

Java Layers Compiler Design

Richard Cardone, Calvin Lin
October 30, 2000

1 Introduction

This document contains a brief overview of the JL compiler's design (version 2) [4]. The compiler supports bounded parametric classes and interfaces, mixins, static virtual typing, deep conformance, constructor propagation, and the class flattening optimization. The intent of this document is to provide a basis for an implementation plan, so we provide the detail needed to gauge the development effort and feasibility of our design. The Java Layers home page [5] contains specifications that describe and justify our choice of language features in greater detail. We now describe our high-level design, the rationale used in making design choices, and major implementation tasks required to realize our design.

2 Design Space

Implementations of parametric polymorphism in Java are often categorized as either *homogeneous* or *heterogeneous* [1,3]. Homogeneous implementations are characterized by single executable representation for all instantiations of a parametric type. The type parameters that appear in a parametric type are *erased* and replaced by a suitably general type. For example, if type parameter *T* of a parametric type is unconstrained, `Object` can be substituted for *T* in the parametric type's erasure. Type safety is achieved by inserting dynamic type casts and bridge methods into code that instantiates parametric types.

Heterogeneous implementations, on the other hand, generate a unique executable representation for each instantiation of a parametric type. Under this approach, the actual type parameters specified in a instantiation substitute for their corresponding formal type parameters and a unique executable is generated. Intuitively, the heterogeneous approach can be thought of as a use of macro expansion and the homogeneous approach as a use of subtype polymorphism and dynamic type casting.

In JL, we choose to implement a heterogeneous solution to parametric polymorphism. The advantages of heterogeneous implementations include less runtime overhead, more accurate reflective capabilities, and the ability to use type parameters in allocation, cast and instanceof expressions. Type parameters can appear wherever a type can appear under the heterogeneous approach. This expressiveness was an important factor in determining how JL should be implemented. The disadvantages of heterogeneous implementations, when compared to homogeneous implementations, include higher memory usage in certain circumstances and possible access control restrictions when parametric types and their actual type parameters are defined in different packages. Implementation tradeoffs between the two approaches have been discussed extensively in the literature [1,2,3].

From an implementation point of view, Java can be extended with parametric polymorphism by using some combination of source code pre-processors, modified compilers, bytecode transformers, specialized class loaders, and modified virtual machines. This design document describes a source-to-source pre-processor that transforms JL source into Java source, which is then compiled by a standard Java compiler. Before describing the details of our chosen design, we discuss alternative designs and the rationale we used in making our choice.

2.1 Modified JVM

Changes to the JVM or Java's architected set of bytecodes are the most disruptive in that compatibility with all current Java implementations is sacrificed. In addition, any such approach is complicated by the need to ensure that Java's type system and security mechanisms are not compromised. For these reasons, implementation strategies that require changes to the JVM were not considered for JL.

2.2 Modified Compiler

Of the remaining approaches, extending a Java compiler with JL semantics provides the most integrated implementation solution. The compiler's parser, semantic checker, code generator and optimizer could be

modified to produce an extended-format bytecode for parametric types. A specialized class loader [1,2] could also be developed to process this extended-format bytecode. The class loader would produce instantiations of parametric types at load-time by transforming the extended-format bytecode into Java's architected bytecode, thus avoiding the need to change current JVMs. This approach could also produce instantiations at compile-time by invoking the same class loader from the compiler and saving the generated instantiations to file. This would remove the need for a specialized class loader in all runtime environments.

In our research environment, the main drawback to modifying an existing Java compiler is that access to and familiarity with the source code of a Java compiler is required. We would incur the considerable startup cost of learning the internals of a particular compiler implementation. In order to attract users, a well-regarded compiler would have to be chosen as the starting point. In order to keep users, the compiler would have to be maintained to keep it competitive as the Java language and techniques used to implement the language evolve. This maintenance is orthogonal to JL research. Therefore, due to its high development cost, the fully integrated solution of modifying an existing Java compiler will not be pursued.

2.3 Bytecode Pre-Processor

Instead of modifying an existing Java compiler, we could implement a pre-processor that takes JL source code as input and generates the same extended-format bytecode as described in the last section as output. This type of pre-processor would parse parametric type definitions, record their JL semantic information, then strip all JL-specific constructs from the source code. The pre-processor would then send the stripped source code to a Java compiler to generate class files that would be used as *bytecode templates*. Finally, the pre-processor would encode the previously recorded JL semantic information in the bytecode templates, extending their format just as we did in the modified compiler approach described above. From this point on, the pre-processor approach and the modified compiler approach are the same; a specialized class loader would be used generate instantiations of parametric types.

