is covered in chapter 9.

Code for a certain class can be generated independently from other classes. The compiler expects the name of a Java class as input. This class is called the *main class* in the following. `j2s` generates code for either the main class or the complete transitive closure of the main class. Refer to section 9.1 for more details.

Code for a method body can be generated independently from other methods. Our compiler simply iterates over all methods of a class and applies control flow analysis and code generation to a particular method before skipping to the next method. This way, only the dynamic data structures of a single method need be kept in memory during the compilation.

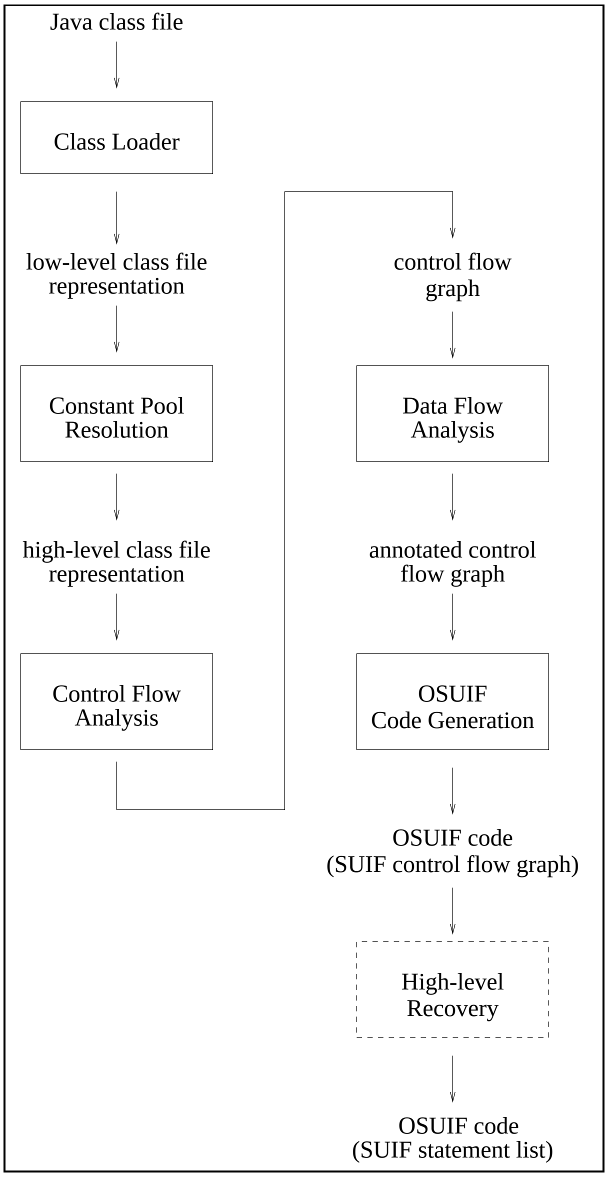

Figure 7.3: `j2s` translation process

Chapter 8

The j2s Front End

The front end of `j2s` consists of the class loader, the resolver, the control flow graph generation, and the data flow analysis to recover lost type information. Each of these passes is discussed in detail in the following sections.

8.1 Class Loader

The *class loader* reads a Java class file and stores the various components of the class file into an in-memory object hierarchy. Every component of the class file format (see section 6.5), such as fields, methods, attributes, and various different entries of the constant pool, is represented by a C++ class. The class loader is the only part of the compiler that is directly concerned with file handling and I/O.

At this stage the original structure of the class file remains unchanged. This means that, for example, the constant pool and indexes in the constant pool are stored unresolved. Furthermore, field and method descriptors (see section 6.5) are kept in the original representation as UTF-8 strings.[1]

The class loader reads in a single class file at a time. Because it does not perform any constant-pool resolution, the class loader does not need to access other class files in order to get a complete representation. For instance, assume class S in the class file `S.class` does inherit from class T in the class file `T.class`. When the class loader reads in S, it only stores the index to the constant pool entry that contains the string constant of the superclass, namely T, into the `super_class` field of the internal `ClassFile` structure—it does neither check if T exists nor looks at the actual string contents.

The class loader ensures that the class file has indeed the format of a class file as given in the JVM specification. This corresponds to the first pass of the verification process as described in section 6.6. If the class loader encounters an unknown attribute (i.e., an attribute that is not described in the JVM specification) it issues a warning and ignores the attribute.

A *dictionary* is part of the class loader, which manages multiple class files. Class files can be looked up by name and are transparently loaded in memory if they have not already been loaded. Thus, the dictionary ensures that class files are loaded at most once.

The class loader is architecture independent. This means that numerical values are handled correctly regardless whether the underlying architecture supports big- or little-endian byte ordering or IEEE 754 floating-point arithmetic[2].

[1] Actually, the JVM specification uses a slightly different format than standard UTF-8.

[2] `CONSTANT_Float` and `CONSTANT_Double` entries use the IEEE 754 bit layout.

8.1.1 Implementation

The mapping of the constant pool structures to C++ classes is straightforward. Every class file table (including the constant pool) is represented by a separate C++ class. Value entries (e.g., `access_flags`) and index entries, which refer to position in other tables (e.g., `super_class`), are kept in private variables of the corresponding C++ class. Figure 8.1 shows the private part of the class `Classfile`, which implements the `ClassFile` table.

```
class Classfile {
private:
  u4 _magic;
  u2 _major_version;
  u2 _minor_version;

  u2 _access_flags;
  u2 _this_class;
  u2 _super_class;

  // constant_pool_count is _constant_pool.count()
  DynamicArray<Constant> _constant_pool;
  DynamicArray<Interface> _interfaces;
  DynamicArray<Field> _fields;
  DynamicArray<Method> _methods;
  DynamicArray<Attribute> _attributes;
  ...
};
```

Figure 8.1: Low-level implementation of the `ClassFile` table

The various types of constant pool entries (e.g., numeric values, string constants, and representations for classes, methods, and fields) are represented by separate classes, which are derived from the abstract base class `Constant` (refer to figure 8.2). Constants for `float` and `double` are transformed to floating-point values. UTF-8 constants are kept in their original form, but are represented as C strings,[3] which saves us from storing the length of the string.

8.2 Resolver

The *resolver* uses the low-level representation of the class loader to construct a high-level view of the Java class file. The constant pool is completely eliminated by resolving all references to it (see also section 6.7).

The remaining components of the class file provide a high-level view of their contents. For example, references to a Java class are no longer symbolic names (i.e., indexes to UTF-8 strings that contain the fully qualified name of the class). Instead, classes are represented by a (unique) reference to an instance of the C++ class `Unit`, which represents a class or interface (see figure 8.3). For example, the `super_class` field of the `ClassFile` structure is no longer represented by an index into the constant pool, but by a pointer to an object of the class `Unit` that stores the corresponding Java class.

Because constant pool resolution resolves all symbolic names, the resolver ends up loading the transitive closure of all referenced classes in the class file that is currently read in. Whenever the resolver

[3]This is possible because Java UTF-8 constants do not contain '`\000`' characters.

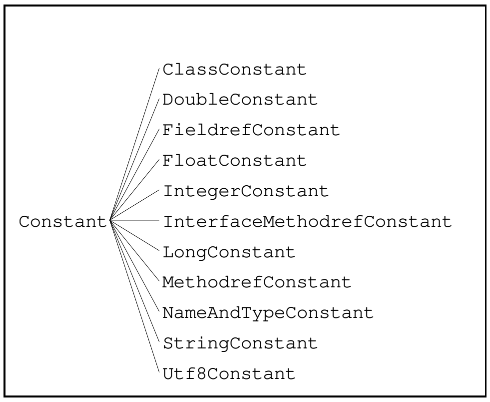

Figure 8.2: Java constant pool entries

```
class Unit {
private:
  AccessFlags* _access_flags;
  FQName* _name;
  Unit* _super_class;
  Array<Unit>* _interfaces;
  Array<Field>* _fields;
  Array<Method>* _methods;
  Name* _sourcefile;

  // low-level representation
  Classfile* _classfile;
  ...
};
```

Figure 8.3: High-level implementation of the `ClassFile` table

sees a symbolic reference to another class, it calls the class loader to load the class. If the class is already loaded, the class loader returns that loaded class. (This happens transparently to the resolver.)

The resolver keeps its own dictionary of classes that have already been resolved. The resolver ensures, with the help of this dictionary, that all loaded class files are resolved at most once. Because the dictionary guarantees that every class is only loaded once, the equality check of two classes can be simply done by comparing their references.

The resolver also implements the remaining checks for pass two of the JVM verifier (refer to section 6.6). For example, it makes sure that all field and method entries have a valid type descriptor.

8.2.1 Representation of Types

The JVM stores field and method descriptors as UTF-8 string constants (see section 6.5 for more details). When the resolver encounters a descriptor string, it parses the string and builds a *type tree*[4]. Parsing of

[4] Actually it is just a “type list.”

the descriptor strings is very easy because they can be described with a *Simple LL(1)* grammar [Plö94].

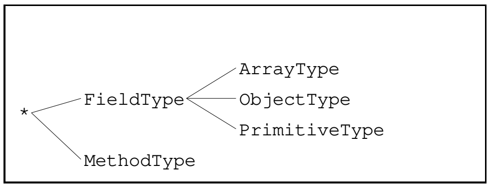

Figure 8.4: Java type hierarchy

The internal representation of types is straightforward. Figure 8.4 shows the class hierarchy for Java types. Object types and primitive types are leaf nodes in the type graph. Object types contain a reference to the Java class that they represent. Array types contain a reference to their element types. A field type is any one of an object type, an array type, or a primitive type. A method type contains a list of field types for its formal parameter types and a single field type for its return type.

Primitive types are implemented by a single class to keep the representation more compact. The `void` type is also a primitive type (even though this does not conform to the JVM specification). It is only legal to indicate that a method has no return type. Figure 8.5 shows the runtime representation of the type

```
[[Ljava/lang/Object;
```

which represents "array of array of instances of class `Object`."

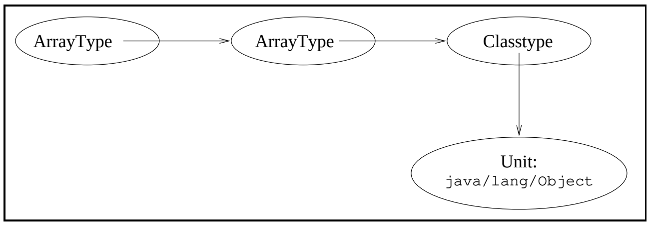

Figure 8.5: Internal representation of type `[[Ljava/lang/Object;`

Objects that represent types are not necessarily unique in our implementation. This simplifies the design, because otherwise the implementation of a type dictionary would have been necessary to guarantee uniqueness (by mapping UTF-8 strings to type objects).

8.2.2 Bytecode Iterator and Visitor

The only class file component that is kept in its original representation is the *bytecode array* (i.e., method body), but the implementation hides this to the programmer by providing a *bytecode iterator* and various *bytecode visitors*.[5] The bytecode iterator, implemented with the class `ByteCodeIterator`, traverses the bytecode array sequentially and calls for every bytecode the corresponding (virtual) method of the visitor. The *abstract visitor* class (`CodeVisitor`) defines an abstract method for every bytecode. A *default visitor*

[5]The iterator and visitor pattern are explained in [GHJV94].

(`CodeVisitorPartial`), which subclasses from this visitor, redefines all methods with an empty body as default behavior. Typically, a specific visitor would subclass from the default visitor and only redefine the methods of the bytecodes that actually have to perform an action.

Visitors are used in several places in the implementation. For example, the resolver makes use of this mechanism by subclassing a print visitor from the abstract visitor class. This class prints out a textual representation of the class file that is similar to `javap -c`.[6] Using the iterator/visitor approach yields a high-level representation of the bytecode array while preserving its original compact coding.

As an alternative approach, one can create an inheritance hierarchy of classes for the bytecodes, but this forces the implementation of more than 200 leaf classes (for concrete bytecodes) plus additional intermediate classes for abstract groups of bytecodes (e.g., branch or binary arithmetic bytecodes). This approach allows to directly attach operations to every bytecode, such as a specialized `print` method for every leaf class.

This design is appealing on first sight, because it allows to add operations for printing, type checking, control-flow analysis, code generation, etc. to every bytecode class. We refrained from this for two reasons:

- Providing all these different kinds of operation for every bytecode class is not such a good idea [GHJV94, Chapter 5]: "The problem here is that distributing all these operations across the various classes leads to a system that's hard to understand, maintain, and change. ... Moreover, adding a new operation usually requires recompiling all of these classes. It would be better if each new operation could be added separately, and the classes were independent of the operations that apply them." Our implementation uses visitor classes to factor out operations that are performed on bytecodes.
- We want to keep the original dense representation of the bytecodes. The most space in a class file is typically consumed by the constant pool (which we resolve) and by the code array. Because the resolver keeps the transitive closure of all classes in memory we are interested in a space efficient in-memory representation of class files.

8.2.3 Lazy resolution

Java bytecodes contain references to the constant pool. Because we keep the bytecodes in their original representation we cannot simply attach the resolved references to them. A separate data structure maps constant pool indexes to their resolved representation. This data structure is created when a visitor is executed for the first time for a specific bytecode array and reused by successive iterations. This means that the actual resolution is only done once and when it is actually needed. As an alternative, the resolution could have been implemented on-the-fly by the iterator, but this would have caused additional overhead whenever a visitor were executed.

8.2.4 Iterator/Visitor Example

The following code shows an example of a bytecode visitor. The `NumGotosVisitor` is derived from the default visitor class `CodeVisitorPartial`:

```
class NumGotosVisitor : public CodeVisitorPartial {
public:
  int num_all;
  int num_gotos;
```

[6] `javap` is a bytecode disassembler tool of Sun's JDK release.

```
  NumGotosVisitor() : num_all(0), num_gotos(0) { }

  virtual void preinstr(int ci)          { num_all++; }
  virtual void goto_(int ci, u2s target) { num_gotos++; }

  virtual void postvisit() {
    cout << "Percentage of goto bytecodes: "
         << ( (num_gotos * 100.0) / num_all ) << endl;
  }
};
```

The following code shows how `NumGotosVisitor` can be called with a bytecode iterator. The iterator traverse a bytecode array and calls the visitor for every bytecode.

```
Method* any_method; // Represents a Java method
Code* bytecode_array= any_method->code();

NumGotosVisitor* the_visitor= new NumGotosVisitor();

ByteCodeIterator* the_iter=
  new ByteCodeIterator(bytecode_array, the_visitor);
the_iter->iterate();
```

8.3 Control Flow Graph Generation

Control flow analysis uses the representation provided by the resolver to construct a control flow graph of a bytecode array. To construct the control flow graph two steps have to be performed:

1. Partitioning of the bytecode array into basic blocks.
2. Computation of the control flow between basic blocks.

Definition 8.1 (Basic Block) *A* basic block *is a sequence of consecutive bytecodes in which flow of control enters at the first bytecode of the sequence and every bytecode only transfers control to the next bytecode in the sequence. In other words, a basic block has one entry and one exit point.*

To compute the basic blocks, we first determine the set of *leaders* of the bytecode array [ASU86].

Definition 8.2 (Leader) *A* leader *is*

- *the first bytecode in the bytecode array.*
- *any bytecode that is the target of a* transfer instruction *(refer to section 6.8).*
- *any bytecode that immediately follows a transfer instruction.*

For each leader, the corresponding basic block contains the leader itself and all consecutive bytecodes up to but not including the next leader or the end of the bytecode array. This way the bytecode array is uniquely partitioned into a set of non-overlapping basic blocks.

Control flow at the end of a basic block either terminates the current procedure (i.e., control is transfered back to the caller) or transfers control to another basic block. Thus, the possible control flow targets of a basic block are determined by the last bytecode of that basic block.

Definition 8.3 (Basic block types) *A Basic block falls into one of the following types:*

- *1-way block: the last instruction in the basic block is an unconditional branch, i.e.,* `goto`, `goto_w`, `jsr`, *or* `jsr_w`. *The block has one out-edge, namely the basic block whose leader is the branch target.*
- *2-way block: the last instruction in the basic block is a conditional branch, i.e.,* `if*`. *The block has two out-edges. One edge goes to the following basic block, and the other edge goes to the basic block whose leader is the branch target.*
- *n-way block: the last instruction in the basic block is a compound conditional branch, i.e.,* `tableswitch` *or* `lookupswitch`. *The number of out-edges edges corresponds to the number of branch targets (including the* `default` *target).*
- *return block: the last instruction in the basic block is a* `*return` *bytecode. Control flow is transfered back to the caller at the end of the basic block. There are no out-edges from this basic block.*
- *fall-through block: the last instruction transfers control to the next bytecode, but this bytecode is the target of a branch instruction. The basic block has one out-edge, whose target is the successive instruction.*

Basic blocks and the control flow between them directly corresponds to a *control flow graph* (CFG): Nodes of the graph are basic blocks; edges are the control flow from one basic block to another. The type of a node is determined by the number of its outgoing edges and is analogous to the basic block type. The *start node* of the control flow graph is the first basic block (i.e., it contains the bytecode with index zero). Our graph representation contains no single, distinguished *exit node*.[7] It can be easily constructed, however, by introducing a single out-edge for every return block that goes to a unique exit node.

Both a 1-way block and a fall-through block have a single out-edge. For a CFG, a distinction between these two block types is not strictly necessary. This distinction, however, simplifies code generation (see section 9.2).

In general, control flow can be represented either implicitly with basic blocks or explicitly with a CFG. In fact, usually one representation can be easily constructed from the other. The `j2s` compiler implements both representations.

8.3.1 Control Flow Graph Example

Figure 8.6 shows an example of a simple Java program. We compiled the Java source with `javac` to obtain a class file for this program. The resulting control flow graph of the class file is depicted in figure 8.7. Every node of the CFG is labeled with the indexes of the bytecode array section that the node covers (e.g., `6..8`). The edges are labeled with `T`(rue) and `F`(alse) for 2-way nodes.

All CFGs shown in this thesis have been generated with the `dot` graph drawing tool [KN93]. It takes graph text files (which consist mainly of node and edge descriptions) as input and generates hierarchical graph drawings as output. The compiler is capable of generating `dot` output from its internal control flow graph representation. We simply piped the graph description that the compiler generates through `dot -Tps` to obtain the PostScript output of the graph.

8.3.2 `jsr` and `ret` Bytecodes

So far we assumed that control flow can be fully reconstructed from the (static) information in the bytecodes. This means that the targets of all control transfer instructions are statically known. This

[7] This representation is often beneficial for data flow analyses because it simplifies the graph algorithms. For example, the algorithm for SSA construction described in [CFRW91] expects an edge from the start to the exit node for every CFG.

```
class Fibonacci
{
  static int fibonacci(int i) {
    if(i==0) {
      return 0;
    } else if (i==1) {
      return 1;
    } else {
      int fib0= fibonacci(i-1);
      int fib1= fibonacci(i-2);
      return (fib0+fib1);
    }
  }

  public static void main(java.lang.String[] args) {
    int iterations= java.lang.Integer.parseInt(args[0]);

    for(int i=0 ; i<=iterations ; i++) {
      java.lang.System.out.print(Fibonacci.fibonacci(i));
      java.lang.System.out.print(" ");
    }

    java.lang.System.out.println();
  }
}
```

Figure 8.6: Source code of Fibonacci class

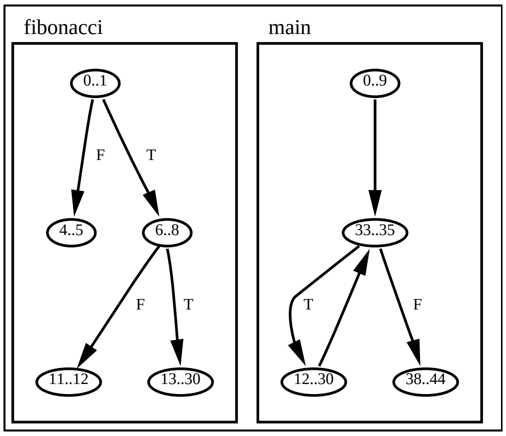

Figure 8.7: Control flow of Fibonacci methods

is usually the case indeed. An `if`-bytecode, for example, has an operand that contains the jump target (coded as relative offset into the bytecode array).

The only exception from this rule is the `ret` bytecode, which does not have a static jump target.[8] Instead, the jump target is given by a local variable, which must contain a value of type `returnAddress` (refer also to section 6.3). In this respect, `ret` is similar to an *indirect jump* instruction. The important difference is that only the `jsr` bytecode can create a `returnAddress`. Furthermore, (arithmetic) operations on `returnAddress` (i.e., operations that resemble "pointer arithmetic") are illegal. These restrictions have the useful property that all targets can be easily inferred (see below).

Figure 8.8 shows class file assembly code in *Jasmin* format (see section 10.2) that illustrates this point. We used Jasmin to generate a small toy example because `jsr/ret` bytecodes are usually only generated for a `try-finally` statement, which yields rather complex code.

The `jsr` bytecode at index 25 is reached through the `jsr` bytecode at either index 7 or index 17. Hence the `ret` bytecode transfers control flow to either index 10 or index 20, but this information is not statically (i.e., without data flow analysis) available.

Since the possible targets are not known at this point of the analysis, the edges of the control flow graph are incomplete. We have the following options to overcome this problem:

- We can approximate the targets by using a superset of the actual targets. Only `jsr` can introduce a new target. Thus, we can simply use the targets of all `jsr` bytecodes in the bytecode array to obtain a superset of the actual targets.

- We can compute the precise targets during data flow analysis.

We chose the latter for the following two reasons:

- Precise targets are necessary to check certain properties that are imposed on `jsr/ret` bytecodes (e.g., proper `jsr` call chain).

- Having precise targets rather than supersets can lead to better code (see section 9.2.12.1).

Therefore, the computation of the `ret` out-edges (that branch to the precise targets) is deferred until the next pass (see section 8.4). Figure 8.9 shows the *complete* control flow graph. The edges that are

[8]Note that `ret` has a completely different behavior than the `*return` bytecodes that terminate a method.

```
    .class public JsrRet

    .super java/lang/Object

    ;
    ; Default constructor
    ;
    .method public <init>()V
0          aload_0
1          invokenonvirtual java/lang/Object/<init>()V
4          return
    .end method

    ;
    ; public int foo(int i)
    ;
    .method public foo(I)I
          .limit stack 10
          .limit locals 10

 0        iload_1          ; get actual parameter ‘i’
 1        ifeq Li_zero     ; ‘i’ equal zero?
    Li_not_zero:
 4        ldc 42
 6        istore_2
 7        jsr Ljsr         ; jump to subroutine
10        iload_2
11        goto Lexit
    Li_zero:
14        ldc 43
16        istore_2
17        jsr Ljsr         ; jump to subroutine
20        iload_2
21        goto Lexit
    Ljsr:                  ; SUBROUTINE
                           ; (return address is on top of stack)
24        astore_3         ; store return address in slot 3
          ; do nothing
25        ret 3            ; jump back with return address in slot 3
     Lexit:
27        ireturn
    .end method
```

Figure 8.8: Jasmin assembly example for `jsr/ret` bytecodes

labeled `<7>` and `<17>` cannot be constructed in this pass as explained above. The number of ellipses drawn around a node indicate the depth of the `jsr` call chain: A `jsr` bytecode introduces a new nesting, whereas a `ret` bytecode decreases the nesting of the call chain.

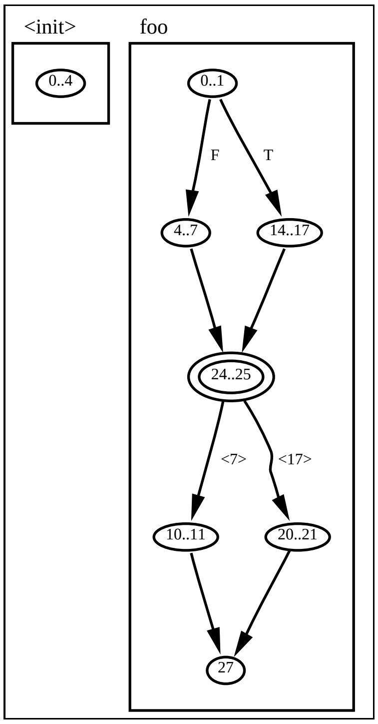

Figure 8.9: Control flow for example in figure 8.8

8.3.3 Exception Handling Control Flow

Even though the `j2s` compiler currently does not generate code to handle a thrown exception, exception handling control flow is fully represented. The CFG contains edges for "normal" and exception handling control flow (EHCF). Exception handling edges represent both explicit and implicit EHCF.

Since the control flow of the currently generated code never follows an exception handling edge, it is also possible to turn off the generation of EHCF information.

Figure 8.10 shows an example of Java source code that contains EHCF; figure 8.11 shows two CFG versions for this code. The left CFG contains both normal control flow (shown with solid arcs) and EHCF (shown with dotted arcs); the right CFG shows only normal control flow. Every node label is prefixed with the `try`-block nesting depths.

The `catch` handlers in the left CFG are nodes `0: 9..10` and `0: 13` for `div` and nodes `1: 11..14` and `0: 25..28` for `divdiv`. (They can be easily identified because only dotted arcs point to them.) Without EHCF the `catch` handlers are not reachable and subsequent analysis detects them as dead code and eliminated them (see section 8.4.3), as the right CFG shows.

