

A Compiler Infrastructure for High-Performance Java^{*}

Neil V. Brewster and Tarek S. Abdelrahman

Department of Electrical and Computer Engineering,
University of Toronto,
Toronto, Ontario, Canada M5S 3G4
{brewste,tsa}@eecg.toronto.edu

Abstract. This paper describes the zJava compiler infrastructure, a high-level framework for the analysis and transformation of Java programs. This framework provides a robust system, guaranteeing under transformations both the consistency of its internal structure and the syntactic correctness of the represented code. We address several challenges unique to Java, which have not been addressed by earlier frameworks. These include automatic maintenance of complex symbol scope information under transformations, insertion of implicit code to accurately model the source program, incorporation of compiled code into the representation, and representation of the complex control flow of exception handling constructs. We include support for the sharing of information between compiler passes, and a framework for interprocedural analysis. We believe that the features we introduce in the zJava compiler infrastructure will result in a means of rapidly prototyping new Java compiler analyses. We give a number of examples illustrating the use and utility of the infrastructure.

1 Introduction

There has been considerable research during the past decade on parallelizing compilers and automatic parallelization of programs. Traditionally, this research focused on scientific applications that consist of loops and array references, typical of Fortran programs [6, 9]. Regrettably, this focus has limited the widespread use of automatic parallelization in industry, where the majority of programs are written in C, C++, or more recently in Java. These programs extensively use pointer-based dynamic data structures such as linked-lists and trees, and often use recursion. These features make it difficult to directly utilize parallelizing compiler technology developed for array structures and simple loops.

The goal of the *zJava* (pronounced “zed Java”) compiler project, developed at the University of Toronto, is to investigate automatic parallelization technology for programs that use pointer-based dynamic data structures and recursion, written using the Java programming language. The zJava compiler infrastructure was developed in support of this research. The infrastructure has a number

^{*} This work is supported by NSERC (Canada) and CITO (Ontario) grants.

of features which facilitate quick and efficient prototyping of new compiler optimizations for Java. These features address challenges in the representation of Java programs at the source level, including automatic maintenance of complex symbol scope information under transformations, insertion of implicit code to accurately model the source program, incorporation of compiled code into the representation, and representation of the complex control flow of exception handling constructs. These challenges have not been addressed by similar previous compiler infrastructures.

The remainder of this paper is organized as follows. Section 2 gives a general overview of the infrastructure. Section 3 describes the main components of the architecture. Section 4 describes some implementation details. Section 5 presents examples of the use of the infrastructure. Section 6 reviews and contrasts related work. Finally, Section 7 provides some concluding remarks.

2 Overview

The zJava compiler infrastructure utilizes both a high level and a low level intermediate form. A front-end parses Java source code into the zJava High Level Intermediate Representation (HLIR), which can be converted back to human-readable Java source. Code generation converts HLIR into the zJava ByteCode Intermediate Representation [10] (BCIR), which provides an interface to the `.class` file format. Program analyses and transformations take place primarily on the high-level form. A high-level representation has the advantage of retaining the syntactic structure of high-level constructs, which benefits analyses such as dependence analysis, array structuring, loop parallelization and transformation, and induction variable elimination.

There are a number of interesting features of the zJava HLIR. We use extensive error checking and reporting to speed up the process of prototyping new compiler passes. These checks guarantee that the representation is maintained consistent under transformations. Both the syntactic correctness of the represented Java program and the correctness of internal HLIR structures are enforced. In addition, internal structures are automatically updated under transformations. The core of the representation is a flat list of statements for each method body. High-level symbol information, including a precise representation of scoping, is included. We augment the represented program to include implicit code, which is not necessarily explicit in the source code but is required [8]. A control flow graph is built on the representation, including the modeling of Java exception flow. We provide mechanisms to save the results of an analysis for use by later compiler passes. When source code is not available, skeletal HLIR can be constructed from `.class` files. Finally, a “rich” class file format facilitates interprocedural analysis, even when source code may not be available.

3 Architecture

3.1 Robust Framework

HLIR is designed minimize the time required to prototype new compiler passes, by detecting and reporting errors as early in the development process as possible. The API is designed such that most errors can be detected at compile-time¹. Where that is not possible, run-time checks and Java exceptions are used to detect and report errors. We define an error as any transformation which would leave the IR in an inconsistent state, both internally and in terms of the syntactic correctness of the represented Java code.