This approach has the advantage of treating Java compilers as black boxes and decoupling JL's implementation from that of any Java compiler. The pre-processor does not replace the Java compiler, so users can continue to work in their preferred Java development environment. One characteristic of this approach, however, is that Java bytecode has to be manipulated by both the pre-processor and the specialized class loader. The pre-processor changes the bytecode when it embeds JL semantic information into class files. The class loader uses the embedded JL information as directives describing what bytecode *fix-ups* are needed to generate instantiations. Fortunately, bytecode-editing tools [6,7,8] are available to assist in this programming task.

The main disadvantage of this approach, however, is the complexity and overhead in developing a pre-processor that fully supports that semantics of constrained parametric polymorphism. Consider, for example, the parametric class C:

```
class C<T implements Ifc>
{
    void m(Date date)
    {
        T t = new T(3);
        t.func1(t.func2(9.0), date);
        int x = t.fld;
    }
}
```

Figure 1 - Type Parameter Usage

In order for the pre-processor to transform the definition of class C into a definition that a Java compiler will accept, type parameter T must be stripped out. To do so, the pre-processor must replace all occurrences of T with a *placeholder type* that is guaranteed to correctly compile. When instantiations are generated, the class loader replaces these placeholder types by the actual type parameters specified in the program.

For class `C` above, the placeholder type must be a non-abstract class that has an integral field named `fld` and has methods `func1()` and `func2()` that will accept the parameters shown. The placeholder type cannot be abstract because it will be used in an allocation expression. This concreteness implies that all the methods declared in interface `IFC` must be given stub implementations. An appropriate constructor must also be implemented. Thus, generating placeholder types requires both the inspection type parameter usage in the source code and the use of Java reflection to inspect constraint types.

Generating placeholder types becomes more complex when F-bounded polymorphism is used. For example, if interface `IFC` in Figure 1 was replaced by polymorphic interface, `FIFC<T>`, the placeholder type for `T` would have to meet all the constraints declared for `FIFC`'s type parameter in addition to those already described.

The overhead of placeholder type generation increases as the number of type parameter constraints increase; both extends clause and implements clause constraints can be specified for type parameters. The constraining types can themselves have an arbitrary number of supertypes, all of which can affect the structure of the generated placeholder type. To help reduce compile-time overhead, placeholder types can be constructed directly as bytecode.

The technique of using placeholder types in extended-format bytecode was explored in previous heterogeneous implementations of parametric polymorphism in Java [1,2]. These implementations did not describe how placeholder types would be generated for code like that in Figure 1. One approach, of course, is to prohibit references to members of type parameters and disallow the use of type parameters in allocation expressions. These restrictions simplify placeholder type generation at the cost of reducing the expressiveness of the JL language implementation. Since language expressiveness was one of the main reasons we chose a heterogeneous implementation, we prefer to explore other implementation techniques rather than restricting the language.

As a final point, a fully integrated compiler approach, such as the one described in Section 2.2, allows the compiler to postpone the type checking of statements that use type parameters until instantiation-time. This ability to delay type checking allows placeholders to be simple names rather than actual types, so the complexity and overhead of generating placeholder types can be avoided under the modified compiler approach.

2.4 Source-to-Source Pre-Processor

Having considered and rejected all previous implementation choices, we now describe the system we plan to implement. The JL compiler's input is JL source files and its output is either JL *template files* or annotation Java class files. Parametric types are always compiled to template files, which are the result of removing most JL constructs from the JL source code. The JL semantic information expressed in the removed constructs is stored as a structured comment in the generated template file. Template files are used whenever parametric types are instantiated. Since template files are slightly modified versions of Java source files, the name source-to-source pre-processor is given to this implementation approach.

Source files that do not define parametric types, but do use JL language extensions, are compiled to annotated Java class files. Instantiation expressions in the source code are replaced by the names of the types they generate. If the instantiation's type doesn't exist, it's generated. Deep conformance type checking is performed when the **deeply** keyword is encountered. All occurrences of the JL keywords **deeply** and **propagate** are then stripped from the source and the stripped source code is sent to a Java compiler for compilation. Finally, the propagation information that was stripped out of the original source code is used to annotate the generated class files.

The class flattening optimization is an optional post-processing step that is applied to the class files of an instantiation. This optimization is implemented as bytecode-to-bytecode transformation based on the Java Application Extractor (JAX) [10] and invoked by the JL compiler just after an instantiated type is generated.