As can be seen by comparing the two CFGs in figure 8.11, the original basic blocks must be split up to capture the finer-grained EHCF. Blocks are split for two reasons:

```
class TryBlock {
  public int div(int a, int b) {
    int res= 0;

    try {
      // division by zero raises ArithmeticException
      res= a/b;
    }
    catch(java.lang.ArithmeticException o) { }
    catch(java.lang.Throwable o) { }

    return res;
  }

  public int divdiv(int a, int b, int c) {
    int res= 0;

    try {
      try {
        res= a/b;
      }
      catch(java.lang.ArithmeticException o) {
        res= java.lang.Integer.MAX_VALUE;
      }
      res /= c;
    }
    catch(java.lang.ArithmeticException o) {
      res= java.lang.Integer.MAX_VALUE;
    }

    return res;
  }
}
```

Figure 8.10: Java source that contains exception handling control flow

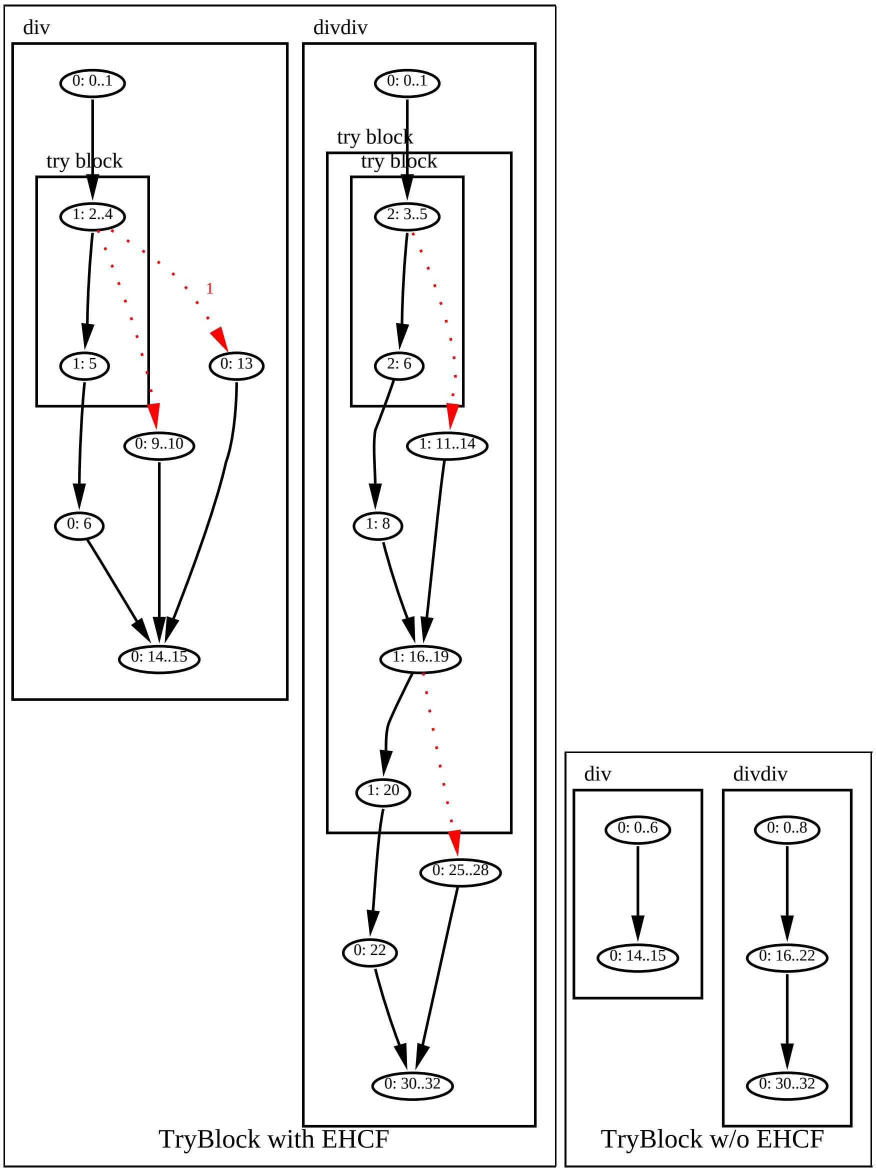

Figure 8.11: Two CFGs (with and without EHCF) for Java code in figure 8.10

- Control flow crosses a `try`-block boundary inside of a basic block.
- A bytecode that can raise an exception is not the last bytecode in the original basic block. In this case, the basic block is split after this bytecode (i.e., the bytecode is now the last bytecode in the newly created first basic block).

In both cases, the original basic block is split into two separate blocks. The first basic block is now a fall-through block that simply transfers the normal control flow to the second block. The second block has the type of the original basic block.

We split blocks at `try`-block boundaries, because this way we can easily attach `try`-block information (such as the `try`-block nesting depth) to a CFG node. (Furthermore, all the nodes that belong to a `try`-block can be made explicit in the `dot` output by drawing a bounding box around them.)

Every bytecode that can raise (or propagate) an exception must be the last bytecode in a basic block to represent the EHCF. If off-line compilation is used only *runtime exceptions* can be thrown during program execution. *Linking exceptions* are detected during compile time. `j2s` generates compile errors for them. Therefore, bytecodes that can raise only linking exceptions must not be taken into account. In the following we list all bytecodes that can propagate exceptions or raise runtime exceptions (see section 6.8 for a more detailed explanation of the bytecode types):

Arithmetic bytecodes: `idiv`, `ldiv`, `irem`, and `lrem` bytecodes throw an `ArithmeticException` if the value of the divisor is zero.

Method call: A method call raises a `NullPointerException` if the "this" reference is null. A method call can also propagate exceptions that are raised by the callee. These exceptions are either *unchecked exceptions*[9] or declared by the `throws` clause of the callee.

Object creation and manipulation: Loads and stores of array elements can raise a `NullPointerException` or `ArrayIndexOutOfBoundsException`. Additionally, `aastore` can raise an `ArrayStoreException`. `newarray` and `anewarray` can throw `NegativeArraySizeExceptions`.

Explicit exception: The `athrow` bytecode can throw any class instance that is a subclass of `Throwable`.

Type check: `checkcast` can raise a `ClassCastException`.

Synchronization: `monitorenter` can raise a `NullPointerException`; `monitorexit` can raise either a `NullPointerException` or an `IllegalMonitorStateException`.

The EHCF for the above bytecodes is made explicit by introducing new exception handling edges from a specific throw point of an exception (which is always the last bytecode of a basic block) within a `try`-block to every `catch` handler attached to the innermost `try`-block. EHCF that leaves the current method (i.e., causes unwinding of the call stack) is not explicitly represented. However, it can be easily inferred.

Alternatively, the EHCF could be represented with a single edge from the throw point to a "dispatch exception" node that in turn contains out-edges to the actual `catch` handlers. It is hard to tell which representation is better suited as long as our implementation does not generate exception handling code.

Type analysis can narrow down the set of `catch` handlers. For example, the `idiv` arithmetic bytecode can only raise an `ArithmeticException`. Hence, control flow can only branch to `catch` handlers that catch `ArithmeticException` or superclasses of `ArithmeticException`. The current implementation does not take this into account.

[9]Unchecked exception are subclassed from either `Error` or `RuntimeException` [GJS96]. All exceptions that are thrown by bytecodes are unchecked exceptions.

Visitor	Purpose
`DetectLeadersVisitor`	Constructs `BasicBlocks` and sets the index of the leader.
`OutEdgesVisitor`	Creates the out-edges and EHCF edges.
`TryBlockVisitor`	Computes the `try`-block nesting for every basic block.

Table 8.1: Bytecode Visitors

Usually only EHCF can reach a `catch` block.[10] However, this is not enforced by the Java verifier. Nonetheless, the compiler issues a warning message when it encounters this control flow because it often indicates that the class file is corrupted.

8.3.4 Implementation

A basic block is implemented with the class `BasicBlock`. In order to keep the representation compact, a basic block object does only store the indexes and not the actual bytecodes that constitute the basic block. In particular, we keep the indexes of the leader, the last bytecode, and the leader of the next basic block. A boolean flag marks basic blocks that are the beginning of a `catch` handler. An enumeration type is used to encode the type of the basic block.

Furthermore, a basic block object (which also represents a CFG node) contains the out-edges for the CFG representation. The out-edges for normal and exception handling control flow are kept in two separate arrays. We chose using arrays because CFGs have on average about two out-edges per node [Cif94]. Hence a non-sparse matrix representation would be memory inefficient (because most of the entries would be zero). We did not explicitly represent the in-edges of a CFG node. The CFG representation is only used by the data flow analysis pass (see next section), which does not need in-edges for efficient execution because it is a forward-flow problem.[11] It is important to realize that `j2s`'s CFG is not meant to be the central data structure in the compilation process. Instead, we want to generate SUIF code as early as possible in the compilation process to take advantage of SUIF during the remaining part of the compilation.

Every node contains its `try`-block nesting depth and a list with all the enclosing `try`-blocks (ordered by their nesting relationship). The compiler issues warnings for try-blocks that are not properly nested.[12])

We made heavy use of bytecode visitors (which have been introduced in section 8.2.2). An abstraction to iterate over basic blocks is implemented with the `BBIterator` class. This basic block iterator creates and executes a separate bytecode iterator for every basic block. (Bytecode iterators can iterate over either the whole bytecode array or just a certain consecutive part of it.) A bytecode visitor can access the basic block that the current bytecode belongs to.

Table 8.1 shows a list of the bytecode visitors that have been used for this pass. All visitors—except for `DetectLeadersVisitor`, which is executed by a bytecode iterator—are executed with a basic block iterator.

[10] Code generated by javac always fulfills this property.

[11] Graph algorithms that generate a depth-first spanning tree and check for reducibility (see section 8.4.4) also operate on the CFG, but it is planned to rewrite them as generic SUIF passes.

[12] The `javac` compiler generates always code with properly nested handlers.

8.4 Data Flow Analysis

Data-flow analysis is needed to recovery lost type information in the class files and to compute precise jump targets for the `ret` bytecode. This pass also removes dead basic blocks and checks if the control flow is reducible. We discuss each of these analyses in the following sections.

8.4.1 Type Recovery

In a class file all type information of local variables and operand stack entries is lost. However, this type information is needed because we want a type-safe intermediate representation of the generated SUIF code.

Lost type information can be recovered with a specific data flow analysis, called *type recovery* in the following.[13] This is possible because every bytecode implicitly specifies the types of the operand stack values and local variable slots that it manipulates (see section 6.8).

Type recovery operates upon the CFG that has been constructed by the previous pass. Conceptually, every bytecode is associated with data-flow information that is created or modified during the traversal of the CFG until a stable point is reached (i.e., further traversal of the CFG does not result in any modification of the data-flow information).

Our implementation uses a *two-level CFG*, which is the most common representation. Each node in a two-level CFG is a basic block, as opposed to a single instruction. This coding reduces the size of the CFG representation and data-flow information, but obscures data-flow analysis by introducing both a local analysis phase and a propagation phase [TH92]. Fortunately, our analysis can be easily separated into these two phases:

Local propagation: Data-flow information is propagated from the beginning to the end of a basic block by sequential *symbolic execution* of every bytecode.

Global propagation: Data-flow information is propagated from a basic block to all successor blocks (called *merging* in the following).

Type recovery uses its own type model (which is similar to the one specified for the JVM):

$$\begin{aligned}
\mathit{PrimType} &= \{\texttt{int},\ \texttt{float},\ \texttt{returnAddress},\ \texttt{long_high},\ \texttt{long_low},\\
&\qquad \texttt{double_high},\ \texttt{double_low}\}\\
\mathit{ClassType} &= \{x|x\ \mathit{is\ a\ class\ type}\}\\
\mathit{ArrayType} &= \{x|x\ \mathit{is\ an\ array\ type}\}\\
\mathit{IfaceType} &= \{x|x\ \mathit{is\ an\ interface\ type}\}\\
\mathit{RefType} &= \mathit{ClassType} \cup \mathit{ArrayType} \cup \mathit{IfaceType}\\
\\
\mathit{StackType} &= \mathit{PrimType} \cup \mathit{RefType}\\
\mathit{VarType} &= \mathit{PrimType} \cup \{\texttt{invalid}\} \cup \mathit{ArrayType} \cup \mathit{IfaceType}
\end{aligned}$$

Every operand stack entry or local variable contains a type in *StackType* or *VarType*, respectively. Note that local variables cannot contain interface types because there is no bytecode that allows to store an interface type into a local variable slot. Operand stack entries have no `invalid` type because every bytecode pushes a well defined type on the stack.

To describe the type recover, we first introduce the following properties:

[13] A similar analysis must be implemented as part of the Java bytecode verifier.

Definition 8.4 (Configuration) *The configuration of a local variable array or operand stack is defined as the size of the local variable array or operand stack and the type of every local variable slot ($\in$ VarType) or operand stack entry ($\in$ StackType), respectively.*

The local variable array and operand stack configuration right before a bytecode at a certain program point is executed, is defined as the configuration of the bytecode.

The start configuration of a basic block is identical with the configuration of the leader of the basic block. The end configuration of a basic block is defined as the configuration after *the last bytecode of the basic block.*

The start configuration of a method is identical with the start configuration of the first basic block.

The data-flow information for type recovery is the bytecode configuration. The analysis keeps for every program point a mapping from every local variable and operand stack entry to the type that they hold.

8.4.1.1 Symbolic Execution

If the configuration of a bytecode is known, we can easily compute the configuration of the successive bytecode by symbolic execution. This means that if we know the start configuration of a basic block, we can compute its end configuration by symbolic execution of the bytecodes in this block. Thus, propagating data-flow information through basic blocks is straightforward.

The configuration of the operand stack after the execution of a bytecode can be computed by applying the stack effect of the bytecode. For instance, the bytecode `iadd` has the following stack effect (see section 6.8 for an explanation of the stack effect notation):

$$a_{\text{int}} b_{\text{int}} \to c_{\text{int}}, \text{ where } \quad c := a + b.$$

This means that the two topmost elements must be of type `int`. Other types are illegal and would cause the analysis to fail. The configuration of the stack after the symbolic execution of the `iadd` bytecode is obtained by popping the topmost element from the previous configuration.

The configuration of the local variable array is modified if

- a $T\texttt{store}_{index}$ bytecode stores a value at *index*.
- a $T\texttt{astore}_{index}$ bytecode stores an array element at *index*.
- the $\texttt{iinc}_{index}$ bytecode increments a local variable at *index*.

For example, $\texttt{lstore}_{index}$ changes the configuration at $index$ and $index + 1$:[14]

$$\begin{aligned} lv[index] &\leftarrow \texttt{long\_high} \\ lv[index+1] &\leftarrow \texttt{long\_low} \end{aligned}$$

For every bytecode both the stack effect and the local variable modifications are (informally) described in the JVM Specification [LY96, Chapter 6]. Type analysis fails if

- the configuration of the operand stack does not conform to the stack effect of the configuration.
- the stack grows beyond the given limit (specified for every method in the class file).[15]
- the configuration of the local variable array does not contain the proper type if an element is accessed.
- an access to the local variable array is out-of-bounds.

Note that if the type analysis fails, the bytecode verifier would have failed as well (i.e., the bytecode sequence is illegal).

[14] 64-bit values are stored into two consecutive 32-bit words.

[15] Actually, `j2s` generates only a warning because it is still possible to generate valid SUIF code.

8.4.1.2 Merging

If control flow from two basic block *merges* (i.e., more than one edge reaches a basic block), then all the end configurations of the *incoming basic blocks* have to be merged (see figure 8.12). The result is the start configuration of the *target basic block*.

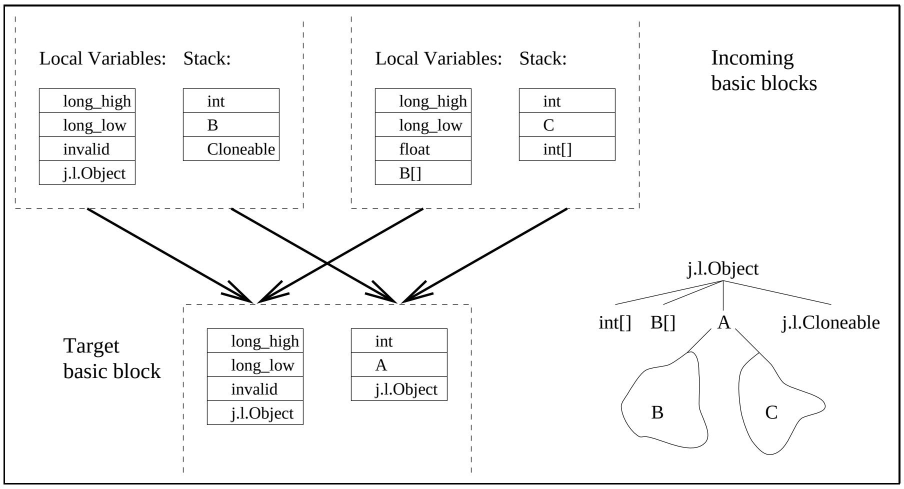

Figure 8.12: Merging Example

Merging of two reference types is described with the $\uparrow$ merge function:

$\uparrow$: *RefType* $\times$ *RefType* $\rightarrow$ *RefType*		
(t_1, t_2)	$\rightarrow$ closest common superclass of t_1 and t_2	$t_1, t_2 \in \mathit{ClassType} \cup \mathit{ArrayType}$
(i_1, i_2)	$\rightarrow$ closest common superinterface of i_1 and i_2	$i_1, i_2 \in \mathit{IfaceType}$
(i, t)	$\rightarrow$ `java.lang.Object`	$i \in \mathit{IfaceType}$
(t, i)	$\rightarrow$ `java.lang.Object`	$\wedge\ t \in \mathit{ClassType} \cup \mathit{ArrayType}$

The $\uparrow$ merge function returns the first "common" superclass or superinterface of the two merge types.[16] Merging of two references never fails because a common supertype always exists.

Merging of two operand stack entries or local variables can be described with the following two merge

[16] During merging, every interface type can be considered to be a subtype of `java.lang.Object`.

functions:

$$
\begin{array}{rll}
\multicolumn{3}{l}{\mathit{merge\_stack\_entry} : \mathit{StackType} \times \mathit{StackType} \to \mathit{StackType}} \\
\hline
(p, p) & \to p & p \in \mathit{PrimType} \\
(r_1, r_2) & \to r_1 \uparrow r_2 & r_1, r_2 \in \mathit{RefType}
\end{array}
$$

$$
\begin{array}{rll}
\multicolumn{3}{l}{\mathit{merge\_variable} : \mathit{VarType} \times \mathit{VarType} \to \mathit{VarType}} \\
\hline
(p, p) & \to p & p \in \mathit{PrimType} \\
(p_1, p_2) & \to \texttt{invalid} & p_1 \neq p_2 \\
 & & \wedge\ p_1, p_2 \in \mathit{PrimType} \\
(\texttt{invalid}, v) & \to \texttt{invalid} & v \in \mathit{VarType} \\
(v, \texttt{invalid}) & \to \texttt{invalid} & v \in \mathit{VarType} \\
(r_1, r_2) & \to r_1 \uparrow r_2 & r_1, r_2 \in \mathit{RefType}
\end{array}
$$

The first argument of the merge function is from an incoming basic block, whereas the second argument is the target block. If the merge arguments are not defined in the function table above, the arguments cannot be applied to the merge function and merging of the local variables or stack entries fails. Merging of an incoming block with a target block is successful if

- both stacks have the same height;
- all corresponding stack entries can be merged with the *merge_stack_entry* function;
- all corresponding local variable pairs can be merged with the *merge_variable* function.

If there is more than one incoming block then the incoming blocks can be merged in random order with the target block. Modifying the merge order of the incoming blocks does not change the result of the merge.

Analogous to symbolic execution, if the merging fails, the bytecode verifier would have failed as well.

8.4.1.3 Putting it all Together

Conceptually, type recovery symbolically executes all execution paths of a program and records the configuration of every bytecode. If the new configuration differs from the old one then both configuration are merged and the result is the new configuration of the bytecode. Thus, type recovery can be implemented as an iterative forward-flow problem.

The first basic block is initialized with the start configuration of the method. The stack is empty and the local variables are initialized with the formal argument types. For instance, the method `C.foo(int i, double d)` has the following start configuration for its local variables:

$$
\begin{array}{lll}
lv[0] & \leftarrow \texttt{C} & (\text{“this” reference}) \\
lv[1] & \leftarrow \texttt{int} & \\
lv[2] & \leftarrow \texttt{double\_high} & \\
lv[3] & \leftarrow \texttt{double\_low} & \\
lv[4] & \leftarrow \texttt{invalid} & \\
\ldots & &
\end{array}
$$

Furthermore, the start configuration of the stack for basic blocks that are the beginning of a `catch` handler is initialized to the type of the exception object. If the target block is a `catch` handler, only the local variables are merged and the stack is left unmodified.

Merging is complicated by `jsr/ret` bytecodes, which weakens the Gosling property (see section 6.6). Any local variable that is neither read nor written in a `jsr` subroutine retains its type across a `jsr` to that subroutine [LY96, Section 4.9.6] [ADM98]. To handle this case, merging of the local variable configuration of a `ret` bytecode with the jump target (i.e., the bytecode following a `jsr` bytecode) simply uses the local variable configuration of the `jsr` bytecode. Thus, the merging pretends that the subroutine call did not modify the local variable configuration. Our method only implements a subset of the checks that is required by the verification process, but a program that passes the verifier is analyzed correctly.

The iteration of the data flow analysis is performed with a simple work-list algorithms. The work-list contains the basic blocks that still have to be merged and is initialized with the first basic block. The algorithms is described in more detail in section 8.4.5.

8.4.2 Computation of `ret` Targets

As already explained in section 8.3.2, data-flow analysis is necessary to obtain the precise targets of a `ret` bytecode. Precise targets can be computed with a simple forward-flow analysis. The data-flow information that is (conceptually) associated with every bytecode is a set of `ret` targets. A ret target indicates the bytecode that control flow can branch to if the next bytecode was a `ret` bytecode. The `ret` target set only changes for `jsr` and `ret` bytecodes and hence never changes inside of a basic block. All other bytecodes propagate the set unmodified.

The JVM specification states that `jsr` calls must be *statically nested*, i.e., the bytecode following a `jsr` bytecode may be returned to only by a single `ret`. Furthermore, `jsr` calls must not be recursive. This means that every basic block has a fixed `jsr` call chain depth that can be statically determined.[17] These restrictions greatly simplify the computation of the `ret` target set.

index	bytecode	`ret` targets	nesting depth
a:	`jsr` *target*	X	n
b:	...	X	n
	...		
target:	...	$Y \cup \{b\}$	$n+1$
	`ret` (to one of $Y \cup \{b\}$)	–	$n+1$

Figure 8.13: Computation of the `ret` target set

Figure 8.13 illustrates the propagation of the data flow for an unnested `jsr` call. If the analysis encounters a `jsr` bytecode (label a), the `ret` target (i.e., the bytecode following the `jsr` bytecode, label b) is added to the set associated with the basic block that happens to be the `jsr` target (set Y). Furthermore, the set is propagated unmodified to the consecutive bytecode (label b). A `ret` bytecode simply does not propagate the set (because this has been already done by the `jsr` bytecode).[18] Note that this analysis also computes precise targets for nested `jsr` calls.

8.4.3 Dead Basic Block Elimination

Definition 8.5 (Dead Basic Block) *A bytecode is dead code if control flow that starts from the root of the control flow graph can not reach the bytecode.*

A basic block is dead code if the leader is dead code.

The type recovery analysis marks all basic blocks that have been touched during the analysis. If a basic block has not been touched, control flow starting at the first basic block cannot reach the leader of

[17] `j2s` generates an error if the `jsr` calls are not properly nested.

[18] This works because of the JVM restrictions for `jsr`/`ret` bytecodes described above.

this particular block. In this case, the block is eliminated from the array of basic blocks to ensure that no code is generated for it.

Because `j2s` currently does not generate code for exception handling control flow, all exception handlers (except for code in `finally` clauses) are unreachable and hence detected as dead code. Furthermore, Java compilers can generate dead code. For example, the JDK1.2beta2 `javac` compiler generates dead code for the following code:

```
class DeadCode {
  void empty() { }

  public int dead_code() {
    try {
      return 42; // Cannot throw an exception!
    }
    finally { empty(); }
  }
}
```

`javac` generates code for an exception handler, which can never be called.

8.4.4 Reducibility of Control Flow

Control transfer instructions in the JVM can (potentially) branch to an arbitrary target bytecode. Because these bytecodes do not adhere to any nesting rules, the resulting control flow graph can be *irreducible* [ASU86]. Structured control flow (e.g., exclusive use of `if-then-else`, `do-while`, and `break` statements) produces programs whose flow graphs are always reducible. This means that compilation of Java source usually yields reducible control flow (but it is likely that this does not hold for a highly optimizing compiler).[19]

Our compiler implements a graph algorithm that checks if control flow is reducible and outputs a warning if this is not the case. To apply this algorithm we first have to generate the *depth-first spanning tree* (DFST) and the *dominator tree* of the control flow graph. The DFST algorithm is described in detail in [ASU86, Algorithm 10.14] and [Cif93].

Currently, the implementation of the dominator tree is very inefficient and the DFST algorithm is only applied to "normal" control flow (i.e., exception handling control flow is ignored).

A DFST allows us to categorize the edges of the original CFG into

- retreating edges (r-e)
- advancing edges (a-e)
- cross edges (c-e)

The edges in the CFG that are part of the DFST are always advancing edges (but not vice versa).

Figure 8.14 depicts a CFG with an embedded DFST. Every node is prefixed with the *depth-first ordering* number. Edges that are part of the DFST are drawn with solid args.
The following definition is used for the reducibility check.

Definition 8.6 (Reducibility) *If not every retreating edge is a back edge in the CFG then the CFG is not reducible.*

[19] The `javac` compiler always generates reducible control flow.

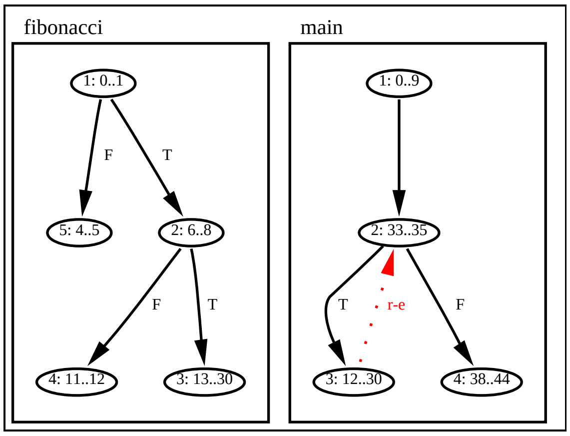

Figure 8.14: DFST for Java code in figure 8.6

A *back edge* is defined in [ASU86, Section 10.4] and not repeated here.

Figure 8.15 shows an example of an irreducible CFG.[20] The retreating edge in the graph is not a back edge because the head of the loop (node `2: 5..5`) does not dominate the tail of the loop (node `3: 8..8`) because control flow can reach the tail through `1: 0..2` → `3: 8..8`.

```
     .class public Irreducible
     .super java/lang/Object

     .method public irreducible()V
     L1:
0      ldc 42
2      ifeq L3
     L2:
5      goto L3
     L3:
8      goto L2
     .end method
```

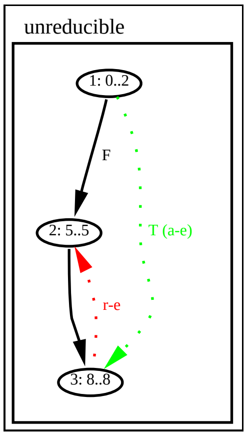

Figure 8.15: Irreducible control flow (Jasmine source and DFST)

It is planned to rewrite the code to use the SUIF CFG representation instead of `j2s`'s internal CFG. The reducibility check could be implemented more efficiently as part of a high-level control structure recovery pass.

[20] In fact, this graph is the *canonical irreducible graph* [Cif94].

8.4.5 Implementation

The abstract class `Word` represents a 32-bit slot of the JVM. It is subclassed by `LocalVariable` and `StackValue` to model a local variable and a stack entry, respectively. The local variable array is represented with the `LocalVariables` class, which implements type-checked loads and stores of array slots. The class `OperandStack` implements type-checked pops and pushes of stack entries. All of the above classes define a binary method `merge()` that performs the merging. `merge()` aborts with an error if merging cannot be performed and returns a boolean value to indicate if merging did modify the configuration of the target block.

Symbolic execution as described in section 8.4.1.1 is implemented with the bytecode visitor `MergeVisitor`. The following code shows how symbolic execution manipulates the bytecode configuration for `iadd` and `lstore`:

```
class MergeVisitor : public CodeVisitor {
private:
  OperandStack* _stack;
  LocalVariables* _lvs;
  ...
public:
  virtual void iadd(jhl_u4 code_index) {
    _stack->ipop();
    _stack->ipop();
    _stack->ipush(code_index);
  }
  ...
  virtual void lstore(jhl_u4, jhl_u2s index, boolean wide) {
    _lvs->lstore(index);
    _stack->lpop();
  }
  ...
};
```

Merging (see section 8.4.1.2) is implemented with the basic block iterator `MergeIterator`. The iterator initializes the configuration of the first basic block and the `catch` handler block and pushes the first basic block on the work-list.