Syntactic Consistency The syntactic consistency of the representation is primarily enforced by controlling the ways in which lists of statements are created and modified. The constructors for creating statement lists require that all parts of a construct be specified at once. For example, it is not possible to create a *while* loop with no body. Additionally, the constructors for each Java statement type require that all necessary components be present. It is not possible, for example, to create a *while* loop header without specifying the loop conditional expression.

The statement list implementation also strictly controls the way in which statements are added or removed. It is not possible to incrementally build a program by adding one statement at a time, even if the final result is syntactically correct. Instead, the representation is built by adding statement lists into other statement lists. Removals which would leave a construct in a syntactically invalid state (such as removing the *then-part* of an *if* construct) are not permitted. Removal of constructs is made possible by the fact that removal of a header statement results in the removal of the entire construct. For example, removing a *try-header* statement will result in the removal of the *try block*, and, if present, the *catch clauses* and the *finally clause*. Methods are provided to remove components of constructs, such as individual *case clauses* in a *switch* statement.

Internal Consistency We have designed HLIR to automatically maintain its internal structures under transformations. For example, when statements are added to or removed from a statement list, the symbol table information is automatically updated, and the types of all symbols and expressions in the new statements are resolved. This relieves users of the need to understand the details of the scope representation in HLIR, and of the need to maintain it correctly themselves. Additionally, HLIR disallows any modifications (such as creating, adding, or removing statements or other objects) which would leave the internal

¹ We use the term *compile-time* to refer to the time at which a zJava compiler pass is compiled, and the term *run-time* to refer to the time at which the zJava compiler executes.

representation in an inconsistent state. For example, removal of an *end* statement, which is used internally to represent the end of a structured construct, is prohibited.

The core functionality of an intermediate representation involves building objects to represent parts of a program (for example, statements), and using these to populate higher-level objects (for example, methods and classes). In this kind of environment, one common programmer error is object *sharing*, where the same object is used as a component of multiple IR objects. For example, several *symbol* objects, representing different integer variables, might each have a reference to a *type* object representing the integer type. Should all of these *symbol* objects reference the same *type* object, a programmer who wished to change the type of one symbol might unwittingly change the type of them all. The concept of object *ownership*, as used in the Collections hierarchy of the Polaris project [6], allows HLIR to prevent such object sharing. When an object is inserted into a list, the list gains “ownership” of that object. An object can be owned by only zero or one list, and a list may not contain any objects it does not own. For example, attempting to add the same statement object to the body of a method more than once will cause a run-time error.

Method bodies are represented in HLIR as flat lists of non-recursive statement structures. This means that compiler passes can simply iterate over the statements of a method body, without specific knowledge of the structure of each statement. A sequential list of statements is also convenient when dividing a method body into basic blocks for the control flow representation.

In HLIR, each construct begins with a *header* statement and ends with a generic *end* statement. Links are included, specific to the construct; for example, an *if-header* contains links to the *then-part*, the *else-part*, and the *if-end*.

3.2 Symbol Table Representation

Representing all information available in the source code also involves capturing scope information. The zJava HLIR creates a symbol table for each unique scope, and chains it to the enclosing scope. A symbol table tree is constructed in parallel with HLIR, and is automatically updated whenever symbol information changes. Symbol lookups automatically proceed up the chain until the highest level scope is encountered. This design implicitly implements variable hiding; the closest declaration of a variable is seen first, so local variables can shadow instance variables. The type of every symbol and expression is resolved before any user compiler passes are invoked.

Every statement represented in HLIR includes a link to the symbol table for the scope in which it resides. Thus, when examining any statement in the program, HLIR presents what appears to be a single symbol table containing all symbols visible to that statement. Qualified symbols (eg. `Foo.a`) are resolved by querying the symbol table of the qualifying type. Symbol table lookups are designed to also automatically examine available superclasses for accessible symbols, loading them from `.class` files if necessary.

3.3 External Class Resolution

Interprocedural analyses often require information about classes for which the source code is not available. For example, the Java class libraries are usually only available in compiled (`.class` file) form. The zJava infrastructure includes the ability to construct skeletal HIR from `.class` files generated by any compiler. Using the zJava ByteCode Intermediate Representation [10] (BCIR), zJava automatically locates and loads `.class` files when needed. The resulting representation includes all information available in the `.class` file, except for the actual bodies of methods—we make no attempt to decompile bytecodes.