One disadvantage of the source-to-source pre-processor approach is that parametric types are fully instantiated at compile-time. This can lead to the generation of a large amount of almost identical code if a parametric type is instantiated with many different actual types. Many implementations of C++ templates suffer from this same problem and engineering techniques applied on an application-by-application basis are usually sufficient to solve memory usage problems.

The code explosion problem, however, is aggravated if the exact same instantiation is replicated in different packages. This latter problem can be avoided by putting all instantiations generated by an application in a designated package. This is the analog of the C++ technique of putting all of the template instantiations of an application in one shared library.

Another disadvantage of the source-to-source pre-processor approach that JL a template file must be distributed if new instantiations of a parametric type are to be generated. Since JL template files are slightly modified versions of Java source code, this may not be desirable in all environments. The same situation occurs when C++ template classes are implemented in header files to give the compiler access to their source code. In both environments, the code privacy problem can be ameliorated with license agreements, encryption techniques, or other ways of controlling access to data.¹

Even with these disadvantages, our source-to-source pre-processor should provide a fast, semantically powerful, heterogeneous implementation of parametric polymorphism for Java at a relatively cheap implementation cost. The most popular implementation of generics in Java, Generic Java [3], is a homogeneous implementation of parameterized types that does not support mixins. The implementation described here represents an alternate approach to genericity in Java.

3 High-Level Design

In this section, we discuss the JL compiler implementation using the source-to-source pre-processor approach introduced in Section 2.4. Our goal is to describe how the compiler carries out its major functions so that the feasibility, completeness and limitations of the design can be assessed. This description also allows us to estimate of the amount of work required to implement JL. We proceed by describing how parameterized types are compiled, how instantiations are generated, and how optimization is performed.

3.1 Parametric Polymorphism

In this section, we illustrate the use of type parameters in JL by example; details about JL's parametric polymorphism can be found in other JL documents [5]. Consider the following type definitions:

```
a. class C<T> {}
b. interface I<T extends I1> {}
c. class D<T implements I2<T>> {}
d. interface J<T extends I3> extends T {}
e. class E<T extends C1 implements I4, U implements I4> extends F<T> {}
```

In the above definitions, class C takes unconstrained type parameter T. Interface I takes type parameter T, which is constrained by interface I1. Class D takes type parameter T, which is F-bounded by I2<T>. Interface J is a mixin. Class E is a mixin that takes two type parameters: Type parameter T is a class that extends C1 and implements I4; a constrained type parameter U is also defined. Type parameter T is also used to constrain the mixed in supertype, F<T>, in class E.

Type parameters can be used in the bodies of their defining types wherever a type is allowed. This includes use in variable and array declarations, as formal parameters or return types in methods, in cast, instanceof or allocation expressions, or as actual type parameters in other type instantiations. Type parameters cannot

¹ One alternative would make JL template files more opaque by representing their parametric types not as source code, but as some serialized form of AST.

be hidden and their scope is the entire definition of the type in which they are defined, which includes the definitions of any nested types. Nested types can use type parameters defined in some enclosing type in their bodies and in supertype expressions.

3.1.1 Nested Parametric Types

Nested parametric types can also be defined in JL. Once a type parameter is defined in a parametric type, it cannot be redefined in any nested parametric type. Nested parametric types, however, can use type parameters defined in enclosing types as constraint types in their own type parameter definitions, in addition to using them in their bodies and in supertype expressions. All nested parametric types are static, thus qualified *this* references do not appear inside nested parametric types.²

Parametric types in JL are treated as type functions, not real types. This means that parametric types can subtype parametric types, but non-parametric types cannot. Parametric and non-parametric types can always subtype a non-parametric type or an instantiation of parametric type. A nested non-parametric type is allowed to subtype a parametric type using a type parameter defined in an enclosing type because the supertype expression is viewed as an instantiation (i.e., the type parameter in the supertype expression will always be bound before the compilation of the nested non-parametric type is attempted).

By default, **deeply** processing does not apply to public nested parametric types. This is in contrast to the way public nested non-parametric types are handled, which is to always apply deep conformance when **deeply** is specified [5]. The **propagate** keyword, however, can be used to enable **deeply** processing of selected nested parametric types, just as **propagate** can be used to enable **deeply** processing with selected non-public, non-parametric, nested types.