The iterator uses a work-list for the type analysis that contains the the basic blocks that still have to be processed. Every iteration pops a basic block (which is the incoming block) from the work-list and calls the `MergeVisitor` for this block to perform symbolic execution. The visitor returns the end configuration of the incoming block to the iterator. The iterator merges the end configuration with the target basic blocks (indicated by the out-edges of the incoming block) and pushes modified target blocks on the work-list (if they are not already present in the list). The iterator terminates if the work-list is empty. The iterator is guaranteed to terminate because merging either fails or converges towards the closest common supertype of two slots (which is bounded by `java.lang.Object`).

The `ret` targets (see section 8.4.2) are computed as part of the type recovery analysis. We could have computed the `ret` targets separately (as an independent pass), but it seemed simple enough to unify both analyses (especially since both are forward-flow problems). Every basic block (`BasicBlock`) contains the (static) `jsr` nesting depth and the set of `ret` targets. The set is implemented as a list that contains the indexes of the `ret` target bytecodes.

The graph algorithms for the reducibility check (see section 8.4.4) require additional fields in basic blocks. A block keeps a boolean array to tell whether an out-edge is part of the DFST and another array that keeps the type of the out-edge (advancing, retreating, or cross edge)

Finally, a basic block contains a boolean marker that can be used for various purposes. Type recovery sets the boolean marker for every basic block that is "touched" during the analysis. This information is used to eliminate dead basic blocks (see section 8.4.3). The marker is also useful for debugging graph algorithms.

Chapter 9

The j2s Back End

The back end consists of the code generation and various components that are needed to generate the final executable. `j2s` supports both separate compilation and full compilation to built an executable. We discuss both (O)SUIF code generation and the compilation models in the following.

9.1 Compilation Models

The compilation process builds several object files that are linked together by the standard link editor of the operating system (e.g., `ld` on Unix systems). The link step includes

Compiled Java code: `j2s` generates *(O)SUIF files* that contain the compiled Java code. The SUIF system is used to generate object files from these SUIF files.

Runtime system: The runtime system (see section 9.4) is built as a separate library. It has been designed and implemented by Andrew Rutz as part of his Master's thesis [Rut98].

System startup: The generated code contains the `main()` function, which initializes the system and calls the actual entry point of the executable given by the Java `main()` method of the main class (see section 9.5).

Global data: The global data for an executable is gathered in a single SUIF file. Compilation of the main class X generates an object file X`_meta_data.o` that contains

- an array of pointers to class meta-data, which describes the transitive closure of X (see section 9.3.4).
- the array meta-data (see section 9.3.4).
- an array of integral numbers that contains the `left` and `right` values for efficient subclass type tests (see section 9.2.14).

Standard C++/C libraries: The runtime system is written in C++ and hence contains calls to C++ and C libraries.[1]

[1] The runtime system does not use C++ features that incur runtime overhead, such as dynamic function calls or exception handling.

Java libraries: Calls to native methods that are not intercepted by the runtime system use the JDK implementation. We can freely exchange native methods calls that are implemented by the JDK or by our compiler because both use the JNI interface. In fact, several JDK functions call back to the runtime system via JNI.

References from one object file to another are implemented with global symbols. Global symbols in SUIF are inserted into the global external symbol table and correspond to `external` declarations in C. They are resolved at the link step when the executable is built.

Generated code for a certain class usually contains method calls and field accesses to other classes. In the following, we call the class we want to compile the *main class* and all other classes that this class references (i.e., the transitive closure of the main class) its *auxiliary classes*.

`j2s` supports separate compilation as well as full compilation of class files. We discuss both models in the following.

9.1.1 Separate Compilation

Separate compilation generates a separate SUIF file for every class. Such a SUIF file has a file set block that contains

- a single file block for the main class. This file block has
  - a definition block that contains
    - procedure definitions for the method code (see section 9.2.3).
    - variable definitions for the compile-time initialization values of static fields (see section 9.2.4).
    - the compile-time meta-data (see section 9.3).
  - a file block symbol table that contains variable symbols that are only referenced in this file block, e.g., variables that hold compile-time strings, the virtual function table, the method and field meta-data, etc.
- a global external symbol table that contains
  - all types, especially procedure types and group types or (incomplete) class types for the object layout, static fields, and meta-data layout.
  - procedure symbols of the methods that the main class implements (i.e., function definitions).
  - procedure symbols that the main class uses (i.e., function declarations) from auxiliary classes.
  - variable symbols for static fields and class meta-data.
- an empty file set block symbol table (which is normally used to share symbols between different file blocks).

Separate compilation implies that the link step has to include all the object files that constitute the transitive closure of the main class.

Our separate compilation strategy is similar to Harissa's [MMBC97], but has been developed independently. The authors of Harissa observe that "[Harissa] also supports separate compilation to reduce compilation time and the size of generated code. ...Since separate compilation conflicts with class hierarchy analysis and method call optimization, it has not been used in our benchmarks." `j2s` can combine the benefits of separate compilation with whole-program analysis, because the (O)SUIF system will offer a linker, which allows to combine separate SUIF files into a single file.[2]

[2]As of this writing, the SUIF 2.0 and OSUIF linker have not been implemented. It would be possible, however, to use the SUIF 1.0 linker on converted code.

Java requires a *dependency analysis* for the recompilation of Java classes. This analysis is quite complex if one wants to avoid unnecessary recompilation. Our compiler environment performs only a date check[3]. If an object file is older than the corresponding class file, the class is recompiled.

9.1.2 Full Compilation

Full compilation is an alternative compilation model, which generates a *single* SUIF file by compiling the whole transitive closure of the main class into this file.

We chose to implement an additional compilation model, because each one has certain benefits compared to the other. Separate compilation has much faster turnaround times, because usually most files of the transitive closure of the main class are already compiled.

For full compilation the code for every Java class is compiled into a separate file block. This implies that the changes in any one of the class files that constitute the transitive closure forces a recompilation of the whole transitive closure. On the other hand, having a single SUIF file is easier to maintain and simplifies the link step. Furthermore, intra-procedural (O)SUIF optimization passes require a single SUIF file, which cannot directly be generated with separate compilation. Until the (O)SUIF linker is available, full compilation can be used as a work around.[4]

For full compilation, certain symbols are shared only across file blocks (as opposed to separate compilation, which has to share symbols across SUIF files). Hence, these symbols can be put into the file set block symbol table (as opposed to the global external symbol table). In general, the scope of certain symbols can be made more restrictive for full compilation compared to separate compilation, but this is not strictly necessary. It is perfectly legal to put all symbols into the global external symbol table (as long as the symbol names are globally unique). In the following, we only describe where the code generation places the symbols for separate compilation.

9.2 (O)SUIF Code Generation

The code generation makes use of the CFG of the control flow analysis and the data flow information gathered by the data flow analysis. As mentioned before, the philosophy of the code generation is to generate simple, straightforward (O)SUIF code. Once (O)SUIF code has been generated we can improve the code with

- generic OSUIF optimization passes (e.g., class hierarchy analysis). Furthermore, Java-specific optimization passes can be written as well.
- SUIF 2.0 and SUIF 1.0 optimization passes. We currently convert the generated code to SUIF 1.0 and run `porky1` optimization passes that perform the removal of unused symbols and types, copy propagation, constant propagation, simple dead code elimination, and simple optimizations on unstructured control flow.[5]

We use C code in this chapter when we give examples of code that has been generated by `j2s`. This C code was obtained by first converting SUIF 2.0 to SUIF 1.0 with `do_convertsuif2to1` and then using `s2c` to obtain the C code from the SUIF 1.0 representation. Naturally, the C code does not reflect the original SUIF 2.0 constructs very well, but it serves as an easy to read and understand description of

[3]This is similar to the default behavior of the the JDK1.1 `javac` compiler, which only recompiles "missing or out-of-data class files that are referred to from source code."

[4]It turned out that full compilation is not feasible for program development because the turnaround times are too long (about two to eight hours). Separate compilation takes typically 5 to 15 minutes.

[5]These optimizations correspond to the following `porky1` switches: `-unused-syms -unused-types -copy-prop -const-prop -dead-code -ucf-opt`.

the semantics of the SUIF 2.0 code.[6] The reader should keep in mind, however, that SUIF 2.0 files can directly be translated to object files (e.g., in ELF format) rather than C code as soon as a SUIF 2.0 back end pass becomes available.

The code generation tries to preserve the class file information in the (O)SUIF code as far as that is possible. For example, our previous analysis of the `jsr/ret` bytecodes allows the code generation to emit code with precise `ret` targets. `j2s` goes through the trouble to determine the precise targets because it would be hard to recover this information even in a Java-specific SUIF pass. On the other hand, the code generation does not try, for example, to reuse temporary variables or performs strength reduction because these optimizations are easily handled by standard SUIF (back end) passes. In general, we refrained from implementing an optimization in `j2s` if it can be easily realized with a SUIF pass.

Code generation translates one bytecode at a time into a sequence of SUIF instructions, without taking the context of the bytecode into account. Conceptually, the compiler implements a SUIF code template for every bytecode. Code generation preserves the original control flow structure of the bytecode array, i.e., the generated code has exactly the same basic block types as the original code. This simplifies code generation because branch instructions must not be modified and jump instruction must not be added or deleted.

Our implementation keeps the list of basic blocks that constitute a bytecode array ordered by the leader indices. Hence code generation can simply traverse the list of basic blocks and generate code for every basic block independently. Generating code for a basic block consists of the following steps:

1. Generate a `label_location_instruction`. The label name is uniquely mangled with the leader index of the block. Note that the label is not necessarily used as a branch target, but a successive SUIF pass can easily remove unused labels.[7]

2. If the configuration of the operand stack of the leader bytecode is not empty, code is generated to copy the spilled stack contents back on the stack. (See section 9.2.9.2 for a detailed description.)

3. The code generation iterates sequentially through the basic block and emits code for every bytecode.

4. If the configuration of the operand stack after the last bytecode is not empty, the stack contents is spilled. (See section 9.2.9.2 for a detailed description.)

If plain SUIF code is generated, `j2s` has to choose a specific implementation strategy for bytecodes that perform object-oriented behavior, whereas OSUIF code allows to express these bytecodes at a higher level of abstraction. For instance, this is the case for method calls with dynamic binding.

In the following sections, we discuss both SUIF and OSUIF code generation for every Java bytecode. If OSUIF code generation is not mentioned explicitly then the same code is generated for both OSUIF and SUIF.

9.2.1 Classes

Representation of Java classes differ significantly for SUIF and OSUIF code generation. SUIF code usually has to flatten the ownership structure that classes impose upon their members with global symbols, whereas OSUIF code preserves this information in a `class_type` (see section 5.4). In general, SUIF code has to mangle symbol names for class members to globally unique names and insert them into the global external symbol table for separate compilation, whereas OSUIF code can insert these symbols into symbol tables of the class type without name mangling.

[6] It is possible to generate a textual representation of SUIF 2.0 code with the `printsuif` application, but it is fairly difficult to read and understand. We did not want to burden the reader with it.

[7] We currently use `porky1 -ucf-opt` to remove unused labels.

OSUIF code has a class type for every Java class, array, or interface. Inheritance links are used to model Java's subclass inheritance. SUIF code uses a `group_type` to model the object layout of Java classes and arrays. Interfaces are not directly represented in SUIF code because they contain only static members, which are coded as global symbols.

Code generation for class member is discussed in the following sections.

9.2.1.1 Forward Declarations

If the main class need not know the object layout of an auxiliary class (e.g., the objects of the class are only used in method calls), a forward declaration is sufficient to describe this class. Forward declarations result in smaller files, especially for OSUIF files.

SUIF allows forward declarations of group types with a boolean flag (see section 4.5). Such a forward declaration must not contain any field symbols. OSUIF offers incomplete class types, which is a more fine-grained scheme. It is explained in detail in section 5.4. `j2s` uses it for the OSUIF code generation.

9.2.2 Arrays

In Java, arrays are an implicit part of the programming language (such as in C or Ada). This is in contrast to other languages (e.g., Eiffel) that have explicit arrays. A Java array extends `java.lang.Object` and implements the interfaces `java.lang.Cloneable` and `java.io.Serializable`[8]. Code generation has to take these features into account, e.g., for subtype checks and the virtual function table layout.

Because arrays are implicit, the JVM has special bytecodes for arrays to get the length of an array,[9] create arrays and load/store array elements. We discuss array

- instance layout in section 9.2.5.1.
- loads and stores in section 9.2.7.3.
- method calls and vtable layout in section 9.2.8.3.1.
- bound checks in section 9.2.16.

9.2.3 Methods

Both static and instance methods are represented with a `procedure_definition` and a `procedure_symbol` (SUIF, see section 4.4) or a `method_symbol` (OSUIF, see section 5.6). The procedure symbol contains the method name and a reference to the procedure definition. The procedure definition contains the formal parameters, the return type (if any), and the method code.

The method name is mangled to a unique name that contains the class name and method signature. The formal parameters of a method are identical with the start configuration of the first basic block's local variables (see section 8.4.1.3). This means that instance methods initialize the local variable at index zero with the "this" reference (which is passed as the first formal parameter). The method code is either instruction list form or CFG form[10].

Both SUIF and OSUIF code use a procedure definition to represent a Java method. SUIF code generation puts the procedure definition into the definition block of the file set of the corresponding class. OSUIF code generation attaches it to the definition block of the class type of the corresponding

[8] This interface has been added in JDK1.1 and is not documented in the Java Language Specification [GJS96].

[9] The array length is not part of the array type; it is initialized at creation time of the array.

[10] `j2s` is able to generate CFG form, but a SUIF I/O bug currently prevents us from writing it out into a SUIF file.

class. The attributes of the OSUIF `method_symbol` represent method access and modifier flags. Furthermore, instance methods that must not be dispatched (see section 9.2.8.3.1) are augmented with a "`not dispatched`" attribute, which is used by the OSUIF virtual function table construction pass.

For separate compilation, the procedure symbol is inserted into the global external symbol table to make the method accessible from other SUIF files.

9.2.3.1 Method Types

The type of a method is represented as a `procedure_type` or `method_type` (see section 5.6) for SUIF or OSUIF code, respectively. Unfortunately, types, unlike symbols, cannot be given a name in SUIF. Hence, it is not possible to look up types by name in a symbol table. It is possible, however, to search for a type that is structurally equivalent to a given type, but this means that we first have to construct the type to find out if it is already present in the symbol table. Our code generation implements a faster scheme by using a dictionary that maps signature strings to SUIF types.[11] Reusing types yields smaller SUIF files. (We experienced a reduction of up to 30%.)

9.2.3.2 Declarations vs. Definitions

A procedure (or method) symbol that has no procedure definition corresponds to a function declaration in C. Separate compilation generates such a function declaration for method calls to auxiliary classes. Note that declarations are only generated for methods that are actually called by the main class.

A procedure (or method) symbol that has a procedure definition corresponds to a C function definition. For separate compilation, definitions are generated only for the functions of the main class; declarations are generated only for functions of auxiliary classes. Hence, full compilation always generates function definitions for non-native Java methods.

9.2.3.3 Native Methods

Native methods (i.e., methods that are implemented with external C functions) need special treatment. Symbols that represent external C functions are inserted into the global external symbol table. Our compiler supports the full JNI specification [JNI97]. To support JNI for native method calls, `j2s` has to

- mangle method names according to the JNI name mangling scheme.[12]
- pass the right method arguments, e.g., the first argument is always the JNI environment pointer followed by the "this" pointer. The "this" pointer is required for both static and instance methods; static methods pass a reference to their Java class.
- use the C calling convention.

The runtime system must build the call stack according to the C calling convention for a native method.[13] The implementation faces the following difficulties:

- From `j2s`'s perspective, the calling of a native method must behave like a non-native call because a non-native method can be redefined with a native one and vice versa.

[11] Note that the signature string has to contain the "this" reference explicitly (which is not the case for a Java method descriptor) because otherwise static and instance methods would wrongly share the same type.

[12] The same name mangling could have been used for non-native methods, but we chose to generate more human-readable names.

[13] This has been implemented in SPARC assembly and is one of the few parts of the runtime system that is platform dependent.

- The runtime system must know the signature of the native method in order to construct the actual parameters. Hence, `j2s` must tell it somehow to the runtime system.

Our compiler solves this problem by generating a *stub function* that forwards the call to the runtime system. The stub function contains only a call to `rt_Native`*ReturnType*`Method()`, which takes a pointer to the method meta-data (see section 9.3.2) of the native method and the actual parameters for this call. Static methods pass a null pointer instead of the "this" reference. The runtime system can access both the method signature and the address of the actual native method (to perform the call) through the method meta-data. The actual parameters (which are used to build the call stack) are passed as a variable-length argument list ("`...`"). For instance, the Java class

```
class Print {
  native static public void print(int i);
}
```

generates the following code:

```
// JNI function (implemented in C)
extern void Java_Print_print__I(void *env, void *this, int);
...

// Prototype of the runtime function to perform native method calls
extern void rt_callNativeVoidMethod(struct method_meta_data *, ...);

// Method meta-data of class Print
static struct method_meta_data Print_method_meta_data[2] =
  {
     { "<init>", 0, 0,  "()V",  -1, Print_1init1, (void *)0, 0, 0 },
     { "print", 0, 264, "(I)V", -1, Java_Print_print__I_rts_stub,
       Java_Print_print__I, 0, 0 }
  };
...

extern void Java_Print_print__I_rts_stub(int arg0)
  {
    rt_callNativeVoidMethod(&Print_method_meta_data[1],
                            (void *)0, /* "this" reference */
                            arg0);
  }
```

Note that code that generates a call to the `Print.print()` function always uses the stub function (either by calling it directly or through the virtual function table).

9.2.4 Static Fields

SUIF code uses a group type to bundle all static fields of a class into one common structure. Static fields are represented with `field_symbols` (which are inserted into the group type). For example, the static fields of the following Java class

```
class StaticFields {
  static final char1= 'a';
  static       char2;
  static final str= "foobar";
}
```

are represented in SUIF code as

```
/* type (corresponds to SUIF variable symbol of a group type) */
struct StaticFields_static_struct
  {
    unsigned int StaticFields__char1;
    unsigned int StaticFields__char2;
    struct java_lang_String* StaticFields__str;
  };

static char *internalized_strings[1] =
  { "foobar" };

/* static storage allocation (corresponds to SUIF variable definition) */
struct StaticFields_static_struct StaticFields_static =
  {
    97, /* static initialization (ConstantValue attribute) */
    0,  /* default initialization */
    &internalized_strings[0]
  };
```

Alternatively, every static field could have been represented as an autonomous global variable symbol. Our approach makes debugging easier, but complicates code generation for field accesses.

OSUIF code uses `static_class_variable_symbol`s to represent static fields and places them into the corresponding class type (see section 5.7). The field access and modifier flags are attached as attributes to the symbol.

9.2.4.1 Declarations vs. Definitions

Variable definitions statically allocate storage for a certain variable symbol. This corresponds to a variable definition in C. Because in SUIF code every static field is associated with a single storage location, only one variable symbol with a variable definition exists for every group type that represent the static variables. This variable definition lives in the definition block of the corresponding class's file block. OSUIF code inserts the definition into the definition block of the corresponding class type.

Variable symbols that have no variable definition are references to static fields of auxiliary classes. They correspond to `external` variable declarations in C. For separate compilation, these symbols are put into the global external symbol table (SUIF code) or inserted into the corresponding class type (OSUIF code).

9.2.5 Instance Fields

SUIF code uses a `field_symbol` (which is part of a group type) to represent an instance field. This representation is essentially analogous to static fields. OSUIF code uses a `class_field_symbol` (see section 5.5). Java field access and modifier flags are represented with the attributes that the class field symbol offers.

9.2.5.1 Object Layout

SUIF code uses a `group_type` to represent the object layout of Java classes and arrays. OSUIF code keeps it (implicitly) in the `class_type` (see section 5.4).

Figure 9.1 shows the general structure of the object layout. The first entry is always the object's runtime meta-data (refer to section 9.4) followed by the object's instance fields.

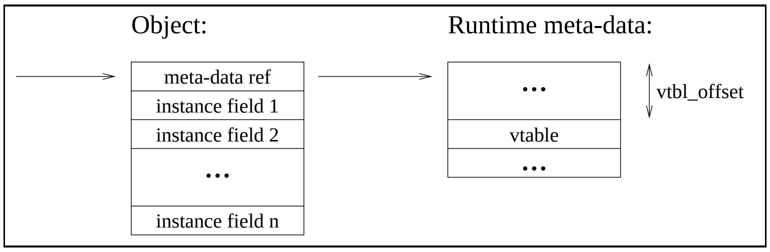

Figure 9.1: Object Layout

All the information that is shared by the objects of a particular class or array[14] (e.g., the virtual function table) is accessible through the meta-data pointer. Having only a single additional reference per object minimizes the memory usage, which is important if an application allocates a large number of small objects.

The object layout for fields is straightforward: If class B subclasses from class A, A's memory layout for instance fields must be a prefix of B's layout for instance fields. This way generated code for class A can be reused in A's subclasses. Note that interface inheritance need not be considered because interfaces cannot contain instance fields.

The layout for a Java array instance contains a field for the array `length` and an (internal) `array` field for the elements. `j2s` builds the following object layout for a `java.lang.Integer[][]` array:

```
// java.lang.Integer[][]
struct java_lang_Integer2_array
  {
    void *__cp;  /* meta-data ref */
    int length;
    struct java_lang_Integer1_array *array[1];  /* array elements */
  };

// java.lang.Integer[]
struct java_lang_Integer1_array
  {
    void *__cp;
    int length;
    struct java_lang_Integer *array[1];
  };
```

The `array` field assumes the size of 1 for the array. We could have used any non-negative size because this data type is used only conceptually as a "type template," which can be laid over an arbitrary memory location. It allows us to represent array accesses with high-level IR instructions instead of low-level pointer arithmetic. This works because neither SUIF nor the generated C code performs array bound checks based on the length of the array type. This scheme makes it also possible to share a single type template for all Java array instances of the same type. (Recall that the length of an array is not part of a Java array type.) A SUIF `array_type` can have an arbitrary number of elements (including zero), but the ANSI C standard forces us to give the array a non-zero length.[15]

[14]Even though the compiler generates compile-time meta-data only once for all array classes, the runtime system creates separate runtime meta-data for every array type.

[15]Note that we cannot take advantage of the fact the `**x` and `*x[]` are identical types in C, because they are completely different types in SUIF.

Figure 9.2 shows the object layout for `java.lang.Integer[3][2]` after the object allocation.

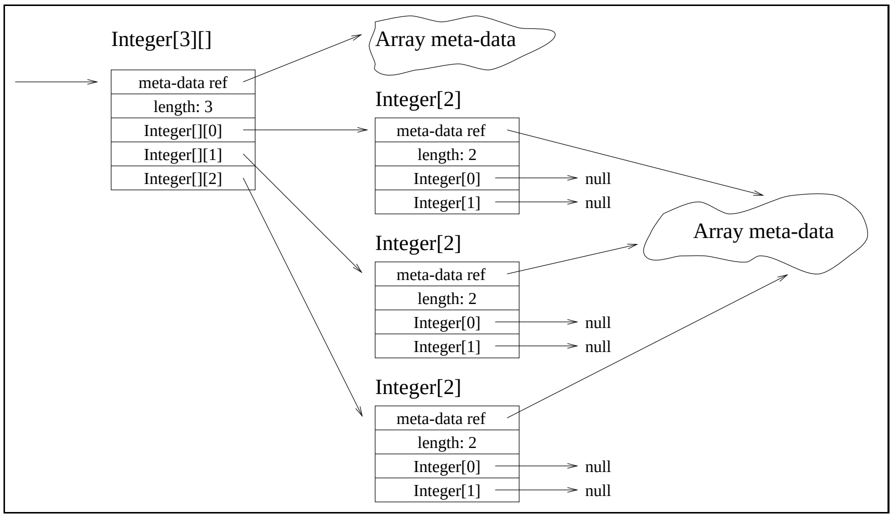

Figure 9.2: Object layout for `Integer[3][2]`

9.2.5.2 Declarations vs. Definitions

For SUIF code, the declaration of a single instance field is not possible. In this case, the whole object layout must be generated. In contrast, OSUIF code can contain incomplete class types as explained in section 5.4.

No declarations exists for instance fields because they are never statically allocated. Instead, they are allocated dynamically on the heap with calls to the runtime system (see next section).

9.2.6 Object Allocation

The JVM provides bytecodes for the allocation of class and array instances. These bytecodes merely allocate the memory and initialize the object fields with default values. A subsequent method call to a constructor brings the object in a proper state.

Allocation of class instances is performed by the `new` bytecode. `j2s` generates a call to the runtime system that takes a pointer to the class meta-data (see section 9.3.1) of the corresponding class. The following code is generated for a class instance of X:

```
// Prototype of the runtime function for "new" bytecode
extern void* rt_new(class_meta_data* clazz);

X* x= (X*) rt_new(X_meta_data);
```

The runtime system

- allocates the appropriate space for the new instance on the heap.[16]

[16]The class meta-data contains the object size.

- initializes the fields of the new instance by zeroing them.[17]
- sets the pointer to the runtime meta-data (see section 9.2.5.1).
- checks if class X needs to be initialized (see section 9.2.17).

Allocation of array instances is very similar. The runtime system provides functions for `newarray`, `anewarray`, and `multianewarray`:

```
// 1-D Array of a Primitive Type
JArray *rt_newarray(int length, atype_enum atype);
// 1-D Array of a Reference Type
JArray *rt_anewarray(int length, ClassMetadata *md);
// multi-D Array of a Primitive Type
JArray *rt_multinewarray(int dims, atype_enum, ...);
// multi-D Array of a Reference Type
JArray *rt_multianewarray(int dims, ClassMetadata *md, ...);
```

Note that in addition to the array type, the number of array elements is given, which is used by the runtime system to compute the object size. In contrast to classes, no initialization check need be performed. Furthermore, multi-dimensional arrays have to be initialized with array objects of the sub-dimension; class instances (as opposed to array instances) are always initialized to `null`. For example, a `multianewarray` to allocate `java.lang.Integer[3][2]` is coded by `j2s` the following way:

```
struct java_lang_Integer2_array* objectref=
  (struct java_lang_Integer2_array*)
    rt_multianewarray(2, java_lang_Integer_meta_data, 3, 2);
```

Figure 9.2 depicts the object layout right after the call.

9.2.7 Field Access

In general, field access in SUIF can be coded with either `field_access_instructions` or `extract_field-_instructions`. Both take a base address and `field_symbol`, but the former returns the address of the field, whereas the latter returns the value of the field.[18] These instructions make it possible to represent field accesses with symbolic names rather than pointer arithmetic.[19] Field access differs slightly for instance and static class fields and for arrays. We discuss each case in the subsequent sections.

9.2.7.1 Instance Fields

An instance field is retrieved with `getfield` and set with `putfield`. (O)SUIF code generation first has to look up the group (class) type that represents the object layout of the class instance. The (class) field symbol can now be looked up in the symbol table of the group (class) type. Once the (class) field symbol is retrieved, the SUIF code to get the address of the instance field can be constructed (see figure 9.3, "Field reference").