3.4 Modularity of Analyses

The zJava infrastructure is designed to allow compiler passes to build on the results of previous analyses, without requiring that each original analysis be repeated. It is important not to equate the concept of modular analysis to the ability to run several independent compiler passes prior to code generation; the latter is a feature of most program restructuring systems, including zJava.

We include support for passing user-defined directives to the compiler in the form of special comments in the source code, and for including such annotations in the output source code. Each directive is encapsulated inside an attribute and attached to a list on the appropriate HIR object. This technique is best suited to storing the results of an analysis (eg. “this loop is parallel”), rather than the actual data structures generated and populated by the analysis (eg., dependence graphs). While the latter would be possible, it could result in unacceptably large blocks of comments in the source code.

The `.class` file specification [11] includes support for user-defined attributes, which are ignored by standard classloaders. We have developed a “rich” class format, designed to store the results of compiler analyses in the form of special attributes. Bytecode generation can store the results of an analysis by including custom attributes while generating BCIR. This technique is very useful for classes which are not typically available in source form. For example, the standard Java library classes could be analysed once, and compiled into augmented class files. Later analysis of user code which calls into these libraries would thus have access to the results of analysing the library classes.

4 Implementation Details

4.1 zJava Compiler Front-end

The zJava compiler infrastructure is implemented entirely in Java. The front-end of the zJava compiler uses the Java Tree Builder (JTB), developed at Purdue University, to automatically generate abstract syntax tree (AST) class definitions based on the input grammar. JTB also generates a Visitor design pattern. The Java Compiler-Compiler (JavaCC) package from Sun Microsystems is used to automatically generate a parser that builds the AST defined by JTB. Finally, visitor classes are defined to generate the high-level representation.

4.2 Collections

At the root of the HLIR class hierarchy is the class `zObject`. Any object of type `zObject` can be cloned, converted to source code, and flagged as either owned or unowned. All HLIR class constructors check the ownership flags of `zObjects` passed into them. If the constructor expects to have ownership of a particular object, but finds that it is already owned, it will throw `zObjectOwnedError`.

Much of the representation involves lists of objects, and this provides another mechanism for ownership enforcement. The classes `zList` and `zRefList` are derived from `java.util.LinkedList` for this purpose. A `zList` may only contain objects of type `zObject`, and any `zObject` may be contained in at most one `zList`. The list manipulation methods are overridden to enforce these constraints. The `zRefList` class exists for cases where objects in the list are expected to be owned by some other list. For example, a *switch* statement references a list of *case* statement objects, each of which is in fact owned by the enclosing statement list. Thus, the *switch* statement utilizes a `zRefList` to store the references to its *case* statements. Any object inserted into a `zRefList` must already be owned. Methods in `zRefList` which provide access to its elements first check that the element is still owned by another list. If an element in a `zRefList` is removed from the `zLinkedList` that owns it, any subsequent attempts to access that element through the `zRefList` will cause a `zObjectUnownedError`.

4.3 HLIR Classes

The `Program` class is the root of HLIR, representing a set of source files compiled on the same command line. `Program` contains a `compilationUnit` to represent the information in each Java source file specified, the core of which is the list of classes declared in the file. Each class is represented by a `classObject`, which contains information specific to the class, in addition to a list of the methods declared. Each method is represented by a `methodObject`, and contains a reference to a `stmtList` object. The `stmtList` contains the list of statements representing the method body. The `zStatement` hierarchy implements classes to represent the various types of Java statements, and the `zExpression` hierarchy defines a class to represent each kind of expression. Certain expressions contain symbols, which are represented in the classes of the `zSymbol` hierarchy. Additional classes are implemented to represent the remaining information available in the source code, such as types and access modifiers.

5 Examples

In this section we detail three examples of HLIR in use. The process of implementing these examples has not only verified the functionality of the infrastructure and the completeness of the API documentation, but has also demonstrated that our goal of rapid prototyping has been achieved.