The high-level SUIF representation to access the address of an instance variable corresponds to the following low-level code:

$$objectref + field\_offset$$

[17] This corresponds to the default initialization [GJS96, Section 2.5.1].

[18] The `extract_field_instruction` can have multiple return values, but `j2s` does not make use of this feature.

[19] Currently we have to use the SUIF 2.0 to SUIF 1.0 converter to translate the generated code to SUIF 1.0. Because SUIF 1.0 has no high-level instructions for field access, the converter replaces the symbolic field names with numerical offsets, which makes the C output less readable.

Note that *field_offset* is a compile-time constant, which is symbolically represented in the SUIF code.

To access the field contents (for `getfield`), an additional dereference is necessary, which is coded with a `load_instruction` (figure 9.3, "Get field"). To store a *value* into the field, a `store_instruction` is used (figure 9.3, "Set field").

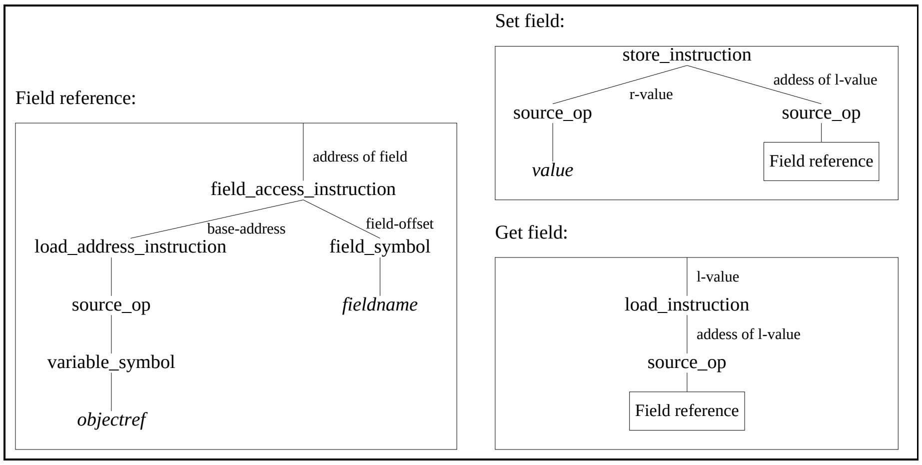

Figure 9.3: SUIF code to get/set fields

9.2.7.2 Static Fields

A static field is retrieved with `getstatic` and set with `putstatic`. SUIF code has to first look up the variable symbol that represents the static variables of the class (see section 9.2.4) in the global external symbol table. The variable symbol's type is the group type that contains all the static field. Hence, the static field can be looked up in the symbol table of this group type. OSUIF code can directly look up the static class field symbol in the corresponding class type.

The actual code that is generated for both SUIF and OSUIF is analogous to the code generated for instance fields. In figure 9.3 *objectref* must be replaced with the variable symbol that represents the static variables of the class. The high-level SUIF representation to get the address of the static field $X.f$ corresponds to the following low-level code:

$$X\_\mathtt{static} + \mathit{field\_offset}$$

Note that the resulting address of a static field is a compile-time constant. (Refer to section 9.2.7.2 for a description of the static field layout.)

9.2.7.3 Array Fields

The "length" field of an array is retrieved with `arraylength` bytecode. The effect of the generated code is identical to `getfield`, but uses an `extract_field_instruction`, which directly returns the value of the length field. (We did not have to factor out the code that retrieves the address of the length fields because the field is immutable once the object has been constructed.)

The generated code to get the address of an array element first accesses the (class) field symbol that represents the internal `array` and then retrieves the array element. The address of an array element is obtained by an `extract_field_instruction` followed by an `array_reference_instruction` as shown in figure 9.4.

The high-level SUIF representation to access the address of an array element corresponds to the following low-level code:

$$(\textit{arrayref}\texttt{->array}) + (\textit{element\_size} * \textit{index})$$

For T`aload` bytecodes, a `load_instruction` is used that retrieves the value of the array element and for T`astore` bytecodes a `store_instruction` is used that stores a *value* into the array element. This is analogous to instance fields.

So far, we did not tell the whole story. The `aastore` bytecode requires an assignment compatibility test. That is why the code generation uses a call to the runtime system

```
void rt_aastore (JArray *arrayref, JInt index, JObject *value);
```

which performs the test and the actual store of the value into the array. In fact, the runtime system offers functions for all T`aload` and T`astore` bytecodes. The code generation can use these calls, but this additional functionality is not strictly necessary (and yields slower code).

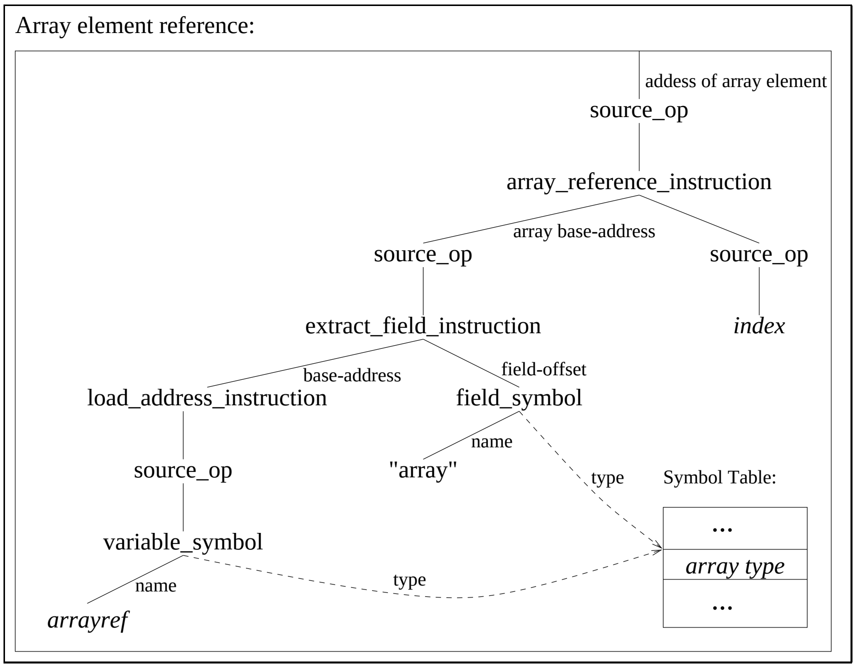

Figure 9.4: SUIF code to load/store array elements

9.2.8 Method Calls

Method calls are generated for calls to both static and instance methods. Calls to instance methods can be statically or dynamically bound. We discuss the code generated for each of these calls in the following.

9.2.8.1 Parameter Passing

Java passes all parameters by value (including object references). Note that only scalar and pointer values need to be passed as parameters because Java does not support expanded objects, such as Eiffel. This is the case for return values as well.

Instance methods expect the "this" reference of the receiver as the first parameter. This corresponds to the OSUIF convention (see section 5.6). The order of the parameters corresponds to the textual order given in the class file.

The representation of a `procedure_type` in SUIF directly corresponds the Java method descriptor, except for the return type. A `void` return type in Java correspond to a function with no result in SUIF.

9.2.8.2 Statically Bound Calls

Method calls that are statically bound (i.e., during compile time) are performed by the `invokespecial` and `invokestatic` bytecodes.

In SUIF, both calls are implemented as normal procedure calls with a `call_instruction`. The code generated for `invokespecial` passes the "this" reference as the first parameter. In OSUIF, a `method_call_instruction` with a static function address is used for `invokespecial`. This way, we preserve the information that a statically bound method call is performed (as opposed to a function call).

Note that SUIF uses a high-level representation for procedure calls. A `call_instruction` takes both the address of the caller and the actual parameters. Other, more low-level IRs, encode a procedure call only with the function address and rely on other instructions to pass the actuals in registers and/or on the call stack.

9.2.8.3 Dynamically Bound Calls

Method calls that are dynamically bound (i.e., during runtime) are performed by the `invokevirtual` and `invokeinterface` bytecodes.

Constructor calls and calls to private methods must always be sent to the defining class and hence are never dispatched. The JVM specification states that `invokespecial` must be used for these cases, which means that the compiler automatically generates statically bound calls for them.

Given an object and a method signature, dispatching finds the function pointer of the Java method that corresponds to the receiver's dispatch type (i.e., class, array, or interface). Since Java is statically typed, the method signature is known during compile time. The receiver's type is either known during compile time or can be determined during runtime by following the reference to the runtime meta-data. Hence, a naïve table-based dispatch implementation could store all possible type/method pairs in a two-dimensional array (figure 9.5). This scheme, called *selector table indexing*,[20] is not practical because the array is too big and very sparse.

9.2.8.3.1 Class Dispatch In a statically typed language that has only single inheritance, inherited methods can keep the same position in the lookup table and the row for each class (see figure 9.5) can be collapsed into a *virtual function table* (vtable) without gaps. Vtables were introduced in Simula and are the preferred dispatch technique for C++ and Modula-3. They can be constructed for Java if only class inheritance is considered. This is the case for *class dispatch*, which is performed with the `invokevirtual` bytecode.[21] For class dispatch, the static type of the receiver, which is either a class or an array, is known during compile time.

[20] In the following, we use the terminology of [DHV95] to describe dispatch techniques.

[21] To be more precise, Java interfaces do not influence class dispatch because they only "force" the presence of certain methods in a class.

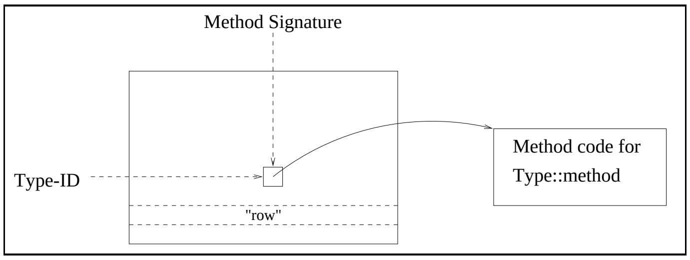

Figure 9.5: Table-based dispatch

All arrays can share a single vtable, which is identical to `java.lang.Object`'s vtable, except for the `clone()` method, which must be redefined. The runtime system implements `clone()` and overrides the appropriate slot in the array vtable with it. The runtime system could reveal the function address of its `clone()` implementation to `j2s` for the vtable construction, but we want to keep the interface to the runtime system as simple as possible. In fact, the runtime system can replace arbitrary methods behind the compiler's back as long as the semantic of a method remains unchanged.

Figure 9.6 shows pseudo Java code that implements classes and interfaces. Note that every class and interface could live in a separate file, which can make the generation of dispatch meta-data more complicated. Figure 9.7 shows the constructed vtables that are used for class dispatch. These vtables are emitted by the compiler as part of the class meta-data (see section 9.3.1) and are copied by the runtime system into the runtime meta-data (see section 9.4).

The following code is generated to get the address of the method if the actual type of the receiver is unknown during compile time (which is usually the case):

$$\underbrace{*objectref}_{\text{runtime meta-data}} \quad + \quad \underbrace{offset}_{vtbl\_offset\ +\ vtbl\_slot}$$

Thus, dispatch is implemented with a single pointer dereference and a constant addition. More efficient dispatch code can be generated if the static type of the receiver is either an array or a final class. In both cases, the dynamic type must be identical to the static type and hence the following code can be emitted for dispatch on the static type X:

$$\underbrace{X\texttt{\_meta\_data}}_{\text{class meta-data}}\ \texttt{->}\ \texttt{vtbl[}vtbl\_slot\texttt{]}$$

This expression can be easily converted into a function pointer constant by a SUIF back end.[22] We chose this representation, as opposed to directly generating the function pointer, because it guarantees that methods that have been replaced in the vtable by the runtime system (see above) still get called.

9.2.8.3.2 Interface Dispatch If interfaces are taken into account, dispatch becomes more complicated. *Interface dispatch* is performed by the `invokeinterface` bytecode. The signature of the method that needs to be called and the interface that supports this method are known during compile time. The receiver can be any class or array instance that supports this interface. Thus, interface dispatch is very similar to the dispatch of method sends in dynamically typed languages, with the important difference that it is guaranteed that the receiver actually understands the method (i.e., no runtime error can occur).

[22] We currently rely on the C compiler to perform this transformation.

```
interface Cloneable {
  Object clone();
}

interface Printable {
  void print();
  void print_verbose();
}

// root of the class hierarchy (not legal Java!)
class Object {
  int hashcode();
}

class A /* extends Object */ implements Cloneable {
  int hashcode();
  void a();
  Object clone();
}

class B /* extends Object */ implements Cloneable, Printable {
  // hashcode() is inherited from Object
  void b();
  void print();
  void print_verbose();
  Object void clone();
}

class C extends B /* implements Cloneable, Printable */ {
  int hashcode();
  // b() is inherited from B
  // print is inherited from B
  void print_verbose();
  // clone is inherited from B
  c();
}
```

Figure 9.6: Pseudo Java class hierarchy

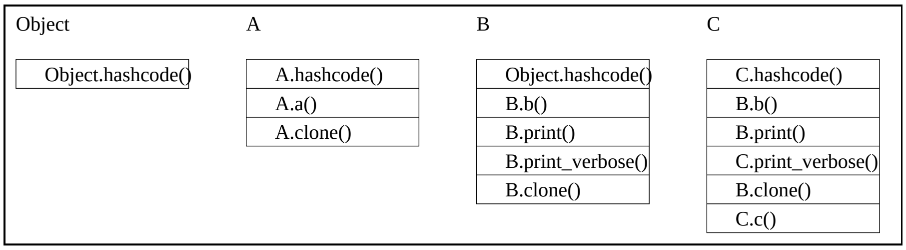

Figure 9.7: Vtables for class hierarchy in figure 9.6

Class dispatch happens far more often than interface dispatch and its implementation is usually easier, more space efficient, and at least as fast as interface dispatch. Furthermore, both dispatch mechanism are conceptually orthogonal in Java. For these reasons, `j2s` uses two different dispatch techniques to implement class and interface dispatch.

Dynamic method lookup [DHV95] techniques are applicable for interface dispatch, namely

Meta-data search: The runtime system can search through the method meta-data (see section 9.3.2) to find the right method. If the search for the receiver's class fails, it proceeds up the inheritance chain until the method is found. This technique is very similar to *dispatch table search*, which is often used as a backup strategy when faster techniques fail. Our implementation falls back to the meta-data search if the global cache lookup fails (see below).

Global lookup caches: A global cache can be used to speed up interface dispatch. (First-generation Smalltalk implementations relied on this scheme.) For Java, each cache entry consists of a class, a method signature, and a method address. The receiver's class, combined with the method signature, hashes into an index of the cache. If the receiver's class matches with the hashed cache line, the resident method address is used for the call. Otherwise, a meta-data search (see above) finds the correct method and updates the cache line. `j2s` generates a call to the runtime system that passes the parameters of the hash function and returns a function pointer as the result of the lookup. For example, the interface call

```
Printable printable= new B();
printable.print();
```

is implemented with the following code:

```
// Prototype of the runtime function for method lookup
void* rt_intf_lookup(void* objectref,
                     char* method_name,
                     char* method_descriptor);

Printable* printable= rt_new(B_meta_data);
void (*method_address)(Printable*) = rt_intf_lookup(printable,
                                                     "print", "()V");
method_address(printable);
```

Any hash function can be used. In fact, if the complete class hierarchy is known, it is possible to compute a *perfect hash function*. The runtime system currently does not take advantage of this property, however, it could be easily implemented without modifying the compiler. Currently, the runtime system uses the following hash function:

```
int lookup_table_slot=
  (objectref ^ method_name ^ method_descriptor) % TABLE_SIZE
```

For the above hash function and a lookup table size of 256 elements, compilation of Java classes with `javac` result in a miss ratio of about 10%.

Inline caches: Often the receiver type of a specific call site changes rarely. This can be exploited by caching the result of the most recent method lookup for every call site [DS84].

Polymorphic inline caching: Inline caches are effective only if the receiver type remains relatively constant at a call site. Polymorphic inline caches [HCU91] reduce the miss ratio by caching several lookup results instead of just the most recent one.

We chose to implement a global lookup cache because it was easy to implement, required minimal compiler support, and allowed us to test the method meta-data (whose main purpose is to support reflection and JNI calls) at an early stage. Furthermore, interface dispatch is used very seldom in Java applications (at least as of this writing) and hence has not a significant performance impact. Choosing a more efficient technique requires extensive study of the runtime behavior of a "representative" set of Java programs to find the best candidate, which exceeds the scope of this thesis.

In contrast to dispatch for dynamically types languages, interface dispatch can be implemented with *static method lookup techniques* as well. In the following, we do not exploit the whole design space because a large number of different dispatch techniques exist, each of them having a myriad of slight variations. Instead, we focus on one promising implementation technique and discuss several variations of it.

An interface supports a fixed number of methods that have to be defined in every class that implements this interface. Thus, these classes can provide for each interface a separate vtable that contains the interface's methods at the same offsets.

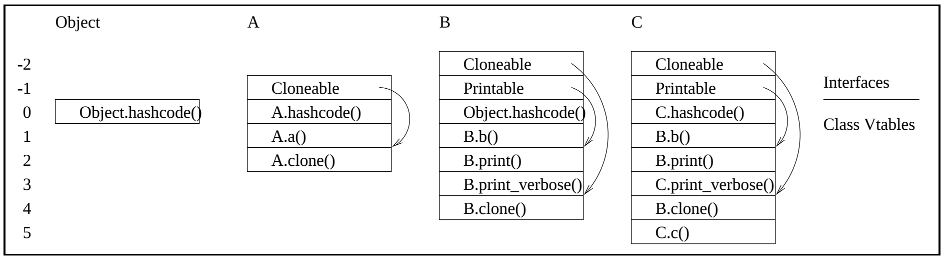

Figure 9.8: Interface dispatch for class hierarchy in figure 9.6

Figure 9.8 shows an example how this scheme can be implemented. Negative offsets at the vtable of a class contain pointers to the interface vtables for every interface that the class implements. Note that the offset of a particular interface vtable pointer can vary for different classes. For example, the interface `Cloneable` is at offset -1 and -2 for A and B, respectively. Figure 9.8 optimized the space that is required for interface dispatch by reusing the class's vtable. (Recall that the vtable of every interface is a subset of the class's vtable.) However, this is not possible if a method is declared along two different interface inheritance paths. In this case, part of the class's vtable has to be duplicated. Figure 9.9 depicts an example of multiple interface inheritance and the corresponding interface vtables for class X, which implements D. Method `a()` is reachable from two different inheritance paths ($B \rightarrow A$ and $C \rightarrow A$). Therefore either B's or C's vtable has to be duplicated for class X.

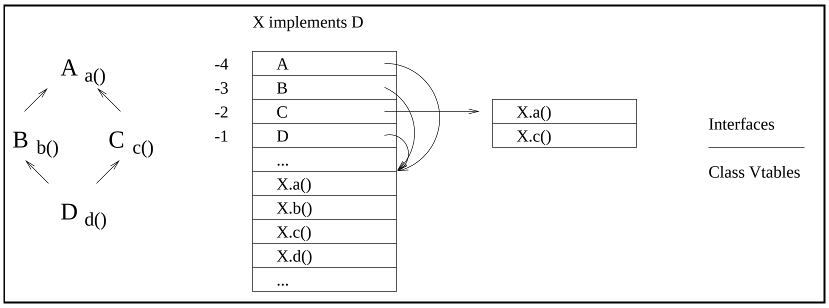

Figure 9.9: Vtable duplication for interfaces

So far, we introduced interface vtables but failed to explain how a dispatch mechanism can actually find the appropriate one. As explained before, we face the problem that interface vtable pointers can be at different (negative) offsets for every class.

In the following, we explain several techniques to access the appropriate interface vtable pointer offset:

- Every interface can be associated with a unique number that is used as the negative offset to access the interface vtable pointers. This implies that the complete class hierarchy must be known at compile time. This approach results in a large and very sparse table whose size is the number of all the interfaces in the transitive closure of the main class. Thus, the total space for the interface vtable data is $\#(classes) * \#(interfaces)$. An application with a transitive closure of 1000 classes and 300 interfaces requires about 300KB.[23] The dispatch is implemented with the following code:

  $$*(*objectref - iface\_vtbl\_offset) + vtbl\_slot$$

  Thus, two pointer dereferences and two additions with constant values are needed.[24]

  To reduce table sizes, *Selector coloring* [DHV95] can be used. The CACAO compiler [KG97], for instance, directly stores interface methods at negative offsets from the class pointer and uses coloring to reduce the number of distinct offsets for the interface methods.

  Alternatively, the per-class interface tables can be merged into a large, global table and compression techniques, such as *row displacement* [DH95] can be used to reduce the size of the global table. Since these tables are usually very sparse we can expect a significant reduction.

- A linear search through the interface vtable pointers can be used to find the right one. Usually the number of interfaces that a class implements is small, which makes this search much faster than a dispatch table or meta-data search. Hence, this technique could be used as a faster backup technique if the global lookup cache misses. If the complete class hierarchy is known at compile time, interfaces can be ordered, which allows to perform a binary search.

- We can further speed up the search by using hash functions that return the proper interface vtable pointer. A global function

  $$H : ClassID \times InterfaceID \rightarrow iface\_vtbl\_offset$$

[23] These are plausible numbers for a medium-size AWT application.

[24] Note that separate compilation forces us to use an additional indirection to access *iface_vtbl_offset* in the global meta-data.

that takes the receiver's class and the statically known interface can be used.

Since we know the interface during compile time, we can also directly call a specialized hash function for the interface:

$$H_{InterfaceID} : ClassID \rightarrow iface\_vtbl\_offset$$

Optionally, every class meta-data could provide its own hash function:

$$H_{ClassID} : InterfaceID \rightarrow iface\_vtbl\_offset$$

$H_{ClassID}$ typically has to hash far less different values than $H_{InterfaceID}$, but can only be accessed with an additional indirection. If the complete class hierarchy is known, perfect hash functions can be computed.

Separate compilation can be supported, but the interface vtables (and hash tables) have to be constructed in the global compile-time meta-data, which means an additional indirection to access them. Dynamic loading of classes can be supported in principle, but requires the re-computation of the interface vtables (and hash tables) every time a new class is loaded.

Although the implementation of these more efficient dispatch techniques is beyond the scope of this thesis, we plan to incorporate at least one of them in the future.

9.2.8.4 Implementation

The abstract interface of the vtable construction is implemented with the class `Vtbl`. The following code shows the public interface:

```
class Vtbl {
public:
  virtual int vtbl_offset(Unit* clazz, Method* method) =0;
  virtual int vtbl_length(Unit* clazz) =0;
  virtual variable_definition* build_vtbl(Unit* clazz) =0;
};
```

The code generation for the class dispatch (see section 9.2.8.3.1) uses `vtbl_offset()` to obtain the vtable slot for the dispatch code. The variable definition returned by `build_vtbl()` contains the vtable for a particular class. The address of the variable symbol that represents the vtable is attached to the appropriate class meta-data (see section 9.3.1). The vtable is implemented as an array of "`void (*)()`" function pointers. For instance, the following code shows `java.lang.Character`'s vtable:

```
static void *vtbl[14] =
  {
    Java_java_lang_Object_clone_rts_stub,
    java_lang_Character_equals,
    java_lang_Object_finalize,
    ...
    java_lang_Character_compareTo__9Ljava_lang_Object_2_9I
  };
```

If a function is called through the vtable, the returned function pointer has to be cast to the appropriate function pointer type before the call is made.

The vtable construction does insert neither constructors nor private methods into the vtable. This is not strictly necessary but yields smaller tables.

We provide two concrete implementations for the vtable constructions, which both subclass from `Vtbl`. The `LazyVtbl` class is used for the SUIF code generation. The vtable construction uses `j2s`'s internal class file representation to build the vtables. The `GreedyVtbl` solemnly operates upon OSUIF IR structures to generate the vtables. This class builds the vtables for all classes in a single run and keeps them in a dictionary, whereas `LazyVtbl` builds them on demand, i.e., when `build_vtbl()` is called. Therefore, `LazyVtbl` is more suitable for separate compilation. On the other hand, `GreedyVtbl` can conceptually be run as a separate pass. In fact, `GreedyVtbl` has been used as the starting point for the OSUIF vtable construction pass. (We plan to migrate to the OSUIF vtable construction.)

9.2.9 Operand Stack and Local Variable Array

The code generation has to map local variables and operand stack entries (see section 6.4) to SUIF variable symbols. Both local variables and stack entries are untyped; SUIF symbols are typed. To get a strongly typed intermediate representation of the SUIF code (i.e., without unnecessary type casts), the type of every local variable and stack entry must be known at every program point. This is precisely the information already inferred in the type recovery pass (see section 8.4.1).

In the following, we explain how untyped local variables and stack entries are mapped to typed SUIF symbols.

9.2.9.1 Local Variables

The JVM specification allows reusing of a variable slot with different types at different program points. Figure 9.10 depicts a brief code sequence that uses local variable slot zero to first store an `int` value (index 2) and subsequently reuses the same slot to store an object reference (index 4).

Java source	#	Bytecodes	Generated Code
`int i = 42;`	0:	`bipush #42`	`int` $\texttt{lvI}_0$;
	2:	`istore 0`	$\texttt{lvI}_0 = 42$;
`Object o = null;`	3:	`aconst_null`	`java_lang_Object*` $\texttt{lvRef}_0$;
	4:	`astore 0`	$\texttt{lvRef}_0 = (\texttt{void*})0$;

Figure 9.10: Local variable translation

Since SUIF variable symbols are typed, a separate SUIF variable with the corresponding type must be created for every type that the local variable slot can hold during execution of a Java method. Hence, a single local variable slot can map to several corresponding SUIF variables. For example, the Java code in figure 9.10 forces us to generate two SUIF variables ($\texttt{lvI}_0$ and $\texttt{lvRef}_0$), both of them representing the local variable slot zero (at different program points).

The mapping of local variables for primitive types and reference types is slightly different:

- A local variable that holds a primitive type can directly be mapped to a SUIF variable with its corresponding SUIF type. This means that for primitive types at most five SUIF variables are needed for a single local variable slot: Each one for `int`, `long`, `float`, `double`, and `returnAddress`.
- A local variable that holds an object references of type T actually represents type T and all of its subtypes. To handle this case, we use the least general reference type of the local variable to represent all reference types that this local variable can hold. This type is determined by the type recovery pass (see section 8.4.1).

Since the SUIF type system resembles that of an imperative language, subtype relationships cannot be modeled. This means that explicit type casts must be introduced in the intermediate representation if the object reference is actually used in a certain type context (e.g., a field access or method call of a superclass of T). The compiler guarantees that these (potentially unsafe) casts are actually legal. (One can argue that this is a breach of the otherwise type-safe IR representation.) OSUIF will model a type system that allows to express subtype relationships, rendering type casts unnecessary.

9.2.9.2 Operand Stack

The mapping of a stack to local variables or registers is a well-known problem. For example, such a mapping must be performed by compilers that translate languages with programmer visible stacks (*stack-based languages*), such as Forth or PostScript, to native code for a register machine.

The mapping of stack entries to C local variables, which is described in [EM95] and [Ert96] for the Forth language, is very similar to the mapping of the Java stack to SUIF variables because SUIF's IR constructs closely resemble the constructs found in an imperative language such as C. In this scheme, every stack entry is represented with a C local variable (see figure 9.11). A local variable represents a stack entry with fixed stack offset. Since stack entries can typically hold different types during execution, they are modeled with a union type (`Cell`). `sp` denotes the virtual stack pointer during compile time.

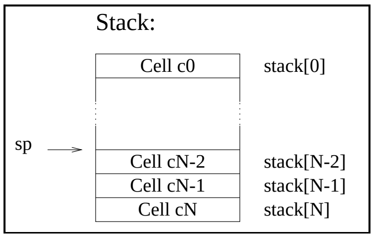

Figure 9.11: Stack mapping with C local variables

Ertl remarks in [Ert96, Chapter 4] that "the emphasis [in the translation to C] is on simplicity. The translation strategy utilizes the optimization capabilities of optimizing C compilers to produce efficient native code." Our implementation is in unison with this approach because, as we explain in more detail in the following, `j2s` uses a straightforward translation of the Java stack to SUIF variables and relies on the standard optimizations of the SUIF compiler system to improve the code.

The stack-based Java class files caused a renewed interest in how to map stack entries to temporary variables or registers. This problem arises, for example, for compilers that translate class files to C or native code [Bot97] [PTB+97] [MMBC97] [KG97] [MDG97] [HGH96]. All of them are based on symbolic execution of the operand stack.

The bytecode to SUIF 1.0 compiler [MDG97] developed at Colorado State University directly constructs expression trees during the symbolic execution. Two cases must be distinguished, based on the nature of the bytecode:

- If the bytecode pushes a result on the operand stack, the operands (which are expression trees) can be combined to an expression tree that holds the result of the computation. This expression tree is pushed back on the stack. Figure 9.12 depicts the compilation of the `iadd` bytecode. The left side shows the symbolic stack before code is generated for `iadd`. The two topmost stack entries are the operands that `iadd` takes. The right side shows the stack after the code generation. The topmost stack element contains the expression of the `iadd` result.

Stack entries contain (sub-)expression trees. Such an expression tree holds either a SUIF instruction (which is the root of the expression tree) or a SUIF variable symbol (which is always a leaf node).

- If the bytecode stores into a local variable, an assignment has to be generated. For example, the code generation for T`store` bytecodes obtains the r-value for the assignment by popping the topmost stack element, which is an expression tree, from the symbolic stack.

This scheme is also used by Bothner [Bot97] and [BK97]. Bothner uses it when generating code for `gcc`'s internal tree representation. Using expression trees saves load/store instructions and the introduction of temporary variables, but, as we argue later, does not necessarily improve the code quality.

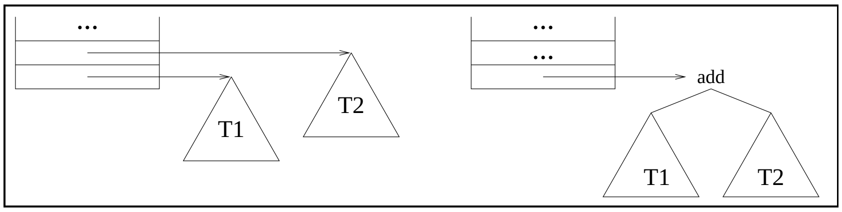

Figure 9.12: Stack mapping with expression trees for `iadd`

The CACAO compiler [KG97] assumes an unlimited amount of virtual registers for the translation to its intermediate representation, but keeps track of registers that must contain the same values. For example, if a stack entry is stored at a certain local variable slot, the compiler does not generate a `move` instruction. Instead, it records that both registers are aliased. (This scheme is similar to a more limited form of copy propagation.) The actual register allocation maps virtual registers to physical registers. Similarly, the Caffeine compiler [HGH96] assigns each stack entry a unique virtual register in its intermediate representation. Register allocation is later used to map the unlimited number of virtual registers to a fixed number of physical registers.

The Harissa compiler [MMBC97] generates C code by using Ertl's scheme (see above) for the stack translation. Figure 9.13 shows Java source and the generated bytecode sequence from this source. The C code with Harissa's translation scheme is shown in figure 9.14 at the first column ("Stack Mapping"). Because the types on the Java stack are known for every program point, C variables with the corresponding types can be generated (`s`-variables). Local variables are represented analogous with `lv`-variables. In the generated C code, an l-value `s`-variable corresponds to pushing a stack value and an r-value `s`-variable corresponds to popping a value from the stack. Toba [PTB$^+$97] uses the same translation strategy. The authors of both implementations point out that useless copies can be eliminated with an optimizing C compiler. Harissa even shows the resulting C code optimized with the SUIF 1.0 system.

The `j2s` compiler uses a very simple scheme for the translation. Every push of the operand stack generates a new temporary variable. The second column ("Unoptimized") in figure 9.14 depicts the code generated by our compiler. As can be seen, the generated instructions are the same as in Harissa's or Ertl's code generation, but we have to generate more temporary variables. Bothner [Bot97] describes this technique as well and notes that this "simple-minded approach uses lots of virtual registers, and the generated code is very bad." This is true, but this "bad code" can be easily optimized.

As mentioned before, we rely on SUIF optimization passes to improve the generated code. In fact, copy propagation and constant propagation followed by dead code elimination [ASU86] yields code that is of the same quality as code generated with expression trees or code that tries to reuse temporary

Java source	Bytecodes	
`int i` $= 42$;	`bipush #42`	
	`istore 1`	
$\mathtt{i} = \mathtt{i} * (\mathtt{i} = \mathtt{i} + 1)$;	`iload 1`	
	`iload 1`	
	`iconst #1`	
	`iadd`	$i = i + 1$
	`dup`	
	`istore 1`	
	`imul`	
	`istore 1`	

Figure 9.13: Example Java source and bytecodes

Stack Mapping	Unoptimized	CopyP		CopyP & ConstP	
$\mathtt{sI}_0 = 42$	$\mathtt{tmp}_0 = 42$	$\mathtt{tmp}_0 = 42$		$\mathtt{tmp}_0 = 42$	D
$\mathtt{lvI}_1 = \mathtt{sI}_0$	$\mathtt{lvI}_1 = \mathtt{tmp}_0$	$\mathtt{lvI}_1 = \mathtt{tmp}_0$	D	$\mathtt{lvI}_1 = 42$	D
$\mathtt{sI}_0 = \mathtt{lvI}_1$	$\mathtt{tmp}_3 = \mathtt{lvI}_1$	$\mathtt{tmp}_3 = \mathtt{tmp}_0$	D	$\mathtt{tmp}_3 = 42$	D
$\mathtt{sI}_1 = \mathtt{lvI}_1$	$\mathtt{tmp}_4 = \mathtt{lvI}_1$	$\mathtt{tmp}_4 = \mathtt{tmp}_0$	D	$\mathtt{tmp}_4 = 42$	D
$\mathtt{sI}_2 = 1$	$\mathtt{tmp}_5 = 1$	$\mathtt{tmp}_5 = 1$		$\mathtt{tmp}_5 = 1$	D
$\mathtt{sI}_1 = \mathtt{sI}_1 + \mathtt{sI}_2$	$\mathtt{tmp}_6 = \mathtt{tmp}_5 + \mathtt{tmp}_4$	$\mathtt{tmp}_6 = \mathtt{tmp}_5 + \mathtt{tmp}_0$		$\mathtt{tmp}_6 = 1 + 42$	
$\mathtt{sI}_2 = \mathtt{sI}_1$	$\mathtt{tmp}_7 = \mathtt{tmp}_6$	$\mathtt{tmp}_7 = \mathtt{tmp}_6$	D	$\mathtt{tmp}_7 = \mathtt{tmp}_6$	D
$\mathtt{lvI}_1 = \mathtt{sI}_2$	$\mathtt{lvI}_1 = \mathtt{tmp}_7$	$\mathtt{lvI}_1 = \mathtt{tmp}_6$	D	$\mathtt{lvI}_1 = \mathtt{tmp}_6$	D
$\mathtt{sI}_0 = \mathtt{sI}_0 * \mathtt{sI}_1$	$\mathtt{tmp}_9 = \mathtt{tmp}_3 * \mathtt{tmp}_6$	$\mathtt{tmp}_9 = \mathtt{tmp}_0 * \mathtt{tmp}_6$		$\mathtt{tmp}_9 = 42 * \mathtt{tmp}_6$	
$\mathtt{lvI}_1 = \mathtt{sI}_0$	$\mathtt{lvI}_1 = \mathtt{tmp}_9$	$\mathtt{lvI}_1 = \mathtt{tmp}_9$	D	$\mathtt{lvI}_1 = \mathtt{tmp}_9$	D

Figure 9.14: Stack translation

variables. Figure 9.14 shows the impact of the SUIF optimizations on the code. The third column shows the code after copy propagation ("CopyP") has been applied. Dead code elimination removes the instructions that are succeeded with a "D." The last column shows the code generated for copy propagation and constant propagation ("CopyP & ConstP"). After dead code elimination, the original sequence of 10 instructions has been collapsed into 2.[25]

Always using a new temporary variable yields stack mapping code that is in SSA form [CFRW91] [App98].[26] As [EM95] points out, code in SSA form can yield better register allocation and simplifies copy propagation for C compilers. This observation holds as well for the SUIF 1.0 optimization passes and back ends. On the other hand, the authors of [BK97], which generate similar code then our compiler, point out: "We believe that the omission of instance variables from the SSA form causes a significant decrease in the quality of code generated." As opposed to [BK97], our current goal is not to generate highly optimized code. This would require further optimization on a much larger scale, such as type inference, inlining, hoisting of array bound checks out to loops, and eliminating class initialization checks.

It should be noted that `j2s`'s translation scheme was independently developed from the other approaches mentioned above. We only contrast them in retrospect with our scheme.

[25] Collapsing the remaining two instructions into a single expression would probably not further improve the code.

[26] Other code, for example, instance variable accesses are not in SSA form.

9.2.9.3 Stack Spilling

So far, we only described the mapping of the stack for straight-line code. If control flow merges with a non-empty stack, different incoming control-flow paths represent stack entries of the same stack pointer offset with different temporary variables. For example, in figure 9.15 control flow merges with a stack height of one at the bytecode at index 10.

We introduce copy instructions to assure that stack entries are kept in the same local variables after control flow merges.[27] This scheme implicitly assumes that all incoming control flow paths have the same *stack configuration* (see definition 8.4), which especially includes the same stack height. This property is guaranteed by the bytecode verifier and is very important for the code generation because unbalanced stacks at control flow merges would force us to introduce run-time stacks as a fall-back method in the compiler.[28]

Java source	#	Bytecodes
`static int is42(int i) {`	0:	`iload 1`
`  return (i==42) ? 1 : 2;`	1:	`bipush #42`
`}`	3:	`if_icmpne 10`
	6:	`iconst #1`
	7:	`goto 11`
	10:	`iconst #2`
	11:	`ireturn`

Figure 9.15: Example code for stack spilling

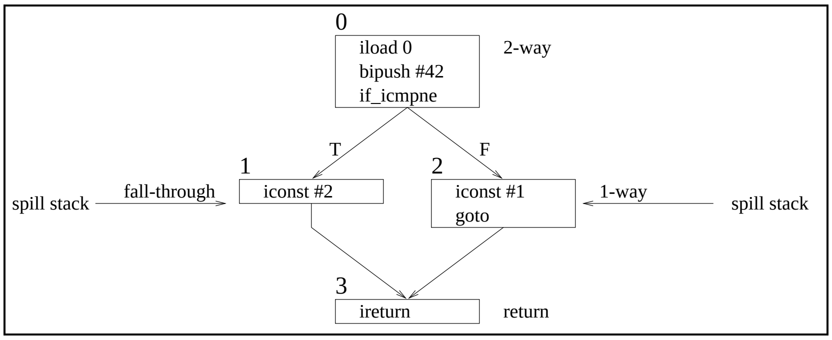

Figure 9.16: CFG of code in figure 9.15 (stack spilling required)

In the following, we illustrate stack spilling with the code example given in figure 9.15. Figure 9.16 shows the corresponding CFG with the basic block types. Stack spilling consists of generating copy instructions at the end of every merging basic block. If the basic block is a fall-through block, then the copy instructions are emitted after the code for the last bytecode has been generated (see figure 9.16;

[27] Copying is not necessary for Ertl's scheme because the respective stack entries of the incoming control flow paths are already in the same local variables.

[28] For example, the Forth language does not enforce balanced stacks [EM95]. (Interestingly, unbalanced stacks happen rarely in Forth and are in many cases due to programming bugs.)

basic block 1); otherwise they must be emitted before the control transfer instruction (see figure 9.16; basic block 2). The code generated for the the target basic block (see figure 9.16; basic block 3) simply uses the variables that contain the spilled stack for the code generation. The start stack configuration (which has been computed by the type recovery pass) is known for every basic block. It contains the SUIF variable names that hold the spilled stack entries. The following code is generated for the Java source in figure 9.15:

```
  extern int StackSpilling_is42(int lvI_0)
    {
      int tmp_0;
      int tmp_1;
      int tmp_6;
      int stkI_0;            /* holds spilled stack value */
      int tmp_10;

      label_0:
 0:   tmp_0 = lvI_0;
 1:   tmp_1 = 42;
 3:   if ((int)(tmp_0 != tmp_1))
          goto label_10;
    label_6:
 6:   tmp_6 = 1;
      stkI_0 = tmp_6;       /* spill stack */
 7:   goto label_11;
    label_10:
10:   tmp_10 = 2;
      stkI_0 = tmp_10;      /* spill stack */
    label_11:
11:   return stkI_0;
    }
```

We show the unoptimized code to demonstrate how the temporary variables (`tmp_*`) are spilled into common variables that represent the stack at the given merge point (`stk`*Type_StackDepth*). In this example the stack (of height one) is spilled into the variable `stkI_0`. Optimizing this code with copy propagation, constant propagation, and dead code elimination eliminates all temporary variables:

```
  label_0:
    if ((int)(lvI_0 != 42))
        goto label_10;
  label_6:
    stkI_0 = 1;      /* spill stack */
    goto label_11;
  label_10:
    stkI_0 = 2;      /* spill stack */
  label_11:
    return stkI_0;
```

Luckily, spilling of the stack happens rather infrequently (see table 9.1). The first row in table 9.1 indicates that, for example, the code generated for the transitive closure of `java.lang.String`, which consists of 358 classes, has only 6.6% basic blocks that must be spilled. On average, these blocks spill 1.3 stack entries.

By introducing the spilling code, we loose the property that all stack mapping code is in SSA. Interestingly, the stack mapping code corresponds to "resolved" SSA ϕ functions, which are introduced

Application	Classes	Methods	Basic blocks	BBs w/ spills	Spills/BB
String*	358	3414	12877 (100%)	855 (6.6%)	1.3
String	1	61	289 (100%)	16 (5.5%)	1.4
sun.tools.java.Environment	1	53	264 (100%)	46 (17.4%)	1.0

Table 9.1: Statistics for stack spills

at control flow merges to unify renamed temporary variables. Figure 9.17 depicts the correspondence between the SSA ϕ function and our code.[29]

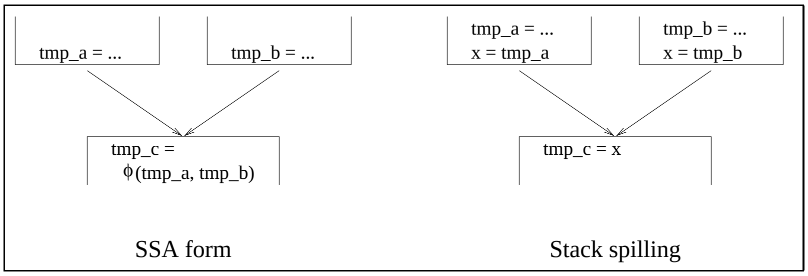

Figure 9.17: SSA form vs. stack spilling

SUIF offers a `ssa_phi_instruction`, but this instruction is supported neither by the Standard SUIF dialect nor by the converter. It might be worthwhile to generate code with ϕ functions as soon as a SUIF 2.0 pass become available that lowers `ssa_phi_instruction`s to Standard SUIF.

9.2.10 Load and Store Bytecodes

The T`load` bytecodes push a local variable at a certain index onto the operand stack; the T`store` bytecodes pop the topmost stack entry by storing it into a local variable at a certain index.[30] The T`load_<n>` and T`store_<n>` bytecodes are identical to T`load` and T`store`, respectively, except that the local variable index is given implicitly in the opcode and not as an operand.

T`load` is translated to a `copy_instruction` that assigns the SUIF variable that represents the local variable to a new temporary variable, which represents the new (symbolic) top-of-stack entry. (Mapping of local variables to SUIF variables has already been discussed in section 9.2.9.1.) For example, `fload_2` is translated to

```
float lvF_2;  /* local variable at slot 2 */
float tmp_x   /* temporary variable that represents topmost stack entry */

tmp_x = lvF_2;
```

Similarly, T`store` is translated to a `copy_instruction` that assigns the temporary variable that represents the current top-of-stack entry to the SUIF variable that holds the corresponding Java local

[29] Appel [App98] describes how a CFG with ϕ functions can be transformed into a set of mutually recursive function.

[30] These bytecodes must not be confused with T`aload` and T`astore`, which load/store array elements and hence are more closely related to `getfield` and `putfield`.

variable. The code generation (symbolically) pops the top-of-stack entry. For example `fstore_2` is translated to

```
float lvF_2;  /* local variable at slot 2 */
float tmp_x   /* temporary variable that represents topmost stack entry */

lvF_2 = tmp_x;
```

The following code implements the translation of the `istore` bytecode to SUIF code:[31]

```
void CodegenVisitor::istore(u4 ci, u2s index, boolean wide) {
  // Stack effect: ... value => ...
  source_op rvalue= pop_source_opnd();

  // Update local variable array configuration
  _lvs->istore(index);

  // Get SUIF variable symbol that represents local variable
  destination_op lvalue= local_var_destination_op(index);

  copy_instruction* instr=
    new copy_instruction(rvalue,
                         to_data_type(lvalue.type()),
                         lvalue);

  emit(instr);
}
```

9.2.11 Arithmetic Bytecodes

All arithmetic bytecodes perform their computation on the operand stack and operate on `int`, `long`, `float`, or `double` types. `byte`, `char`, and `short` types have only special load/store instructions (see section 9.2.10) and are treated as `int`s for arithmetic operations.

Binary arithmetic instructions pop the two topmost stack entries, perform the computation, and push the result back on the stack. Unary arithmetic instructions pop the topmost stack element, perform the computation, and push the result back on the stack. As explained in detail in section 9.2.9.2, the operand stack is represented with temporary variables. For example, the following code is generated for `iadd`:

```
int tmp_a;
int tmp_b;
int tmp_c;

tmp_c = tmp_a + tmp_b; /* Stack effect: ... tmp_a tmp_b => ... tmp_c */
```

The temporary variables `tmp_a` and `tmp_b` represent the (symbolic) two topmost stack elements before `iadd` is executed. The result is stored in `tmp_c`, which is the new (symbolic) top-of-stack entry after `iadd` executed. The following code translates `iadd` to SUIF code:

```
void CodegenVisitor::iadd(u4 ci) {
  // Stack effect: ... value1 value2 => ... result
  source_op value2= pop_source_opnd();
  source_op value1= pop_source_opnd();
```

[31] The actual code factors out common functionality with method calls. For easier reading, we "inlined" these function calls.

```
  desination_op result= ipush_destination_opnd();

  binary_arithmetic_instruction* instr=
    new binary_arithmetic_instruction(k_add,
                                      value1, value2,
                                      to_data_type(result.type()),
                                      result);
  emit(instr);
}
```

The `iinc` bytecode increments a local variable by an integral constant. For example, `iinc 2 #42` generates the following SUIF code:

```
int lvI_2;

lvI_2 = lvI_2 + 42;
```

9.2.12 Control Transfer Bytecodes

The JVM instruction set offers unconditional jumps, a limited form of indirect jumps (`jsr`/`ret`), conditional branches, and table-based multi-way branches. We discuss each of them in the following.

The `goto` bytecode is used for unconditional jumps and directly represented with a SUIF `jump_instruction`. A `jump_instruction` takes a `code_label_symbol`, which represents the jump target. Code label symbols are identified by their name, which must be unique within every procedure. We use the bytecode index as part of the name mangling to obtain a unique name: `label_`*index*. Since the target is known statically (i.e., during compile time), the label can directly be generate. We know that a corresponding `label_location_instruction` (which marks the jump target) exists in the code, which is generated at the beginning of each basic block (see section 9.2). Note that since SUIF has symbolic labels, code for forward branches is directly emitted and need not be backpatched.

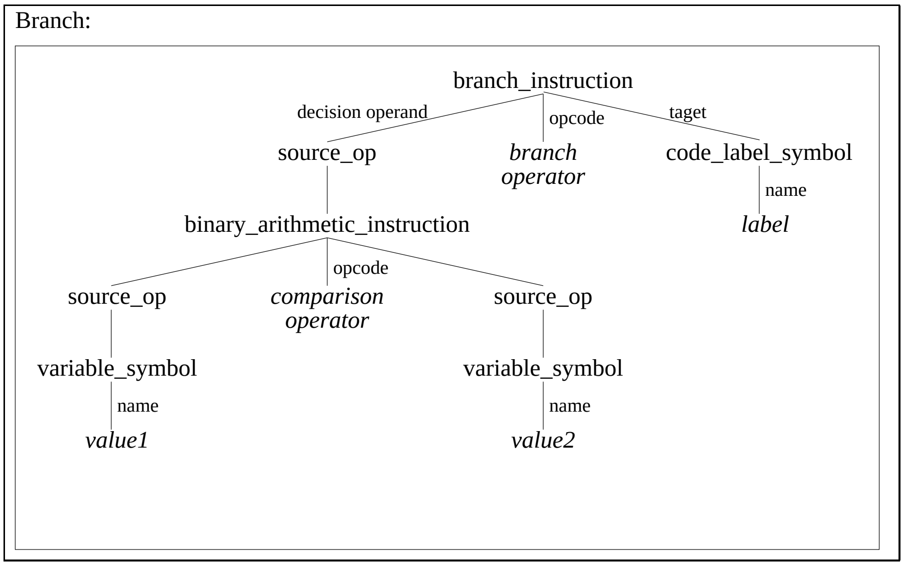

Figure 9.18: SUIF code for `if_icmp<cond>` bytecodes

The if_icmp{eq, lt, ...} bytecodes pop the two topmost integral values from the operand stack and branch if the comparison succeeds. These conditional jumps are represented with branch_instructions that have binary_arithmetic_instructions as their decision operand (see figure 9.18). Binary arithmetic instructions take two operands (which correspond to the two topmost stack entries) and a comparison opcode (e.g., is_equal_to or is_less_than). The result is a SUIF boolean value. This result is used to branch conditionally, depending on the opcode, which is either branch_if_false or branch_if_true. Since SUIF does not offer the comparison opcodes $\geq$ and $>$, the if_icmpge and if_icmpgt bytecodes are translated with a "double-negation," respectively. For example, the following code is generated for if_icmpge:

```
if (!(int)(tmp_a < tmp_b))
  goto label_x;
```

The if{eq, lt, ...} bytecodes compare the integral topmost stack entry against zero. Since SUIF does not offers a special instruction for this case, the generated code is similar to if_icmp<cond>, except that the second operand for the binary arithmetic instruction is a zero constant.

Both ifnonnull and ifnull check for a null reference. The generated code is analogous to ifneq and ifeq, except that the top-of-stack element is checked against the NULL pointer.

Both tableswitch and lookupswitch bytecodes can directly be represented with a multi_way_branch_instruction. However, our current implementation for lookupswitch uses a sequence of branch_instructions because the converter to SUIF 1.0 does only support "Dense Integer Multi-Way Branches," i.e., all multi-way branch instructions must have only finite integer case constants and each case constant must be exactly equal to the previous case constant plus one.

9.2.12.1 jsr/ret Bytecodes

The jsr/ret bytecodes have already been discussed in detail in section 8.3.2 and 8.4.2. Both bytecodes use the returnAddress type, whose values are indexes to opcodes in the current bytecode array. These indexes are used as jump targets and can be seen as virtual addresses. The code generation has to represent these virtual addresses and map them to the corresponding real addresses of the generated code. Since the bytecode array size is limited to 65535 bytes, an index can be represented with an int value.

The jsr is similar to a goto, except that in addition the virtual program counter, which represents the return address, is (symbolically) pushed on the operand stack. A ret bytecode can map the virtual return address to the real address with a switch statement. Actually, we currently use a sequence of branch_instructions to mimic the switch due to restriction of the converter (see lookupswitch above). The following code is generated for the Jasmin assembly in figure 8.8:

```
extern int JsrRet_foo(struct JsrRet *lvRef_0, int lvI_1)
  {
    int lvI_2;
    int stkRefAdr_0;
    int stk_0_int;

  label_0:
    if ((int)(lvI_1 == 0))
        goto label_14;
  label_4:
    lvI_2 = 42;
    stkRefAdr_0 = 10;              /* jsr: Safe virtual PC... */
    goto label_24;                 /*      ...and jump        */
  label_10:
    stk_0_int = lvI_2;
```

```
    goto label_27;
  label_14:
    lvI_2 = 43;
    stkRefAdr_0 = 20;               /* jsr: Safe virtual PC... */
    goto label_24;                  /*      ...and jump        */
  label_20:
    stk_0_int = lvI_2;
    goto label_27;
  label_24:                         /* ret: virtual adr -> real adr      */
    if ((int)(stkRefAdr_0 == 20))   /* If virtual adr matches...         */
        goto label_20;              /* ...jump to corresponding real adr */
    if ((int)(stkRefAdr_0 == 10))   /* If virtual adr matches...         */
        goto label_10;              /* ...jump to corresponding real adr */
  label_27:
    return stk_0_int;
  }
```

This mapping of virtual addresses to real addresses is inefficient and rather awkward. We contemplated to directly store the real address at the `jsr` bytecode and to use an indirect jump for `jsr`. The SUIF `load_address_instruction` can be used to obtain the address of a code label symbol. This address can subsequently be used in a `jump_indirect_instruction`. Unfortunately, the converter to SUIF 1.0 does not support indirect jumps ("No Jump Indirect Instructions").[32]

It is planned to lift this restriction at a future SUIF release.

This would allow the code generation to map virtual addresses to real addresses during compile time. This code cannot be represented in ANSI C, but in GNU C, which offers first-class labels (i.e., labels that can be used as values).[33] Hence, the code generation could generate SUIF code that would be translated to the following GNU C code:

```
  ...
  label_4:
    lvI_2 = 42;
    stkRefAdr_0 = &&label_10        /* jsr: Safe real PC... */
    goto label_24;                  /*      ...and jump     */
  label_10:
    stk_0_int = lvI_2;
    goto label_27;
  label_14:
    lvI_2 = 43;
    stkRefAdr_0 = &&label_20;       /* jsr: Safe real PC... */
    goto label_24;                  /*      ...and jump     */
  label_20:
    stk_0_int = lvI_2;
    goto label_27;
  label_24:
    goto *stkRefAdr_0;              /* ret: Indirect jump   */
  ...
```

[32] Indirect jumps are also not supported by `s2c`. In fact, taking the address of a label cannot be represented in ANSI C code. Instead, GNU C can be used, which offers first-class labels.

[33] A native code back end would directly generate indirect jumps instructs. It is planned that `s2c` will support indirect jumps by generating GNU C (as opposed to ANSI C).