5.1 Statement Insertion

This simple example shows the insertion of a `while` loop into an existing method body. The source code equivalent of the loop is given below.

```
int i = 0;
while (i < 10) {
    System.out.println("Hello World!"); ++i;
}
```

The Java code given in Figure 1 demonstrates the construction and insertion of this loop. It is assumed that `method_body` references list of statements currently in the method, and that `idx` is the index into that list at which the new loop is to be inserted. The declaration of `i` will be translated into a symbol table entry which will be merged with the existing information, and each use of `i` will be resolved to point to this entry. The expression `System.out.println` will be separated into three variable expressions and then combined as two field accesses, and the type of each will be resolved by examining the appropriate `.class` files.

```
javaType int_t = new javaType(javaType.t_int);
varSymbol loopVar = new varSymbol("i", int_t, new fieldModifier());
loopVar.setInitializer(new varExpression("5"));
LinkedList declsList = new LinkedList(); declsList.addLast(loopVar);

// Loop conditional: i < 10
varExpression leftExpr = new varExpression(new String("i"));
varExpression rightExpr = new varExpression(new String("10"));
binaryExpression conditionalExpr = new binaryExpression(leftExpr,
    binaryExpression.op_lessthan, rightExpr);

// Loop body:
LinkedList argsList = new LinkedList();
argsList.addLast(new StringLiteralExpression("Hello World!"));
naryExpression arguments = new naryExpression(argsList);
varExpression methExpr = new varExpression("System.out.println");
methodInvocation invokeExpr = new methodInvocation(methExpr, arguments);
stmtList bodyList = new stmtList(new exprStatement(invokeExpr));
preUnaryExpression incrExpr = new preUnaryExpression(
    unaryExpression.op_plusplus, new varExpression("i"));
bodyList.addLast(new stmtList(new exprStatement(incrExpr)));

// Loop insertion:
whileStatement whileStmt = new whileStatement(conditionalExpr);
stmtList newList = new stmtList(new declStatement(declsList));
newList.addLast(new stmtList(whileStmt, bodyList));
method_body.add(idx, newList);
```

Fig. 1. Using HLIR to construct a `while` loop.

5.2 Data Structure Visualization

The Data Structure Visualizer (DSV) provides a means of graphically representing the objects and pointers in a program, dynamically displaying common data structures (linked lists, binary trees, etc) as a program executes. A zJava compiler pass has been implemented which restructures a Java program, inserting appropriate calls to the DSV library. This pass makes use of zJava source directives to inform the compiler which variables represent nodes to be displayed, which nodes represent various types of pointers, and which classes make use of the node class. Much of the functionality of HLR is used, including augmenting classes with the addition of static initializer blocks, class fields, and entire methods. As well, each expression is examined, to identify those which assign to a node or allocate a new node. Statements are then automatically added around each of these expressions, calling into the DSV library to update the display.

5.3 Software Architecture Visualization

The Portable Bookshelf (PBS), developed at the University of Toronto, is an implementation of the Software Bookshelf [7], a web-based framework for the presentation and navigation of information representing large software systems. The PBS system is used to generate landscapes of each subsystem, displaying objects within the subsystem and the relationships between them. The zJava compiler infrastructure has been used to implement the first stage of landscape creation—the extraction of file-level facts from source programs. An HLR pass extracts the set of low-level relationships defined in [1], such as *methodDefBy* (Method **A** is defined by Type **B**), *castsTo* (Method **A** casts an expression to Type **B**), and *arrayType* (Array **A** is an array of Type **B**). Existing tools are then used to infer higher-level relationships, and to generate the landscapes.

6 Related Work

High-level program manipulation and analysis frameworks exist for several languages. Those which represent programs at a high level and support source-to-source transformations are closest to our work. The Polaris Fortran compiler [6] includes automatic enforcement of syntactic consistency, and incremental maintenance of IR structures, including control flow information. The Sage++ toolkit [5], uses a common high-level IR to provide a framework for building Fortran, C, and C++ restructuring systems. The Paraphrase-2 automatic parallelizing compiler [12] compiler uses the combination of a data dependence graph, control flow graph, and call graph to represent C and Fortran programs. The SUIF [9] research framework is targeted to imperative languages such as Fortran and C, providing an intermediate format and a set of common optimization passes. OSUIF [4] is an extension of SUIF2.0, introducing support for object-oriented optimizations. Score [13] and Vortex [3] support both high-level and low-level language constructs. The Illinois Concert system [2] is an optimizing compiler for the concurrent object-oriented programming model ICC++.