9.2.13 Comparison Bytecodes

The comparison bytecodes `lcmp`, `fcmp<op>`, and `dcmp<op>` push an `int` result on the operand stack. Typically, they are used to construct branches for `long`, `float`, and `double` types by using the integral result at a subsequent control transfer instruction (see section 9.2.12). For example, the bytecode sequence

```
lcmp
ifle
```

realizes a conditional branch for the `long` type. It is compiled to the following SUIF code:

```
  tmp_result = 0;
  if ((int)((long long)(tmp_a - tmp_b) <= 0))
    goto label_lcmp_x;
  tmp_result = 1;
label_lcmp_x:
  if (!(int)((long long)(tmp_a - tmp_b) < 0))
      goto label_lcmp_y;
  tmp_result = -1;
label_lcmp_y:
  if ((int)(tmp_result <= 0))
      goto label_x;
```

This could be further optimized. The subtraction "`(tmp_a - tmp_b)`" is a common subexpression, which is detected and factored out by the SUIF 1.0 common subexpression elimination. The above code sequence corresponds to the semantic of the following code:

```
  if((int)(tmp_a >= tmp_b))
    goto label_x;
```

In fact, this corresponds to (a fictitious) "`ifge`" for the `long` type. We currently do not optimize this case, because our code generation uses a fixed code template for every bytecode. However, the generated code could be annotated and transformed later by a `j2s`-specific optimization pass. Luckily, comparison bytecodes are executed rather seldom.

9.2.14 Type Inclusion Tests

A *type inclusion test* determines whether one type is a subtype of another. We use the following notation to denote a subtype relationship:

$$S <: T :\Leftrightarrow S \text{ is assignment compatible to } T$$

Assignment compatibility is defined in [LY96, Section 2.6.6] and not repeated here.

Type tests are performed by `instanceof`, `checkcast`, and `aastore` bytecodes. The type T is known during compile time for both `instanceof` and `checkcast`. Furthermore, T can be any one of a class, an array, or an interface type. Since S is always the type of an object, it can never be an interface type. We discuss the various cases (differenced by T) in the following.

9.2.14.1 Subclass Test

Let's assume T is a class type. If S is an array, the type check only succeeds if T equals `java.lang.Object`. If S is a class type, the type test can traverse the inheritance graph, starting with S, towards the root class (i.e., `java.lang.Object`). It T is not encountered during the traversal, the test fails. In our implementation, this traversal can be performed by the runtime system with the class meta-data information (see section 9.3.1) generated by the compiler. In fact, an earlier version of `j2s` implemented the subclass test this way.

This scheme, described by Wirth [Wir88], is widely used for single inheritance, especially if the inheritance hierarchy can dynamically grow. Unfortunately, tests are slow because the time complexity of the lookup is bounded only by the depth of the inheritance graph. To be efficient, type test must perform in constant time. Cohen [Coh81] proposed the first practical algorithm for performing subtype tests in constant time. His algorithm is restricted to single inheritance. The COCAO Java compiler [KG97] constructs a bit matrix that contains the subtype relationship. Therefore, type checks are implemented as bit tests. Unfortunately, this scheme does not scale very well for large class hierarchies. For a program that contains more than 1000 different classes, interfaces and array types, the size of the bit matrix becomes already more than 122KB. Note that all constant-time subtype tests require the complete class hierarchy to compute an efficient representation of the subtype relationship.

Our implementation uses *relative numbering*, which allows type tests to be performed in constant time and has a very compact representation. This scheme is briefly described in [VHK97][34] and has been implemented in both the DEC SRC Modula-3 system and Fernández's Modula-3 compiler [Fer96]. Since our compiler knows the complete class hierarchy, we can compute the relative numbering during compile time. We choose relative numbering because (1) its data structure is very compact and easy to represent since it consists of only two integral values per class and (2) the actual check is reasonably fast, being comparable to Cohen's algorithms (see below).

```
global int c:= 0;

proc rn(Node: n) {
  c++;
  n.left:= c;
  foreach (child in n.children)
    rn(child);
  n.right:= c;
}
```

Figure 9.19: Relative numbering algorithm

Figure 9.19 shows the pseudo code of the algorithm. Each node in the hierarchy tree has a `left` and a `right` value. These numbers are determined by a depth-first traversal of the tree. When a node is first encountered, the current value of `c` is incremented and then assigned to `left`; when traversal leaves the node on the way up, the current value of `c` is assigned to `right`. Thus, for every node n the `left` values of all of its sub-nodes are greater than `n.left` and less or equal than `n.right` and hence the the following relation holds:

$$S <: T \Leftrightarrow T.\texttt{left} \leq S.\texttt{left} \leq T.\texttt{right}$$

This scheme allows us to implement the subclass test in constant time as a sequence of at most three loads and two comparison instructions. Cohen's Algorithms needs at most two loads, one addition, and two comparison instructions.

[34] A very similar scheme is described in [CDG96].

9.2.14.2 Interface Test

Java also requires to test if an instance of type S implements a certain interface T. If S is an array instance, this check is trivial because it only succeeds for the interfaces `java.lang.Cloneable` and `java.io.Serializable`. For class instances, the check requires the traversal of the interface `extends` relationship graph. Note that this graph is not a tree because a class can implement multiple interfaces and an interface can extend multiple interfaces.

Every class keeps a list of its interfaces in the class meta-data. Our implementation requires the traversal of the graph structure, which can be costly.[35] This check has the worst time complexity of all tests in our implementation, but it is executed very seldom. Table 9.2 shows the distribution of class, array, and interface tests for the execution of the JDK1.2beta2 `javac` compiler. The row for "Startup" gives the checks that are executed in class and interface initialization code during system startup. The test runs indicate that interface checks are at least two orders of magnitude smaller than class checks for compile runs with `javac`.

Application	`checkcast`			`instanceof`			`aastore`
	Class	Array	Interface	Class	Array	Interface	
Startup	19	0	0	14	0	0	733
`javac` Empty	1191	10	0	715	0	0	2296
`javac` Toy	9336	45	6	5058	0	0	5492
`javac` String	43302	435	326	20695	0	0	14375

Table 9.2: Statistics for type checks

Efficient techniques for type inclusion test of graphs exist [VHK97] [KVH97], but they require more complex data structures to represent the type encoding and more sophisticated algorithms to compute the encoding. If such a graph algorithm is used, the subclass and interface test can be collapsed into a single test by simply treating interface and class "inheritance" the same when computing the encoding. We refrained from this approach due to the added complexity and lack of time for the implementation.

9.2.14.3 Array Test

If T is an array type, the subtype test fails if S is not an array type as well. Subtype checks for arrays have to check the dimensions[36] and element types of the arrays. An array that contains a primitive type encodes it with an integer value. Thus, a simple value comparison can be used if both arrays contain primitive types. The relative numbering test handles the cases where both arrays contain class types. Therefore, the array test is implemented in constant time.

9.2.14.4 Implementation

All type test are implemented as calls to the runtime system. Since T is known during compile time for `instanceof` and `checkcast`, the runtime system offers different functions for the type checks depending on the type of T. The following functions are available for `instanceof`:

```
// T is a Class
JInt rt_instanceof_class(JObject *S, ClassMetadata *T);
```

[35] Actually, the runtime system implements a specific lookup cache, which keeps the results of previous interface type tests, thus speeding up the test.

[36] Java compilers statically reject checks that have different array dimensions.

```
// T is an Interface
JInt rt_instanceof_interface(JObject *S, ClassMetadata *T);
// T is a 1-D Array of a Primitive Type
JInt rt_instanceof_array(JObject *S, atype_enum Tatype);
// T is a 1-D Array of a Reference Type
JInt rt_instanceof_aarray(JObject *S, ClassMetadata *Tmd);
// T is a multi-D Array of a Primitive Type
JInt rt_instanceof_multiarray(JObject *S, int TdimsNum,
                              atype_enum Tatype);
// T is a multi-D Array of a Reference Type
JInt rt_instanceof_multiaarray(JObject *S, int TdimsNum,
                               ClassMetadata *Tmd);
```

Analogous function are implemented for `checkcast`. This way selection of the proper test is shifted from runtime to compile time.

The numbers that are used for the relative numbering check are emitted together with the global meta-data (see section 9.3.4). Note that these values cannot be computed independently during separate compilation of a certain class because the class hierarchy tree of the main class is not known at this point.

9.2.15 Exception Handling

Currently, `j2s` does not handle a raised exceptions. However, we present two possible implementation techniques for our compiler:

- A straightforward implementation can use `setjump()/longjump()` calls. The generated code is plain SUIF and can be translated to ANSI C code.

  The Toba compiler [PTB+97] uses `setjump()/longjump()` functions to implement exception handling. This technique is portable and relatively simple to implement, but it imposes a high runtime overhead for the "normal" control flow. A mapping from the actual program counter to the virtual program counter of the JVM must be maintained (to determine the code that raised the exception).[37] This is usually implemented with an additional local variable. Whenever a `try`-block is entered during runtime, `setjump()` is called and the current environment is pushed. If a `try`-block is left, the corresponding environment is popped.

- We planned to generate OSUIF code that uses OSUIF's exception handling functionality (see section 5.9), but, as of this writing, OSUIF exceptions are neither fully designed nor implemented. Besides, OSUIF exception can probably not be translated to C, but would require a fully-fledged (SPARC) back end or a translator from OSUIF to C++.

  The CACAO compiler [KP88] uses a table-based scheme, which is probably similar to the one that OSUIF will implement. CACAO translates all virtual addresses to physical addresses. Every method has its own constant area, which is accessible with negative offsets from the method pointer. Among other entries, this constants area contains the frame size of the method (for stack unwinding) and the exception table (which contains physical addresses to represent the range of the `try`-blocks and the entry points of the exception handlers).

Even though a raised exception is currently not handled, the generated code and runtime system implement all runtime checks that are required by the JVM specification (refer to section 9.2.16). The raising of an exception for a runtime check is implemented with a call to the runtime system:

[37] The GNU compiler offers first-class labels, i.e., labels whose address can be taken. They can be used to implement jump tables based on the actual program counter, but this code is not ANSI C compliant.

```
void rt_athrow_class(ClassMetadata *exceptionClass);
```

Note that the type of the exception is given as a reference to the class meta-data, because the type is known during compile time. For a full implementation, the runtime system would instantiate an exception object of this type and then raise the exception.

Exceptions that are thrown explicitly with the `athrow` bytecode are implemented with a call to the following function:

```
void rt_athrow(JObject *exceptionInstance);
```

The actual parameter contains the exception object.

9.2.16 Runtime Checks

Runtime checks are checks that must be performed during runtime and that are required by the JVM specification if a certain bytecode is executed.[38] If a runtime test fails, an exception is thrown. Section 8.3.3 lists all bytecodes that can raise an exception and hence are in need of a runtime check.

Runtime checks are performed either directly by the generated SUIF code or within calls to the runtime system. The generated code raises an exception with a call to the runtime system (see section 9.2.15). The following checks are emitted by `j2s`:

Null pointer check: This check makes sure that an object reference (which is always a temporary variable) does not contain a null pointer.[39] If the check fails, a `NullPointerException` is thrown. The following code is generated for this check:

```
  if (tmp_x != (void*)0)
    goto label_x;
  rt_athrow_class(&java_lang_NullPointerException_meta_data);
label_x:
  ...
```

Obviously, this check imposes a significant overhead during execution. Alternatively, the runtime system offers a more efficient scheme to perform the null pointer check. All Java bytecodes that require this check dereference the object reference. Hence, the runtime system can use a signal handler that catches memory accesses to the zeroth address and raise a `NullPointerException`. This scheme is much faster, but less portable. Both the runtime system and `j2s` have switches to turn off the null pointer check.

Hardware traps are the fastest (but least portable) scheme to catch `NULL` pointer dereferences. The MachSUIF extension for SUIF 2.0, which is developed by Harvard University, should allow the installation of such a user trap handler. We plan to use this scheme as soon as MachSUIF and a (SPARC) back end for MachSUIF becomes available.

Zero value check: This check is analogous to the null pointer check, except that it makes sure that an integral value (`int` or `long`) is nonzero. If the check fails, an `ArithmeticException` is raised.

Array bound check: This check makes sure that array accesses are within the bounds of the array `length`:

[38] Optimizations can reduce the number of the checks executed during runtime by moving or eliminating them, but these transformations must not change the program behavior.

[39] The null pointer is represented as a pointer to the `void_type` in SUIF. The generated C code represents it with "`(void*)0`."

$$0 \leq index < arrayref\text{->}\texttt{length}.$$

If the check fails an `ArrayIndexOutOfBoundsException` is thrown. Both the index and the array length are 32-bit two's-complement integers. The array length is non-negative. Therefore, the bounds check can be implemented with a single *unsigned* comparison of the index against the array length:[40]

```
  int tmp_index, tmp_length;
  ...

  if ((unsigned int) tmp_index) < tmp_length)
    goto label;
  rt_athrow_class(&java_lang_ArrayIndexOutOfBoundsException_meta_data);
label:
  ...
```

All other checks are implemented by the runtime system.

9.2.17 Class and Interface Initialization

A class or interface is initialized at its *first active use*, as described in the Java Language Specification [GJS96, Section 12.4.1].

A class can be initialized only after all its superclasses have been initialized. In contrast, initialization of an interface[41] does not require initialization of any of its superinterfaces. A class or interface must be initialized only once. A class or interface is initialized with a call the static method `<clinit>`. If this methods does not exist, the class can be considered to be already initialized.

The class meta-data (see section 9.3.1) maintains the boolean flag `initializeFlag`, which states if the class or interfaces has been initialized. The check for the first active use is required for certain bytecodes and is either an explicit call to the runtime system or performed as part of another call to the runtime function (see below). For explicit checks, the runtime system offers the function

```
void rt_init_class(ClassMetadata *);
```

which sets the `initializeFlag` and calls `<clinit>` for the given class (and its superclasses) or interface if the flag was not set before.

The following actions potentially require a check according to the first active use definition:

Object creation: The creation of a class instance requires a `new` followed by a method call to `<init>`, to bring the object into a proper state. Since the specification states that a class T must be initialized if "a constructor for class T is invoked," the initialization check is performed inside the `<init>` method (see below), and not at the `new` bytecode.

For array instances, the element type of the array must be initialized if it happens to be a non-primitive type. Hence, only `anewarray` and `multianewarray` (and not `newarray`) require a check, which is performed by the runtime system. (The array itself does not require initialization because it has no `<clinit>` method.)

Instance method call: As stated before, a call to the `<init>` method, which is coded with

$$\texttt{invokespecial } T\texttt{.<init>()V},$$

[40] This "trick" has already been implemented by early Pascal compilers. CACAO [KG97] uses it as well.

[41] Even though interfaces cannot contain code at the Java language level, they can have a `<clinint>` method generated by the compiler.

requires an initialization check for class T. Since this check must (conceptually) be performed at every call site to `<init>`, we can also emit it at the preamble of every `<init>` method without changing the semantics of the initialization process. This approach keeps the code size smaller.

For every non-`<init>` instance method call to class T, the "this" reference is an object of class S and $S = T \vee S <: T$. If the "this" reference is `null`, an exception is thrown. The method call is not performed and hence the class must not be initialization. Since the JVM verifier enforces that a non-`null` reference points to a fully constructed object, the `<init>` method for S and all of its superclasses must have been called. Hence S and all of its superclasses (which includes T) must have been already initialized. This means that a non-`<init>` instance method never requires a class initialization check.

Instance field access: The initialization check is not necessary because whenever an instance field declared in class T is accessed, the object must be of class S and $S = T \vee S <: T$. Hence T must have been already initialized (same reasoning as for instance method calls; see above).

Static method call: If a static method of class T is called, T (and its superclasses) must be initialized. Analogous to `<init>`, the actual check is performed at the beginning of every static method and not at the call site.

Static field access: *Constant static fields*, i.e., fields that are `final` and have a `ConstantValue` attribute [LY96, Section 4.7.3], must not be checked because their values are compile-time constants and are directly inserted into the generated code.

A non-constant static field access requires the initialization of its defining class or interface. This means that the code generation emits an initialization check for `getstatic` and `putstatic` bytecodes. Our compiler implements a minor optimization by omitting the check inside of those `<clinit>` methods whose class is identical or a subclass of the defining class of the field access.

To summarize, the runtime system performs initialization checks for `rt_anewarray()` and `rt_multianewarray()` functions, whereas explicit calls to `rt_init_class()` are generated by `j2s` for `<init>` methods, static method calls, and static field accesses.

9.2.18 Implementation

The code generation is implemented with bytecode visitors (see figure 9.20). The abstract `CodegenVisitor`

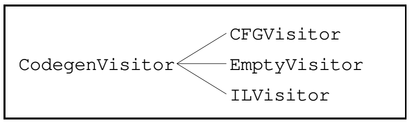

Figure 9.20: Bytecode visitors for code generation

contains an `instruction_list` that holds the generated code for the currently compiled bytecode array (`ILVisitor`) or basic block (`CFGVisitor`):

```
class CodegenVisitor : public CodeVisitor {
protected:
  instruction_list* _ilist;

  Method* _method; // current Java method
```

```
  file_block* _fb; // current file block (of Java method)
  procedure_definition* _pb; // current procedure definition (of Java method)
  ...

  // append 'instr' to '_ilist'
  void emit(instruction* instr);
  ...
};
```

The `ILVisitor` is used to generate code in instruction list form. As an example, the code to generate the corresponding SUIF instructions for `istore` and `iadd` is given in section 9.2.10 and 9.2.11, respectively.

The `CFGVisitor` generates CFG code. For the CFG form, every basic block is compiled to an instruction list, which is put into a `cfg_node`. Control flow is modeled explicitly with edges between the CFG nodes. The code generation for control transfer bytecodes simply constructs these edges. The `EmptyVisitor` does not generate any code, i.e., the method body is empty. This visitor is useful for debugging.

Analogous to the `MergeVisitor` (see section 8.4.5), a basic block iterator (`CodegenIterator`) is used to traverse the bytecode array and call the visitor.

9.3 Meta-Data Structure

The *meta-data* for a class file serves different purposes. Part of the meta-data is required for reflection (introduced with JDK 1.1), for interface dispatch (with our current implementation; see section 9.2.8.3.2), and to support of the Java Native Interface (JNI). Moreover, it is used to share common data structures between the runtime system and the compiler. The meta-data layout has been designed with Andrew Rutz, who also implemented `j2s`'s runtime system [Rut98].

The actual representation of the meta-data has a wide design space. On one extreme, the compiler could generate no meta-data at all. This would force the runtime system to load the corresponding class file to extract the meta-data information. This approach requires the runtime system to implement a (rudimentary) class loader and causes undesirable startup overhead and consistency headaches. Bothner [Bot97] describes a possible solution that incorporates the class file directly into the executable as static data:

```
static const char X_classfile[] =
  "\xCA\xFE\xBA\xBE...";
```

The runtime systems executes the class file information at program startup. This technique eliminates the consistency problem of the previous approach, but the other shortcomings still remain.

It is much better if the compiler directly generates internal meta-data. `j2s` generates the meta-data at a high level of abstraction similar to the resolved class file representation of the compiler (see section 8.2).

Our implementation uses SUIF `group_type`s to code the structure of the meta-data. The meta data is incorporated into the generated SUIF file as static data with a `variable_definition` that contains a `multi_value_block` (see section 4.4) to initialize the group type. The multi value block contains `constant_value_block`s and `expression_value_block`s to initialize the fields that make up the group type. A constant value block initializes an integral constant, whereas an expression value block initializes a (non-`NULL`) pointer reference.

The generated meta-data is a compromise between the `j2s` compiler, which strives towards code that is easy-to-generate, and the runtime system, which wants fast and easy access to the meta-data information. The meta-data contains only information that cannot be inferred from other information.

For example, the defining class of a method or field is not coded directly because it can be inferred with a traversal of the existing meta-data. Static fields and methods are identified with a bit flag even though a direct access to a boolean variable would have been more convenient and faster for the runtime system.

In general, the runtime system is free to generate its own meta-data structure that contains `j2s`'s meta-data as a subset, but represents `j2s`'s meta-data in a form more suitable for the runtime system. On the other hand, transforming the meta-data can lead to a longer system startup.[42] In fact, the runtime system generates its own meta-data, called *runtime meta-data* (see section 9.4 for a detailed discussion).

In the following, we describe the meta-data that is generated by the code generation for classes, methods, and fields. In addition, *global meta-data* is needed to "glue" the class meta-data together. These structures are generated during compile time and must not be confused with the runtime meta-data, which is built at runtime during system startup (see section 9.5).

9.3.1 Class Meta-Data

`j2s` generates separate class meta-data[43] for every class file that it compiles. Note that this implies that `j2s` does not generate meta-data if it encounters an array or primitive type during the compilation of a class file. The meta-data for an array is part of the global meta-data (see below). The runtime system is responsible for generating the meta-data for primitive types (which is used for reflection and JNI). The following data structure is used to represent the class meta-data:

```
struct class_meta_data { char *classname;
                         void *jclass;
                         int typeFlag;
                         int vtblNum;
                         void (*)() (*vtbl)[1];
                         int accessFlags;
                         int initializeFlag;
                         int instanceSize;
                         struct class_meta_data *superClass;
                         int superintfsNum;
                         struct class_meta_data *(*superintfs)[1];
                         int methodNum;
                         struct method_meta_data (*methods)[1];
                         int fieldNum;
                         struct field_meta_data (*fields)[1];
                         int stringNum;
                         char *(*strings)[1]; }
```

The `classname` holds a C string with the fully qualified name of the class or interface as described in [LY96, Section 4.2]. The `jclass` pointer provides a hook for the runtime system to associate additional, internal information to the meta-data structure. It is initialized to `NULL`. Of course, the runtime system is not required to make use of it.

The `typeFlag` field contains three bit-flags that determine whether the class meta-data structure represents a class, an interface, or an array. (The runtime system can use additional bit positions for its private encoding.)

The class access and modifier flags are kept in the `accessFlags` field [LY96, Section 4.1].

[42] The runtime system can try to remedy this problem by transforming the meta-data of a class "on demand," i.e., when the class is referenced for the first time.

[43] The name "class" meta-data is slightly misleading because it is used for interfaces and arrays as well.

The vtable (`vtbl`) is kept in a separate table that contains the function pointers of the compiled Java methods. The runtime system copies the vtable into the runtime meta-data at system startup (see section 9.4).

`initializeFlag` contains a boolean value that encodes whether the class has been already initialized (see section 9.2.17). `j2s` sets the flag for classes that do not require static initialization (because in a sense these classes are already completely initialized at compile time). During runtime both the runtime system (with a call to `rt_init_class()`) or the code generated by the compiler can potentially set the flag.

The `instanceSize` field contains the full size (in bytes) that an instance for the corresponding class requires. The field is meaningless for interfaces and arrays.

The superclass is kept in `superClass` and is represented with a pointer to the superclass's meta-data. For interfaces and `java.lang.Object`, this field is set to `NULL`. The superinterfaces are kept in a separate table (`superintfs`), which contains pointers to other class meta-data.

`methods` points to a table that contains the static and instance methods that belong to the class meta-data. Each method is represented with a separate method meta-data entry (see section 9.3.2). An analogous structure exits for the static and instance `fields` (see section 9.3.3).

Finally, the compile-time strings of the class are kept in a table of pointers to C strings that contain the original UTF-8 encoding of the Java string. These strings represent the `CONSTANT_String` entries in the constant pool of the corresponding class file and must be "interned" by the runtime system as specified in [GJS96, Section 3.10.5].

9.3.2 Method Meta-Data

The method meta-data is used to represent a static or instance method of a Java array, interface or class type. `j2s` generates the following data structure:

```
struct method_meta_data { char *simpleName;
                          void *jmethod;
                          int accessFlags;
                          char *descriptor;
                          int vtblOffset;
                          void *fnPtr;
                          void *nmPtr;
                          int exceptionsNum;
                          struct class_meta_data *(*exceptions)[1]; }
```

The `simpleName` contains the name of the method as it is given in the class file. Note that this name can contain UTF-8 characters and usually differs from the mangled name that `j2s` generates for its method symbols.

`jmethod` is analogous to the `jclass` field of the class meta-data. The method's access and modifier flags are kept in `accessFlags`.

The `descriptor` field contains the method descriptor as specified in [LY96, Section 4.3.3]. The `vtblOffset` contains the methods's offset in the vtable for non-private instance methods (-1 otherwise). (The generated code does not access this field; it is required for JNI calls through the runtime system). The vtable offset is given in multiples of "`sizeof(void (*)())`."

The pointer to the function that holds the generated code for the methods is kept in `fnPtr`. Non-native methods have a pointer to the actual code, whereas native methods contain a pointer of the corresponding stub function (section 9.2.3.3 gives a detailed description and shows an example of the generated meta-data). The pointer to the actual native method are kept in `nmPtr`; it is used for JNI calls. Non-native methods contain a `NULL` pointer.

The exceptions that the method can throw is kept in a separate table and referenced with the `exceptions` field.

9.3.3 Field Meta-Data

The field meta-data is used to represent a static or instance field of a Java interface or class type. `j2s` generates the following data structure:

```
struct field_meta_data { char *simpleName;
                         void *jfield;
                         int accessFlags;
                         char *descriptor;
                         int offset;
                         void *address; };
```

The first three entries are analogous to the method meta-data. The `descriptor` contains the field descriptor as described in [LY96, Section 4.3.2]. The field `offset` for instance field is given in multiples of bytes relative to the beginning of the object layout (see section 9.2.5.1). Static fields contain -1. Finally, the `address` field contains a pointer to the storage location of a static field's data; it is used for static field accesses through JNI. Instance fields have a `NULL` entry.

9.3.4 Global Meta-Data

The global meta-data consists of data that must exit only once per executable and hence cannot be included into an object file that contains compiled class file code for separate compilation. Furthermore, part of the global meta-data depends on the main class that the executable is build for. We discuss each part of the global meta-data in the following sections.

9.3.4.1 Transitive Closure of Classes

The array `j2s_AllClasses` is used to represent the transitive closure of all Java classes and interfaces that the main class references. A class or interface is represented with a pointer to the corresponding class meta-data. By convention, the first entry in the table is a reference to the generic array meta-data (see below). For example, the class `HelloWorld`

```
class HelloWorld {
  public static void main(java.lang.String[] args) {
    System.out.println("Hello world!");
  }
}
```

generates the following transitive closure:[44]

```
extern struct class_meta_data ARRAY_meta_data;
extern struct class_meta_data java_lang_Cloneable_meta_data;
...

int j2s_AllClassesNum = 352;
struct class_meta_data *j2s_AllClasses[352] =
  {
    &ARRAY_meta_data,
    &java_lang_Cloneable_meta_data,
```

[44]The complete closure consists of 354 classes and interfaces for JDK1.2beta3.

```
    &java_lang_Object_meta_data,
    ...
    &HelloWorld_meta_data,
    ...
    &sun_tools_jar_JarException_meta_data
  };
```

Note that the meta-data interface to the runtime system consists only of the `j2s_AllClasses` and `j2s_AllClassesNum` external variables. The other components of the class, method, and field meta-data are (transitively) accessible through the `j2s_AllClasses` array.

9.3.4.2 Array Meta-Data

The array meta-data represents the meta-data for all Java arrays, as opposed to classes or interfaces, which require a separate class meta-data entry for every class or interface. We can use a single class meta-data entry for all arrays types because they differ only with respect to their element types. The vtable, for example, is identical for all arrays.

This property is especially convenient for separate compilation, because it avoids the generation of (potentially) redundant array meta-data. For instance, let's assume that the transitive closure for the compilation contains two classes that both use a `String[]` array. Separate compilation for both classes cannot know if another class in the transitive closure uses a `String[]` as well. Hence, the code generation must conservatively generate `String[]` meta-data twice. Multiple array meta-data would increase the code size and would complicate the runtime system.

Even though `j2s` generates only a single meta-data entry for all arrays, the runtime system generates separate runtime meta-data for every array type. (This is transparent to the code generation.) The runtime system can generate separate meta-data based on the element type of the array because `j2s` passes the element type as an argument for every array instance creation (see section 9.2.6).

The entries for the array meta-data correspond to the behavior described in the Java specification [GJS96, Section 10.7], except that

- the (implicit) `length` field is not represented in the field meta-data because it cannot be directly accessed via JNI.
- the vtable is identical to `java.lang.Object`'s vtable, which means that the `clone()` method is not overridden. Overriding the `clone()` method is performed by the runtime system at system startup (see section 9.2.8.3.1 for further explanation).

9.3.4.3 Size of Data Structures

The following global external variables contain the size of the class, method, and array meta-data:

```
extern int j2s_ClassMetadataSize;
extern int j2s_MethodMetadataSize;
extern int j2s_FieldMetadataSize;

int j2s_ClassMetadataSize = 72;
int j2s_MethodMetadataSize = 36;
int j2s_FieldMetadataSize = 24;
```

At system startup, the runtime system makes sure that the size of its own representation is identical with the size of `j2s`'s representation. This check catches simple interface mismatches between the data structures. It was especially useful during the early development stage when the meta-data format had to be modified more frequently.

9.4 Runtime System

The runtime system is an independent module, which is built as a library and included at the link step to obtain the final executable. The Java language requires a sophisticated runtime system. Even though we do not support all Java features, implementing the JNI functionality is already non-trivial. The runtime system has been implemented by Andrew Rutz as his Master's thesis project and is described in [Rut98].

In the following, we briefly describe the purpose of the runtime meta-data and the interface between the runtime system and `j2s`.

9.4.1 Runtime Meta-Data

The runtime system maintains its own internal data-structures that are not visible to the generated code and that are generated during system startup (see section 9.5). As mentioned before, part of these internal data structures is a more appropriate representation of the meta-data generated by `j2s`, called *runtime meta-data*. The runtime system has to access this meta-data quite frequently for type checks, interface dispatches, JNI calls, etc.

Every object has a reference to its corresponding runtime meta-data (see figure 9.1). The runtime system sets up the reference to the runtime meta-data at object creation.

Ideally, the structure of the runtime meta-data should be hidden completely from `j2s`. On the other hand, the generated code needs a fast access to the vtable for every instance. To hide the runtime meta-data, every instance could theoretically contain two fixed references: One to the runtime meta-data and another on to the vtable. Unfortunately, an efficient implementation does not allow the luxury of having an additional slot for the vtable per object. Our implementation copies the vtable from the class meta-data into the runtime meta-data (see figure 9.21), which achieves fast vtable access and minimal space overhead per object.

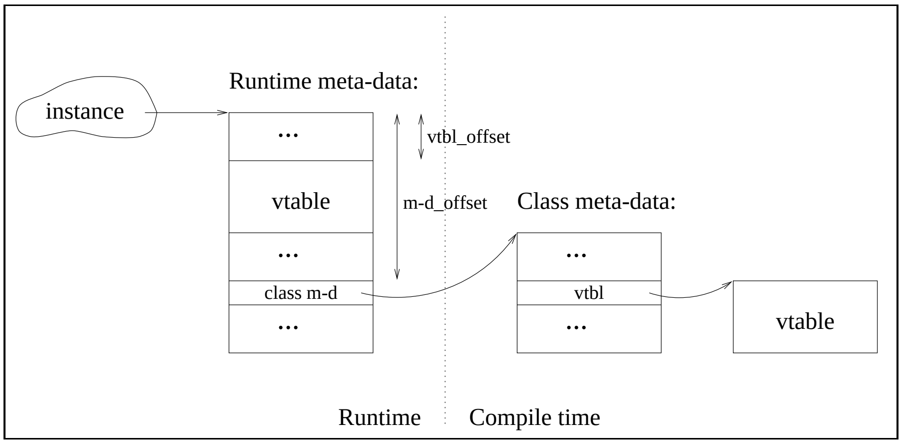

Figure 9.21: Runtime meta-data (with vtable)

9.4.2 Interface

We describe here only the interface between the runtime system and `j2s`. We aimed to keep the interface as small and simple as possible, but to generate efficient code (e.g., fast dispatch and field accesses without unnecessary indirections) it was necessary to expose certain implementation details of the generated code to the runtime system. The interface consists of the following components:

Runtime functions: The runtime system offers various functions for the code generation, which are described in section 9.2. All calls to the runtime system use the C calling convention.

Object and field layout: The runtime system and `j2s` agree on the following facts:

- For all instances, the pointer to the runtime meta-data is at offset zero. Other slots in the object layout are instance fields, whose offsets can be determined with the field meta-data. The array layout (see section 9.2.5.1) is fully exposed to the runtime system to allow fast access for JNI calls and for the `rt_aastore` function (see section 9.2.7.3).
- The data types for fields of class instances are taken from the Java specification, except for `boolean`, `byte`, `char`, and `short`, which take the size of an `int`. This encoding saves the introduction of casts for the code generation because all values that are stored into fields (with `putfield/static`) with the types `boolean`, `byte`, `char`, or `short` are 32-bit wide.
- The data types for array elements are taken from the Java specification, except for `boolean`, which has the same size as a `byte`.[45]

Values: The JVM specification does not define concrete values for booleans and the `null` reference. We consistently use the following values:

- The boolean value for `true` is represented with 1; `false` is represented with 0.
- A `null` reference is identical to a C `NULL` pointer.

Meta-Data: The meta-data generated by `j2s` is part of the interface (see section 9.3). The following constraints are imposed upon the runtime system:

- The `initializeFlag` in the class meta-data can be set by both the runtime system and the generated code. Hence, the runtime system is not allowed to copy this field into its internal meta-data. (The currently generated code does not modify the `initializeFlag`.)
- The vtable in the class meta-data and the runtime meta-data must be identical (see figure 9.21). This means that the runtime system has to patch both tables if it changes a vtable slot.

The runtime meta-data exposes the following parts of its structure to the code generation:

- The offset of the beginning of the vtable (which is identical with the first vtable slot; see figure 9.21, `vtbl_offset`). This offset is needed to generate more efficient class dispatch that saves an additional indirection compared to dispatch code that uses the vtable in the class meta-data (refer also to section 9.2.8.3.1).
- The offset of the beginning of the pointer to the class meta-data (see figure 9.21, `m-d_offset`). The currently generated code never references the class meta-data, but this might change.
- The sizes of the internal data structures that the runtime system uses to access the class, method, and field meta-data. To prevent interface mismatches, `j2s` checks during compile time if these sizes are identical to its own representation.

[45] The JDK does this as well. It is a good compromise between object size and fast access.

9.5 System Startup

Compilation of the main class X generates the object file X_startup.o. This file contains the main() function, which is the entry point of the executable.

```
extern int main(int, char **);
extern struct class_meta_data X_meta_data

extern int main(int argc, char **argv)
  {
    rt_Init();
    rt_Run(&X_meta_data, argc, argv);
    rt_Exit();
  }
```

The main() function initializes the runtime system with a call to rt_Init(). This includes generation of the runtime meta-data (from the compile-time meta-data), string internalization, etc. Next, rt_Run() is called, which passes the class meta-data (X_meta_data) of the startup Java class and the command-line arguments. The runtime system searches the meta-data of the startup class for a static method with the descriptor

main([Ljava/lang/String;)V.

If this method is found, the runtime system builds a String[] array that contains the command-line arguments and calls this method. Otherwise, the runtime system terminates the program with an error message. rt_Exit() currently only optionally prints out statistical information that has been gathered during the program run.

Chapter 10

Testing

Testing is an important part of every software development process [Som92]. We tested `j2s` with the following (glass-box) tests:

- We ran small, isolated class files specifically written to test certain features of the compiler. One test lets the compiler generate code for every bytecode, which should yield a good path testing coverage.

  Most tests are designed to pass `j2s` without causing an error. We either inspected the generated C code or ran the generated executable to verify correct behavior.

  We also used the Jasmin assembler (see below) to generate class files that do not conform to the JVM specification and hence must cause `j2s` to terminate with a well-defined error.

- We ran bigger tests that consist of several class files. These tests range from toy applications (such as `hello world`) to "real-world" applications (such as `javac`). Actually, a Java `hello world` application is quite complex because various classes must be initialized, strings must be interned, I/O has to work for files and standard output, etc.[1] Among others, we tested `javac`, `javadoc`, and `javap`, which are part of the JDK distribution. The `javac` test is the most valuable and is described in more detail in section 10.1.

- As a stress-test, we compiled all class files that are shipped with Sun's JDK release. We achieved our goal to compile all class files without causing `j2s` to fail.

10.1 Testing of `javac`

Testing `javac` combines two advantages: First, it is easy to verify the correct test result (see below). Second, the compiler covers exactly the subset of Java that we support. Actually, `javac` is multi-threaded because it creates a separate thread for every class it compiles, but, luckily, it can be run as a single-threaded application by ignoring the thread primitives. Furthermore, `javac` can raise an exception to indicate a compile error. Since we compile only correct Java programs, no exceptions are raised.[2]

The `javac` test consists of:

[1] One can draw a parallel to Smalltalk systems, which test a wide range of their functionality by simply printing the result of an integer addition [Kra83, page 57].

[2] Actually, there is a single program point where `javac` throws a `ClassNotFound` exception. We had to change three lines in `sun.tools.java.Environment` to work around this problem.

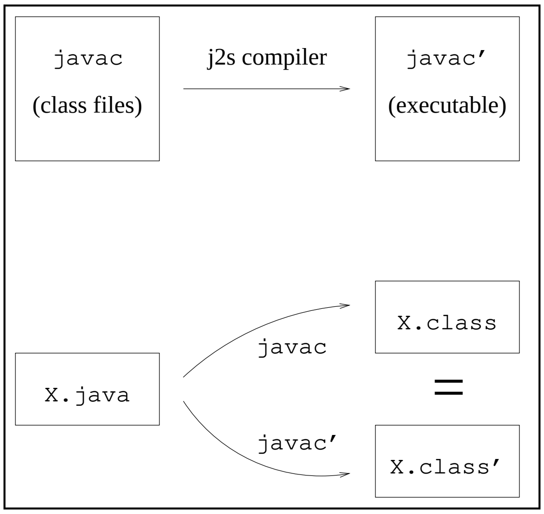

Figure 10.1: `javac` test case

1. `j2s`'s compilation of the 538 class files that constitute the transitive closure of `javac` (see top of figure 10.1). The resulting executable is called `javac'` in the following.

2. compilation of a set of Java source files with both the original `javac` and the `javac'` compiler. The generated class files must be identical (see bottom of figure 10.1).

We compiled the Java source files of JDK1.1.5 and JDK1.2beta2. (The JDK1.2beta2 release contains only part of the source files and most of them are similar to the corresponding JDK1.1.5 sources.) We successfully compiled all sources, i.e., identical class files were generated by `javac'` and `javac`.

Since the source code of the `javac` compiler is part of the JDK source code distribution, it is possible to bootstrap `javac`. Figure 10.2 illustrates the bootstrapping with *T-diagrams* [ASU86]. The given components are `javac` (source code and pre-compiled class files) and `j2s`, which we assume to be already compiled for machine architecture N.[3] The actual bootstrapping first generates `javac'` (second row in figure 10.2) and then bootstraps `javac` with `javac'` (third row in figure 10.2).[4]

10.2 The *Jasmin* Assembler

The *Jasmin* assembler[5] [MD97] allows to generate class files that cannot be generated with a Java compiler. We use Jasmin to generate class files that

- are impossible to generate with any Java compiler (e.g., irreducible control flow, code that does not pass the verifier, or `try`-blocks that encapsulate a certain code region).

- cannot be generated by a specific compiler (e.g., `javac`). The code in figure 8.8, for example, cannot be generated with `javac` (and most other compilers) because `jsr/ret` bytecodes are usually only generated for a `try-finally` statement [LY96, Section 7.13].

[3] Currently N=SPARC since the runtime system contains SPARC assembly.

[4] In fact, a successful bootstrap requires that both `javac` and the bootstrapped `javac` are identical.

[5] Homepage: `http://www.mrl.nyu.edu/meyer/jvm/`.

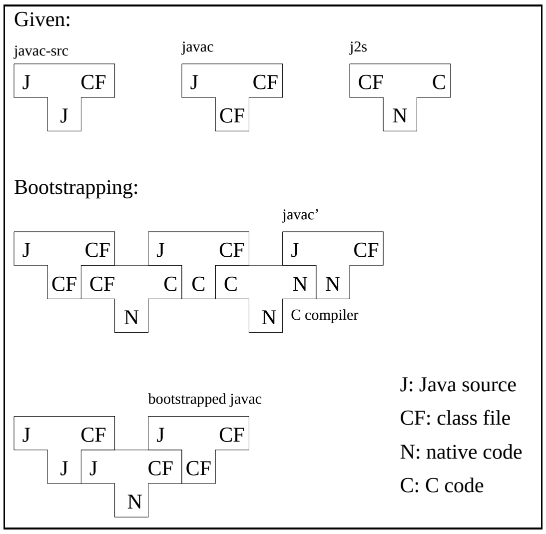

Figure 10.2: Bootstrap of `javac`

We used Jasmin to generate small test cases, which turned out to be very convenient to pinpoint "trivial" bugs in the early development phase.

Chapter 11

Performance Evaluation

In the following, we describe our set of benchmark programs, and give a brief performance evaluation of `j2s` relative to the JDK Virtual Machine implementation.

11.1 Benchmark Programs

Table 11.1 shows the benchmarks that we used. All of them are *macro-benchmarks*, as opposed to *micro-benchmarks*, which often give misleading information about the performance of real-world applications. To give a sense of the scale of the benchmarks, we show the size of every application, which consists of the runtime system and the generated code for the transitive closure reachable by the main class of the application. The size of the runtime system is about 0.5 MB (which includes `libjava`, `libjvm`, and C/C++ standard libraries).

Benchmark	Description	Code size/MB
`javac`	Java source to bytecode compiler	5.42
`javap`	Java bytecode disassembler	5.09
`sim`	Sylvia Dieckmann's GC simulator[1]	3.3
`compress`	File compression with LZW	7.1[2]
`db`	Simple address database	7.1[2]

Table 11.1: Benchmarks

11.2 Results

Figure 11.1 graphically visualizes the overall speedups that we achieved for the benchmarks. Every benchmark was executed with several different inputs. (See appendix A for more details.) The speedups indicate how much faster the executable generated by j2s runs in comparison to the JDK1.2beta2 VM. When generating an executable with j2s, the following (orthogonal) options are available:

1. The runtime system can be optimized (O) or left unoptimized (U).

[1]Sylvia Dieckmann's homepage: `http://cs.ucsb.edu/~sylvie`.

[2]The code size includes the SPEC harness application and the SPEC compress, jess, db, mtrt, and jack benchmarks.

2. The generated C code can be optimized with the SUN C compiler (O) or left unoptimized (U). (See below for further explanation.)

3. The Boehm-Demers-Weiser garbage collector can be enabled (G) or disabled (N).

We show the resulting speedups for all combinations of the above options. For the bars in figure 11.1, the first letter indicates if the runtime system was optimized (O) or unoptimized (U), the second letter indicates if the generated C code was optimized (O) or unoptimized (U), and the last letter indicates if garbage collection was enabled (G) or not (N).

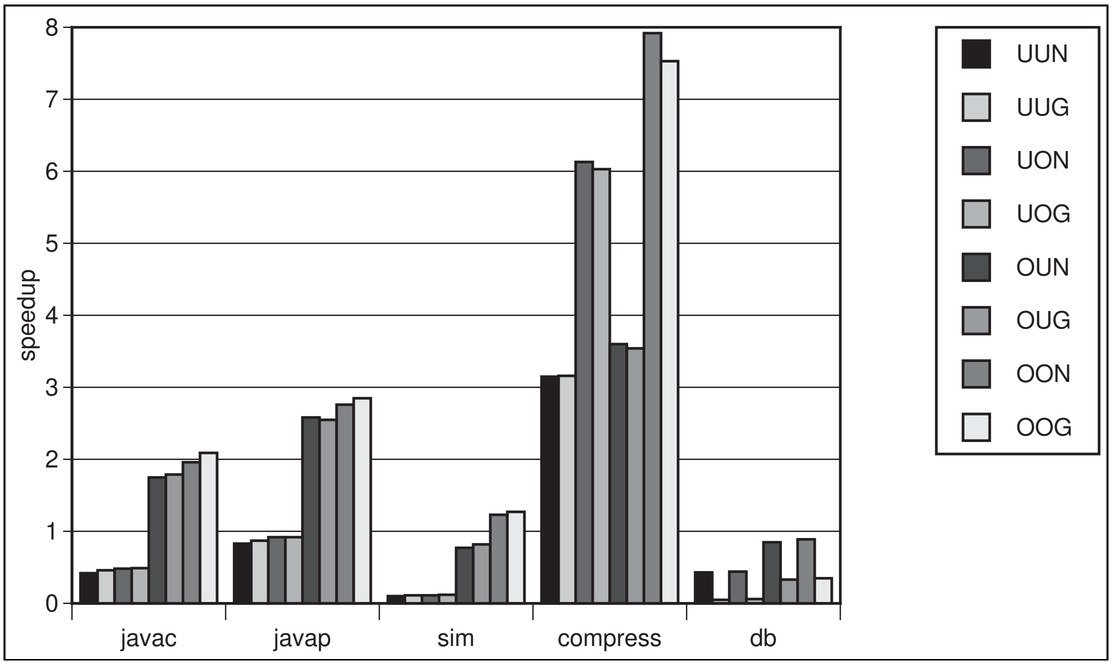

Figure 11.1: Speedups

As can be seen, garbage collection has only a minor performance impact. The same is true—to a lesser extend—for optimizing the generated C code, except for the compress benchmark. On the other hand, the performance of the runtime system plays a significant role in the overall performance of the compiled code. For example, figure 11.2 shows how much time is spend in which part of the code during the execution of "`javac String.java`" with an unoptimized and optimized runtime system. (The garbage collector was enabled and the C code optimized.) With the unoptimized runtime system, 89% of the execution time is spend in the runtime system compared to only 54% for the optimized runtime system.[3]

We made five runs on a lightly loaded dual-processor UltraSPARC machine and took the median execution time. The runs were very similar, differing by less than 5% of the execution times. We compare `j2s`'s performance with the JDK VM, because it is the reference implementation for the JVM specification and most other compilers compare their performance to it.[4]

[3]Since currently a lot of time is spent in the runtime system, optimizing the C++ code of the runtime system with `-O2` has a significant performance impact.

[4]One can also compare Java implementations on different platforms, however, this approach suffers from similar inaccuracies than Relative MIPS.

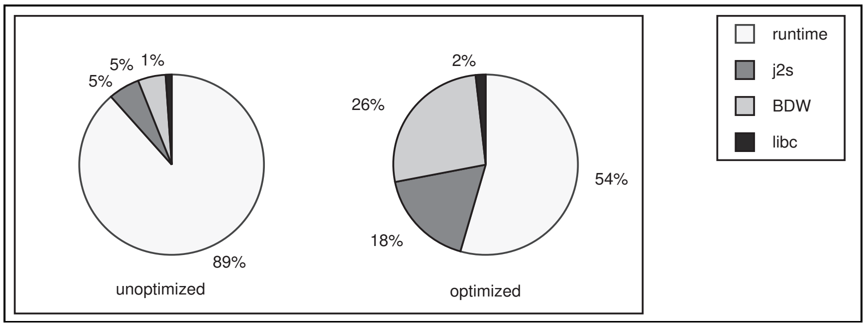

Figure 11.2: Execution profile of `javac` executable

Optimized executables are generated with the following steps:

1. `j2s` generates the SUIF 2.0 code for all class files that consist of the transitive closure of the application.

2. The converter translates SUIF 2.0 to SUIF 1.0. No optimizations are applied in the SUIF 2.0 representation.

3. Various `porky1` optimizations are applied to the SUIF 1.0 code: constant folding (`fold`), simple optimization on unstructured control flow (`ucf-opt`), copy propagation (`copy-prop`), constant propagation (`const-prop`), forward propagation (`forward-prop`), and simple dead code elimination (`dead-code`).[5]

4. The `structure` optimization pass is run, which "builds structured control flow where it can be done in a natural way." The pass uses a simple algorithm and is rather limited in its capability to recover high-level control flow. Figure 11.2 depicts the impact of the `structure` pass on the generated control structures.

Construct	Number of constructs in C source	
	w/o `structure`	w/ `structure`
`goto`	25928	6329
`if` w/o `else`	21415	20106
`if` with `else`	0	1128
`do`/`while`	0	8

Table 11.2: Impact of `structure` optimization pass on `javac`

5. `s2c` translates SUIF 1.0 to C. The C code is compiled with `-native -xO2 -xlibmil` by the SUN Workshop 3.0.1 C compiler. It is hard to tell which SUIF optimization passes are subsumed by the C compiler. Usually a back end implements only target-specific optimizations (such as instruction scheduling and register allocation), but not general-purpose optimizations.

[5] We could not use common subexpression elimination (`cse`) because a bug in this pass eliminates type casts, which subsequently causes `s2c` to generate illegal C code.

The optimized runtime system was compiled with `-native -O2 -xlibmil` by the SUN Workshop 4.2 C++ compiler. Furthermore, assertions were disabled and various optimizations (e.g., lookup caches, type inclusion tests with relative numbering) enabled.

As of this writing, we are not finished tuning the runtime system; further speedups can be expected. Our goal is to achieve a runtime performance close to Toba [PTB+97], which has a speedup of about 3.3 for `javac` compared to the JDK1.0.2 VM on SPARC.[6]

The execution time reported for `j2s` seems to be not quite fair because `j2s` does not implement exception handling, threads, and locks. Each of these features usually introduces a certain runtime overhead. We argue that this runtime overhead is small particularly for the benchmarks introduced in section 11.1 because

- they do not require threads, i.e., they are single-threaded. These kind of applications can profit from a lightweight synchronization scheme, such as *thin locks* [BKMS98], which introduces only a small synchronization overhead. Thin locks could be implemented by the runtime system with only minor modifications to the `j2s` compiler.

- they do not raise an exception. An efficient exception handling implementation, such as a table-based scheme, does not introduce any overhead if a `try`-block is entered or left, but only if an exception is actually raised. Since the applications never raise an exception, our implementation mimics such an efficient exception handling scheme.

For these reasons, we can expect that a fully-fledged `j2s` implementation yields very similar results for our benchmarks. Note that our benchmarks do not indicate how multi-threaded applications or applications that throw many exceptions would perform.

[6]Toba's C code was optimized using the SUN C compiler with `-xO4 -xcg92`.

Chapter 12

Related Work

This chapter gives a brief overview of existing Java compilers. The `j2s` compiler is compared with other compilers that aim for a similar application domain. Moreover, decompilers are covered because they usually have to deal with high-level control structure recovery, which can also be applied to Java bytecodes.

12.1 Other Java Compilers

An astoundingly broad variety of Java compilers has been implemented by the research community. These compilers expect Java source and/or Java bytecode and generate C (as a portable target language), native code, or intermediate representation for a compiler system. Implementation techniques range from off-line to JIT compilation (see section 7.2 for a more detailed discussion of these techniques).

Among the Java compilers that we looked at are:

- Toba is a system for generating stand-alone Java applications developed at the University of Arizona [PTB$^+$97]. It includes a Java bytecode to C compiler (off-line compilation) that covers the whole Java language specification.
- Bothner is developing a `gcc`-based Java compiler [Bot97]. A new gcc front end, `cc1java`, translates Java bytecode or Java source to the intermediate representation of gcc [Sta95].
- CACAO is a JIT Java compiler that generates native code for the Alpha architecture [KG97] [Gra97] [KP88] [KEG98].
- Caffeine is an (optimizing) compiler from Java bytecode to native code [HGH96].
- Briki is a research compiler for Java class files [CL97] to test new compilation techniques. It translates class files to a high-level IR. Machine dependent optimizations (such as array remapping, which requires information about memory organization) operate on the IR. Briki optimizes short-circuit operations and uses Baker's algorithm [Bak77] for goto elimination. The compiler can produce Java source (for ease of debugging and better understanding the quality of the generated code).
- Harissa tries to reconcile JIT and off-line compilers by permitting to mix compiled and interpreted code [MMBC97]. The system provides a compiler and an interpreter. The interpreter is integrated in the runtime system. Thus, a compiled program is still able to dynamically load and execute

classes. The compiler translates Java bytecode to C, incorporating optimizations based on class hierarchy analysis.

- Colorado State University has written a Java front end for the SUIF 1.0 compiler system [MDG97]. Not too surprisingly, they are calling it `j2s` as well. Because SUIF 1.0 has no support to describe object-oriented constructs, annotations (see section 3.2.8) have been used to model them.
- Part of the Vortex project [DDG$^+$96] is a Java front end (refer also to section 2.2).
- `JTA` [FdFR96] [FdF96] is a Java bytecode to ANDF translator.
- The `TurboJ` Java Compiler[1] is a Java compiler from the OSF Research Institute. It is written in Java and generates ANSI C code. This compiler changes non-native methods in class files to native methods and generates C code for these native methods. This means that methods that would be usually interpreted by a JVM are transparently replaced with native code. By using this strategy, TurboJ is not required to provide its own runtime system (to handle, for example, memory allocation, garbage collection, or threads). On the other hand, this scheme is less general than actual compilation of a class file and introduces dependencies to a specific runtime system. The current implementation uses the JDK runtime environment (JRE).
- A Java front end is planned for the FLINT compiler system [Sha97b] (refer also to section 2.2).
- The `guavac` compiler[2] is an off-line compiler that expects Java source and generates Java bytecode. It is written in C++ and distributed under the Gnu public license.
- There are various commercial (JIT) compilers from Borland, IBM, Micro$oft, SunSoft, Symantec, etc.

Our compiler is comparable to other off-line Java bytecode front ends that target an IR with object-oriented features and that are part of a complete compiler system. Targeting a compiler system has several advantages. Our compiler front end, for example, can take full advantage of the available SUIF optimization passes, native code back ends, and debugging tools (e.g., the C back end).

The Vortex and FLINT Java front ends seem to be most closely related to the `j2s` compiler. The SUIF 1.0 Java bytecode compiler from Colorado State has to extend the SUIF 1.0 IR with object-oriented information, which yields a good but not seamless integration compared to other compiler systems that already provide an object-oriented IR. The `gcc`-based Java front end from Cygnus Solutions loses all object-oriented information when compiling to the GNU IR. Other compilers, such as Briki, define a high-level object-oriented IR (which might be even tailored for Java), but they do not offer all the benefits of a complete compiler system, such as various optimization passes, debugging tools, and extensibility of the system.