The zJava infrastructure is unique in a number of aspects. Automatic consistency maintenance, demonstrated by Polaris for Fortran programs, is more complex in a Java representation. The conversion to a flat representation is complicated by the introduction of more complicated structures. The representation of symbol scope is made more difficult by inheritance relationships and by the possibility of variable declarations anywhere in a block. The semantics of the Java exception handling constructs greatly complicates the control flow representation. In order to accurately model a Java program, HLIR must implicitly insert code which is not present in the source. To facilitate interprocedural analysis, we provide functionality to augment `.class` files with analysis results, and the ability to incorporate compiled code into the high level representation.

Table 1 summarizes the functionality provided by several of the systems discussed above, including zJava. The comparison criteria are: automatic syntactic consistency enforcement under transformation; support for modular analysis; the ability to convert the IR into source code; the representation of both high and low level constructs; and the set of languages supported by the compiler.

	Automatic Consistency	Modular Analyses	Source-to-Source	High and Low Level	Languages Supported
zJava	•	•	•	•	Java
Polaris	•	•	•	○	Fortran
SUIF	○	•	•	•	Fortran, C, C++
Score	○	○	○	•	N/A
Sage++	○	•	•	○	Fortran, C, C++
Paraphrase-2	○	○	•	○	Fortran, C
Concert	○	•	○	○	IC++
Vortex	○	○	○	•	Cecil, C++, Modula-3, Java bytecode

Table 1. Comparison of various high-level compiler frameworks. A filled circle implies that the system has the associated functionality.

7 Concluding Remarks

The zJava infrastructure is a program analysis and source-to-source transformation system for Java. It provides a robust framework for the prototyping of new compiler analyses and optimizations. To facilitate rapid development of new compiler passes, we have designed the high level intermediate representation to detect misuse as early in the process as possible. HLIR prohibits transformations which would leave the representation in an inconsistent state, both in terms of its internal structures and in terms of the syntactic correctness of the represented Java program. We have included support for annotations in both the source code and the bytecode, and the ability to construct the IR from compiled code.

We have successfully verified the functionality of the zJava infrastructure as a program analysis and manipulation tool through the construction of control flow information, the extraction of low-level facts, and the augmentation of user programs to interface with other libraries. Further, a self-compile test and the compilation of the SPECjvm98 benchmarks have verified that our infrastructure produces correct code, and does not alter the semantics of the original program under direct source-to-source transformation.

The zJava infrastructure has been released in the public domain as a tool for researchers to analyse and restructure Java programs. Our current research involves the incorporation of the Omega dependence test in zJava for the parallelization of loops, and the implementation of path expression analysis for the parallelization of Java programs at the method level.

References

1. I. T. Bowman. Architecture recovery for object-oriented systems. Master's thesis, University of Waterloo, 1999.
2. A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, and X. Zhang. Supporting high level programming with high performance: The Illinois concert system. In *Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, 1997.
3. J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *ACM Conference on Object Oriented Programming Styles*, 1996.
4. A. Duncan, B. Cocosel, C. Iancu, H. Kienle, R. Rugina, U. Hölzle, and M. Rinard. OSUIF: SUIF 2.0 with objects. In *2nd SUIF Compiler Workshop*, 1997.
5. F. Bodin et al. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *OONSKI*, 1994.
6. K. A. Faigin, J. P. Hoeflinger, D. A. Padua, P. M. Petersen, and S. A. Weatherford. The Polaris internal representation. Technical Report CSRD-1317, University of Illinois at Urbana-Champaign, February 1994.
7. P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.
8. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
9. M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.
10. D. Lew. BCIR: A framework for the representation and manipulation of Java bytecode. Master's thesis, University of Toronto, 2000.
11. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.
12. C. D. Polychronopoulos, M. B. Girkar, M. R. Haghghat, C. L. Lee, B. P. Leung, and D. A. Schouten. The structure of Paraphrase-2: An advanced parallelizing compiler for C and Fortran. In *LCPC Workshop*, pages 423–453, 1989.
13. G. E. Weaver, K. S. McKinley, and C. C. Weems. Score: A compiler representation for heterogeneous systems. In *Heterogeneous Computing Workshop*, 1996.