12.2 Decompilers and High-level Control Structure Recovery

A high-level control structure recovery pass for Java bytecodes would check for reducible flow graphs and would try to recover high-level control flow structures from this graph. Source-to-source transformations that replace `goto` statements with structured ones ("goto elimination") [Ram88] [EH93] are a similar form of high-level control structure recovery.

High-level control structure recovery is an essential part of every decompilation process [Cif93] [Cif96]. A *decompiler*[3] is a software tool that translates a program written in a low-level language (source

[1] Homepage of the TurboJ project: `http://www.opengroup.org/RI/java/turbo/`.
[2] Homepage: `http://HTTP.CS.Berkeley.EDU/~engberg/guavac/`.
[3] The Decompilation Page: `http://www.csee.uq.edu.au/csm/decompilation/`.

language) into a semantically equivalent program in a high-level language. A decompiler usually reverses (or tries to reverse) the process of a compiler. The source language of a decompiler is typically machine language or an intermediate representation.

The translation from Java bytecode to Java source can be viewed as decompilation.[4] Because the Java language does not support unstructured control flow (e.g., `gotos`), a decompiler that targets Java is forced to implement some form of high-level control structure recovery.

Cifuentes describes the implementation of the `dcc` decompiler,[5] an i80286 decompiler for DOS binary programs that generates C code [Cif94]. Her thesis gives a chronological overview of various research and commercial decompilation tools (from 1960 to 1993).

Several decompilation tools that translate Java bytecode to Java source are available or in development:

- Mocha[6] is a decompiler written in Java. The current Beta version seems to be unstable, but a significantly improved version is under development. Jasmine[7] is a shareware decompiler that has been derived from Mocha.
- Jad[8] is a freely availabe decompiler written in C++. It seems to be very stable and the recovered control flow is easy to read.
- Krakatoa [PW97], developed at the University of Arizona, is written in Icon. It uses a slightly modified version of Ramshaw's goto elimination algorithms [Ram88] to generate valid Java constructs for control flow. The Java source generated by this phase is hard to read. For this reason Krakatoa applies *rewriting rules* based on reachability and equivalence properties of program points. The current Icon implementation was intended as a test-of-concept. The authors plan to reimplement the decompiler in Java.
- Both WingDis[9] and DejaVu[10] are commercial decompilers. [Dye97] compares the quality of the generated Java code for Mocha, WingDis, and DejaVu with several small test class files.

[4] Decompilation of class files is easier than decompilation of conventional object files because a class file has a well defined structure and conveys much more semantical information.

[5] Homepage: `http://www.it.uq.edu.au/groups/csm/dcc.html/`.

[6] Homepage: `http://www.brouhaha.com/~eric/computers/mocha.html/`.

[7] Homepage: `http://members.tripod.com/*SourceTec/jasmine.htm/`.

[8] Homepage: `http://web.unicom.com.cy/~kpd/jad.html`.

[9] Homepage: `http://www.wingsoft.com/wingdis.shtml`.

[10] Homepage: `http://www.isg.de/OEW/Java`.

Chapter 13

Evaluation

The design and implementation of `j2s` was influenced by two main factors, namely, the design goals of the compiler and the availability of SUIF 2.0 and OSUIF.

When the implementation of `j2s` was started, neither SUIF 2.0 nor OSUIF was available. We first implemented the loader and resolver, because both are completely independent from the back end. The loader (see section 8.1) has been implemented as a stand-alone library.[1] The resolver (see section 8.2) is a separate library as well, but is built on top of the loader library. The actual part of the compiler that uses SUIF is implemented as another library, that in turn is built on top of the loader and resolver. In fact, these three libraries have been implemented as SUIF *subpackages*.

We later modified the loader and resolver to use the SUIF 2.0 assertion and error/warning facilities, which are offered by the `sty` library (see section 4.2). Using SUIF 2.0 functionality means that both loader and resolver are no longer independent stand-alone libraries, but `j2s` has now a consistent look for assertions and messages that conform to SUIF 2.0.

Because we originally wanted to keep the loader and resolver independent from SUIF, the code for both libraries does not contain any SUIF code of the `wallow`, `zot`, `pyg`, or `suif` libraries. This decision caused some rough edges in the design because certain functionality that could have been easily attached to a class implemented in the resolver had to be factored out. For example, the resolver library represents every JVM constant with a separate class (see figure 8.2). It would have been natural to offer a method `suif_type()` that returns the corresponding SUIF `type` for every constant. In contrast, our implementation uses a static method that maps constants to SUIF types. This method contains a `switch` statement that dispatches on the class type of the constant.[2] On the other hand, having a dedicated class that performs the mapping of types for constant, field, and method types might be easier to maintain. In fact, this is one of the few classes that might require minimal changes if the compiler is ported.

After the implementation of the front end was finished, SUIF 2.0 was still not available. We decided to generate SUIF 1.0 code (1) because the framework for the code generation is pretty much the same for every back end, and (2) to familiarize ourselves with the SUIF 1.0 system. This turned out to be a good decision because many concepts in SUIF 1.0 are similar to the ones in SUIF 2.0. For example, both designs use the same SUIF file structure and the instruction list form is very similar. In general, an experienced SUIF 1.0 user should have no trouble to understand and write SUIF 2.0 code. The main obstacle in learning SUIF 2.0 is the lack of documentation; therefore, the source code had to be inspected. The header files are difficult to read, because every class contains a myriad of functions for

[1] The loader has been implemented with Ralph Keller and is also used in his implementation for *Binary Component Adaptation* [KH98].

[2] A `switch` statement (that dispatches on class types) often indicate a deficiency in the design.

the SUIF standard functionality, such as tags, cloning, visitors, and the `zot` interface. Extending SUIF 2.0, as it had to be done for OSUIF, is a whole other story. We briefly outlined the complicated nature of this endeavor in section 5.13.

The first SUIF 2.0 alpha release became available after we finished the code generation for the imperative part of the Java language. Migrating the code generation to SUIF 2.0 was surprisingly easy and is briefly described in [KH97b]. At the early development stage of the code generation, SUIF 2.0 was a moving target. We had to rewrite code on a couple of occasion to reflect changes in the design. Large parts of the SUIF 2.0 code generation had to be written without being able to test or debug the generated code. Furthermore, SUIF 2.0 was completely undocumented. At the early stage we could only print a very low-level representation of the in-memory SUIF IR tree.[3] Later on, it was possible to write out SUIF 2.0 files and to print a high-level representation of the file with `printsuif`. Finally, a SUIF 2.0 to SUIF 1.0 converter pass became available, which translates a subset of the Standard SUIF dialect to SUIF 1.0. This allowed us to generate C code with `s2c`. The generated C code is reasonably easy to read and debug. Unfortunately, the C code does not always closely reflect the generated SUIF 2.0 code, because not every SUIF 2.0 construct can be directly represented with a corresponding SUIF 1.0 construct. The same holds for the translation from SUIF 1.0 to C. We had to modify parts of the code generation to work around deficiencies in the converter.

The generated C code revealed many subtle bugs in the code generation. In retrospect, (design) bugs could have been discovered much earlier in the development process if testing of the generated code had been possible.

Because the development of `j2s` started before SUIF 2.0 was released and grew incrementally while functionality was added to the SUIF 2.0 alpha releases, our implementation does not make optimal use of the functionality offered by SUIF 2.0. The SUIF 2.0 system offers excellent support for data structures, such as lists, hash tables, lexicons, strings, infinite numbers, and bit-vectors. `j2s` implements its own data structures (e.g., fixed-sized and variable length arrays, associate arrays, and strings) instead of using SUIF's implementation.[4] On the other hand, the SUIF library is complex and it would have been impossible to understand and use all features right from the start.

The OSUIF code generation was added as soon as the OSUIF design reached a stable point. New OSUIF features were incorporated incrementally to `j2s` for testing. Instead of migrating from SUIF 2.0 to OSUIF, both SUIF 2.0 and OSUIF code generation were maintained in parallel. This turned out to be a good decision since the OSUIF development progressed far slower than expected. In fact, as of this writing it is not possible to generate an executable from OSUIF code.

The migration from SUIF 1.0 to SUIF 2.0 and the parallel maintenance of SUIF 2.0 and OSUIF delayed the development process, but it was still faster than starting to write SUIF 2.0 (or OSUIF) from scratch.

The visitor and iterator patterns (as described in section 8.2.2) proved to be a good abstraction and have been used extensively throughout the implementation. In general, we tried to apply the object-oriented paradigm to C++, while restricting ourself to a subset of the language. For example, we did not use multiple inheritance, `private`, `protected`, or `virtual` inheritance, exception handling,[5] references, operator overloading, or copy constructors. We limited the use of the preprocessor to assertion macros.

The implementation of `j2s` helped discover various bugs in SUIF 2.0 and served as an early proof-of-concept for the SUIF 2.0 design. The author gained extensive knowledge of SUIF 2.0 while implementing `j2s`, which has been documented in the SUIF 2 FAQ[6] and chapter 4 of this thesis.[7] The `j2s` compiler

[3] This printout is similar to `printzot` on a SUIF file.

[4] We could not use C++ STL [Str97] as an alternative because is also was not available at this time.

[5] Many C++ compiler either do not support it or are buggy.

[6] URL: `http://www.cs.ucsb.edu/~osuif/suif_2/public/suif96_faq.html`.

[7] An earlier version of this chapter appeared as technical report [KH97a] and is the only available documentation of the SUIF 2.0 system as of this writing.

was the first SUIF 2.0 client. In the meanwhile, other SUIF 2.0 clients, such as a C++ front written by at Stanford University and back ends written at Harvard University, are in development.

`j2s` was (and still is) a guinea pig for OSUIF.[8] It influenced the OSUIF design and pointed out design bugs, implementation bugs, and missing functionality in the OSUIF development process. For example, `j2s`'s separate compilation brought up the need for incomplete class types. Chapter 5, which has been extracted as a separate document, is currently the only OSUIF documentation available.

As part of the `j2s` development, we implemented several visualization tools. These tools are not strictly part of the thesis work, but they were helpful to debug the generated code. They are implemented as separate SUIF subpackages (i.e., SUIF back end passes or stand-alone executables) and generate `dot` output:

`cfg2dot`: Visualizes the control flow graph of methods in a class file. `cfg2dot` has been used to generate several figures in this thesis (e.g., figure 8.7).

`dfst2dot`: Shows the depth-first spanning tree that `j2s` generates for methods in a class file. Both figure 8.14 and figure 8.15 have been generated with `dfst2dot`.

`ch2dot`: Illustrates the transitive closure of a Java class as a class hierarchy graph. The graphical output distinguishes between interfaces/class and subclass/subtype relationships.

The `dot` output is generated with a small library, which contains service routines to generate nodes, edges, and digraphs in `dot` format. (It is included in the OSUIF distribution.)

13.1 Conclusions

We implemented `j2s`, a Java bytecode front end for the OSUIF/SUIF 2.0 compiler system. Despite its restrictions, `j2s` can compile a large number of Java applications, such as the transitive closure of classes that constitute the JDK `javac` compiler. `j2s` seems to be reasonably stable for a research compiler and achieves speedups of 2 compared to the JDK1.2beta2 Virtual Machine.

`j2s` is the first front-end available for both SUIF 2.0 and OSUIF. It provided valuable input for the design and implementation of OSUIF, and—to some extend—of SUIF 2.0 as well.

Our compiler, together with the (O)SUIF system, should serve as a good platform for Java-related compiler research. We also hope that our implementation is a useful example implementation for other (O)SUIF front ends.

[8]The author was part of the OSUIF development team at the University of California Santa Barbara.

Appendix A

Performance Data

The following table shows execution times for the benchmarks introcuced in chapter 11:

Benchmark	Input	Runtime (sec.)								
		JVM[1]	UUN	UUG	UON	UOG	OUN	OUG	OON	OOG
`javac`	`Empty`	1.449	0.835	0.825	0.751	0.747	0.525	0.611	0.480	0.516
	`Object`	1.757	3.812	3.686	3.587	3.391	0.964	0.910	0.852	0.798
	`String`	2.749	9.500	8.728	8.506	8.179	1.946	1.908	1.706	1.543
`javap`	`Empty`	1.206	0.432	0.418	0.398	0.747	0.380	0.422	0.402	0.403
	`Object`	1.224	0.611	0.607	0.591	0.575	0.417	0.429	0.389	0.385
	`String`	1.331	1.338	1.294	1.296	1.217	0.460	0.525	0.499	0.485
`sim`	`Trivial1`	2.728	25.116	23.639	23.130	22.372	3.542	3.499	2.331	2.217
	`Trivial2`	2.754	24.925	23.626	23.033	77.030	3.518	3.470	2.253	2.204
	`PingPong1`	4.165	46.670	5.911	37.990	34.945	5.421	4.959	3.341	3.180
	`PingPong2`	4.121	39.107	37.454	37.855	36.451	5.392	4.964	3.295	3.212
`compress`	Small	51.878	17.212	17.455	9.076	9.537	15.134	15.897	7.321	7.883
	Medium	61.380	20.863	20.982	11.083	11.839	17.755	18.332	8.556	8.991
	Large	673.111	211.443	212.793	108.036	109.330	186.158	189.965	83.423	87.526
`db`	Small	0.607	1.284	1.286	1.427	1.230	0.953	0.943	1.024	0.900
	Medium	6.584	31.121	29.521	28.304	27.660	9.242	8.929	8.112	7.814
	Large	286.876	649.994	4874.98	636.271	4620.571	338.144	873.234	328.395	825.89

[1]To compute the speedups in chapter 11 we always executed pairs for every run: First the JDKbeta2 VM and second the executable generated by `j2s`. By running pairs, we hope to mitigate changes in the load balance and get more accurate benchmark results. The JDKbeta2 VM execution times that we give here are the average execution times of these eight JVM runs.

Bibliography

[AD97] Ole Agesen and David Detlefs. Finding references in Java stacks. *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, August 1997.

[Ada89] Douglas Adams. *The Hitchhiker's Guide to the Galaxy, 10th Anniversary Edition.* Harmony Books, 1989.

[ADM98] Ole Agesen, David Detlefs, and J. Eliot Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. *PLDI '98*, June 1998.

[AFM97] Ole Agesen, Stephen Freud, and John C. Mitchell. Adding type parameterization to Java. *OOPSLA '97*, pages 49–65, October 1997.

[AH95] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type analysis: A comparison of optimization techniques for object-oriented languages. *OOPSLA '95*, pages 91–107, October 1995.

[ANJN75] U. Ammann, K. V. Nori, K. Jensen, and H. Nageli. The Pascal (P) compiler implementation notes. Technical report, Institut für Informatik, Eidgenoessische Technische Hochschule, Zürch, 1975.

[App98] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN*, 33(4):17–20, April 1998.

[Arm98] Eric Armstrong. HotSpot: A new breed of virtual machine. *JavaWorld*, 3(3), March 1998.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.

[Bak77] B. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[BG97a] Aart Bik and Dennis Gannon. `Javab`—a prototype bytecode parallelization tool. Technical Report TR 489, Indiana University, Computer Science Department, July 1997.

[BG97b] Aart Bik and Dennis Gannon. `Javar`—a prototype Java restructuring compiler. Technical Report TR 487, Indiana University, Computer Science Department, July 1997.

[BK97] Zoran Budimlic and Ken Kennedy. Optimizing Java: Theory and practice. *Concurreny: Practice and Experience*, 9(6):445–463, 1997.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization in Java. *PLDI '98*, June 1998.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. *OOPLSA '98*, 1998.

[Bot97] Per Bothner. A Gcc-based Java implementation. *IEEE Compcon '97*, February 1997.

[Bra95] Marc Michael Brandis. *Optimizing Compilers for Structured Programming Languages*. PhD thesis, Swiss Federal Institute of Technology Zurch, 1995.

[Bri96] Preston Briggs. The massively scalar compiler project. Unpublished, September 1996.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 19(9):807–820, September 1988.

[BW90] Alan Burns and Andrew Wellings. *Real-Time Systems and their Programming Languages*. Addison-Wesley, 1990.

[CC95] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[CD93] Charles Consel and Oliver Danvy. Tutorial notes on partial evaluation. *POPL '93*, pages 493–501, January 1993.

[CDG96] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical Report 96–06–02, University of Washington, Department of Computer Science, June 1996.

[CDG+97a] Andrew Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Supporting high level programming with high performance: The Illinois Concert System. *High-level Parallel Programming Models and Supportive Environments (HIPS '97)*, April 1997.

[CDG97b] Cyrille Comer, Gary Dismukes, and Franco Gasperoni. Targeting GNAT to the Java Virtual Machine. *TRI-Ada '97*, pages 149–161, 1997.

[CFRW91] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependency graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Cha95] David Chase. Implementation of exception handling — I. Unpublished, November 1995.

[Cha96] David Chase. Implementation of exception handling — II. Unpublished, September 1996.

[Cif93] Cristina Cifuentes. A structuring algorithm for decompilation. *XIX Conferencia Latinoamericana de Informatica*, pages 267–276, August 1993.

[Cif94] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.

[Cif96] Cristina Cifuentes. Structuring decompiled graphs. *Lecture Notes in Computer Science Vol. 1060 — Compiler Construction*, pages 91–105, 1996.

[CL97] Michael Cierniak and Wei Li. Briki: An optimizing Java compiler. *IEEE Compcon '97*, February 1997.

[Coh81] Norman H. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, 1981.

[CS93] Ron Cytron and David Shields. FRIL — a fractal intermediate language. Unpublished, July 1993.

[CU90] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented languages. *PLDI '90*, pages 150–164, June 1990.

[DCI+97] A. Duncan, B. Cocosel, C. Iancu, H. Kienle, R. Rugina, U. Hölzle, and M. Rinard. OSUIF: SUIF with objects. *Second SUIF Compiler Workshop*, pages 1–7, August 1997.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. *OOPSLA '96*, pages 83–100, October 1996.

[DF84] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, 1984.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. *IEEE Symposium on Security and Privacy (SOSP '96)*, pages 190–200, May 1996.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programms using static class hierarchy analysis. *ECOOP '95*, pages 77–101, August 1995.

[DH95] Karel Driesen and Urs Hölzle. Minimizing row displacement dispatch tables. *OOPSLA '95*, pages 141–155, October 1995.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on piplined processors. *ECOOP '95*, pages 253–282, August 1995.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. *OOPSLA '96*, pages 292–305, October 1996.

[DS84] L. Peter Deutsch and Allan Schiffman. Efficient implementation of the Smalltalk-80 system. *POPL '84*, pages 297–302, 1984.

[Dye97] Dave Dyer. Java decompilers compared. *JavaWorld*, 2(7), July 1997.

[EH93] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. ACAPS Technical Memo 76, McGill University, School of Computer Science, September 1993.

[EM95] M. Anton Ertl and Martin Maierhofer. Translating Forth to efficient C. *EuroForth '95*, 1995.

[EP94] Dawson R. Engler and Todd A. Proebsting. DCG: An effiecient, retargetable dynamic code generation system. *Architectural Support for Programming Languages and Operating Systems (ASPLOS '94)*, pages 263–272, 1994.

[Ert96] Martin Anton Ertl. *Implementation of Stack-Based Languages on Register Machines.* PhD thesis, Technische Universität Wien, April 1996.

[FdF96] Christian Fabre and François de Ferrière. Java-ANDF feasibility study — intermediate report. *Open Software Foundation Research Institute*, June 1996.

[FdFR96] Christian Fabre, François de Ferrière, and Fred Roy. Java-ANDF feasibility study — final report. *Open Software Foundation Research Institute*, December 1996.

[Fer96] Mary F. Fernández. *A retargetable, optimizing linker*. PhD thesis, Princeton Universitly, January 1996.

[FH91] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, 1991.

[FK96] Michael Franz and Thomas Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, December 1996.

[Fra94] Michael Steffen Oliver Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1994.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJ87] Carlo Ghezzi and Medhi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 1987.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[GM95] James Gosling and Henry McGilton. *The Java Language Environment — A White Paper*. Sun Microsystems, October 1995.

[Goo75] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1983.

[Gra97] Reinhard Grafl. CACAO: Ein 64bit Java VM just-in-time compiler. Master's thesis, Technische Universität Wien, January 1997.

[Hav97] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, July 1997.

[HBG+97] Urs Hölzle, Lars Bak, Steffen Grarup, Robert Griesemer, and Srdjan Mitrovic. Java on steroids: Sun's high-performance Java implementation. *Hot Chips IX, Stanford University*, August 1997.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. *ECOOP '91*, pages 21–38, July 1991.

[HGH96] Cheg-Hsueh Hsieh, John C. Gyllenhaal, and Wen-mei Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. *29th Annual International Symposium on Microarchitecture*, pages 90–99, December 1996.

[HS96] Jonathan Hardwick and Jay Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, August 1996.

[JCR97] JavaSoft. *Java Core Reflection — API and Specification*, January 1997.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. *OOPSLA '88*, pages 18–26, September 1988.

[JIC97] JavaSoft. *Inner Classes Specification*, October 1997.

[JNI97] JavaSoft. *Java Native Interface Specification, Release 1.1*, January 1997.

[Joh96] Raph E. Johnson. Type-checking Smalltalk. *OOPSLA '86*, pages 315–321, September 1996.

[KEG98] Andreas Krall, Anton Ertl, and Michael Gschwind. JavaVM implementation: Compilers versus hardware. *Computer Architecture (ACAC '98)*, pages 101–110, 1998.

[KF96] Thomas Kistler and Michael Franz. A tree-based alternative to Java byte-codes. Technical Report 96–58, Department of Information and Computer Science, University of California, Irvine, December 1996.

[KG97] Andreas Krall and Reinhard Grafl. CACAO — a 64bit Java VM just-in-time compiler. *PPoPP Workshop on Java for Science and Engineering Computation*, June 1997.

[KH97a] Holger Kienle and Urs Hölzle. Introduction to the SUIF 2.0 compiler system. Technical Report TRCS97–22, Department of Computer Science, University of California Santa Barbara, December 1997.

[KH97b] Holger Kienle and Urs Hölzle. `j2s`: A SUIF Java compiler. *Second SUIF Compiler Workshop*, pages 8–15, August 1997. Also available as Technical Report TRCS97–16, Department of Computer Science, University of California Santa Barbara.

[KH98] Ralph Keller and Urs Hölzle. Implementing binary component adaptation. Technical Report TRCS98–21, Department of Computer Science, University of California Santa Barbara, 1998.

[Kli81] Paul Klint. Interpretation techniques. *Software—Practice and Experience*, 11(10):963–973, October 1981.

[KN93] Elefherios Koutsofios and Stephen North. *Drawing graphs with* dot. AT&T Bell Laboratories, October 1993.

[KP88] Andreas Krall and Mark Probst. Monitors and exceptions: How to implement Java efficiently. *ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 15–24, February 1988.

[Kra83] Glenn Krasner. *Smalltalk-80: bits of history, words of advice.* Addison-Wesley, 1983.

[KVH97] Andras Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchy encoding of types. *ECOOP '97*, pages 128–145, 1997.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1996.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types in Java. *POPL '97*, pages 132–145, January 1997.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine.* O'Reilly, March 1997.

[MDG97] Sumith Mathew, Eric Dahlman, and Sandeep Gupta. Compiling Java to SUIF: Incorporating support for object-oriented languages. Technical Report CS-97-114, Computer Science Department, Colorado State University, July 1997.

[ME98] Martin Maierhofer and M. Anton Ertl. Local stack allocation. *Springer Lecture Notes in Computer Science 1383*, pages 189–203, 1998.

[MMBC97] Gilles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code. *COOTS '97*, June 1997.

[NAJ+81] K. Nori, U. Ammann, K. Jensen, H. Nageli, and C Jacobi. Pascal-P implementation notes. *Pascal — The Language and its Implementation, John Wiley & Sons*, pages 107–136, 1981.

[OQL97] Andrea Olgiati, Stuart Quick, and John Lumley. One year of experience with SUIF. *Second SUIF Compiler Workshop*, pages 141–145, August 1997.

[OSF91] Open Systems Foundation. *OSF Architecture-Neutral Distribution Format Rationale*, June 1991.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translation theory into practice. *POPL '97*, pages 146–159, January 1997.

[PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. *OOPSLA '94*, pages 324–340, October 1994.

[PC97] Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late than never"? *POPL '97*, pages 120–131, January 1997.

[Ple96] John B. Plevyak. *Optimization of Object-oriented and Concurrent Programms*. PhD thesis, Universiy of Illinois at Urbana-Champaign, August 1996.

[Plö94] Prof. Dr. Erhard Plödereder. Syntaxanalyse. Vorlesungs-Skript, 1994.

[PTB+97] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for applications — a way ahead of time (WAT) compiler. *COOTS '97*, June 1997.

[PW97] Todd Proebsting and Scott Watterson. Krakatoa: Decompilation in Java. *COOTS '97*, June 1997.

[Ram88] Lyle Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, October 1988.

[Rob89] Eric S. Roberts. Implementing exceptions in C. Technical Report 40, DEC Systems Research Center, Palo Alto, CA, March 1989.

[Rut98] Andrew Rutz. A runtime environment for a static Java compiler. Master's thesis, Department of Computer Science, University of California Santa Barbara, 1998. To appear Fall 1998.

[SCG94a] Stanford Compiler Group, Stanford University. *The BUILDER Library, Version 1.0*, 1994.

[SCG94b] Stanford Compiler Group, Stanford University. *The SUIF Library, Version 1.0*, 1994.

[Sha97a] Zhong Shao. An overview of the FLINT/ML compiler. *ACM SIGPLAN Workshop on Types in Compilation (TIC'97), Amsterdam, The Netherlands*, June 1997.

[Sha97b] Zhong Shao. Typed common intermediate format. *USENIX Conference on Domain-Specific Languages, Santa Barbara, CA*, pages 89–101, October 1997.

[Smi96] Michael D. Smith. Extending SUIF for machine-dependent optimizations. *First SUIF Compiler Workshop*, January 1996.

[Som92] Ian Sommerville. *Software Enginnering*. Addison-Wesley, fourth edition, 1992.

[Sta95] Richard M. Stallman. *Porting GNU CC*. Free Sofware Foundation, Boston, MA, November 1995.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[Taf96] Tucker Taft. Programming the Internet in Ada 95. *Ada Europe 96 Conference: Reliable Software Technologies*, 1996.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit — a tool for building optimizers. *PLDI '92*, pages 82–93, July 1992.

[USCH92] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. Object, message, and performance: How they coexist in Self. *IEEE Computer*, 12(10):53–64, October 1992.

[VHK97] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. *OOPSLA '97*, pages 142–157, October 1997.

[Wad97] P. Wadler. Editorial: A HOT opportunity. *Journal of Functional Programming*, 2(7):127–128, March 1997.

[WFW+94] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and imizing compilers. *ACM SIGPLAN Notices*, pages 31–37, December 1994.

[Wil97a] Chris Wilson. Private email communication, August 1997.

[Wil97b] Chris Wilson. Private email communication, November 1997.

[Wil97c] Chris Wilson. SUIF 2.0 tutorial. Given at the Second SUIF Compiler Workshop, August 1997.

[Wir88] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, 1988.

[Yel95] Frank Yellin. Low level security in Java. *Fourth International World Wide Web Conference, Boston, MA*, December 1995.