3.2 Compiling Parametric Types

When a parametric class or parametric interface is encountered in a source file, it's parsed and its AST representation is created. JavaCC and JJTree parser/generator tools are used to create the AST representation. The compilation of any parametric type is a depth-first, recursive process in which all nested parametric types are compiled before the enclosing parametric type is compiled. To compile a parametric type, the JL compiler:

1. Performs semantic checking on JL constructs.
2. Records the semantics expressed by the JL constructs.
3. Strips all JL constructs out of the AST.
4. Writes the recorded semantic information and the stripped source code to a template file.

The following sections describe how these steps are carried out.

3.2.1 Checking JL Constructs

The following invariants are enforced in the definitions of all parametric types:

1. Type parameters are defined only once in the scope of a type.
2. Type parameter names are different from all nested type names in their defining type.
3. The **propagate** keyword does not modify private constructors or private nested types.

3.2.2 Recording JL Semantics

The JL compiler collects semantic information for each parametric type. This semantic information collected is extracted from the JL constructs found in each parametric type's AST. Type parameter definition clauses, which include type parameter names and their constraints, are recorded. Each use of the **deeply** keyword outside type parameter clause is also recorded, as is each use of the **propagate** keyword on constructors and nested types. A list of nested parametric types is also recorded for each parametric

² We leave it to future work to explore the semantics and implementation of non-static nested parametric types.

type. Some of the information collected for a parametric type is used during the compilation of its nested parametric types.

3.2.3 Stripping the AST

All type parameter definition clauses, nested parameterized types, uses of **propagate**, and uses **deeply** are stripped out of a parametric type's AST representation. By the time this occurs, all nested parameterized types have been completely compiled. Instantiations that appear within type parameter definition clauses are stripped out with the clauses in which they appear; all other instantiations are left intact. Uses of type parameters and uses of the **This** virtual type are left unchanged.

3.2.4 Creating the Template Files

The compilation of a parametric type results in the creation of its template file and the creation of template files for each of its nested parametric types. Each template file contains two parts. The first part is a header that describes the JL semantic information pertaining to all type parameter definitions, nested type parameters that have been stripped out, uses of **propagate** and uses of **deeply**. This information provides the set of instructions for instantiating the parametric type when all its actual type parameters are specified. The second part of a template file contains the stripped source code that resulted from Step 3. JL templates differ from standard Java code in that instantiations and unbound type parameters (including *This*) can appear in template code.

3.3 Compiling Non-Parametric Types

In JL, non-parametric types can contain instantiation expressions, nested parametric types, uses of the **propagate** and uses of **deeply**. Non-parametric type can subtype other non-parametric types or instantiations, but non-parametric types cannot subtype parametric types. After parsing and creating the non-parametric type's AST, a depth-first, recursive traversal of all nested types and nested parametric types is performed. On each recursive call, the JL compiler:

1. Compiles any nested parametric types.
2. Processes instantiations.
3. Processes **deeply** keywords.
4. Processes **propagate** keywords.
5. Strips all JL constructs from AST.
6. Invokes Java compiler on stripped code.
7. Embeds **propagate** and nested parametric type information in the resultant class file.

The following sections describe how these steps are carried out.

3.3.1 Processing Nested Parametric Types

Nested parametric types are compiled according to the process described in Section 3.2.

3.3.2 Processing Instantiations

Step 2 is concerned with the instantiation of parametric types. Instantiation generate new Java types and the names of these types replace their corresponding instantiation expressions in the AST. The process of instantiation is described in detail in Section 3.4.

3.3.3 Processing Deeply

The **deeply** modifier may appear in extends or implements clauses that specify supertypes in a non-parametric type definition. When the JL compiler encounters **deeply**, it loads the types specified as constraints on the type being defined. The JL compiler then determines whether the type being defined conforms deeply [5] to each of the targets. To check for deep conformance, nested types defined in the constraining types are compared to the nested types defined in the AST. AST constraint checking results in one of three cases and determines whether compilation will continue or abort.

In the first case, a constraining nested type matches a nested type in the AST and the AST's nested type inherits from the constraining nested type as required. The check succeeds. In the second case, a constraining nested type does not match any nested type in the AST. In this case, the JL compiler generates a new nested type in the AST that inherits from the constraining nested type. This case also results in success. In the last case, the constraining nested type matches a nested type in the AST, but the AST's nested type does not inherit from the constraining nested type as required. In this case, compilation aborts with an error message.

3.3.4 Processing **Propagate**

The **propagate** keyword can be used in two distinct ways. When used as a modifier on nested types, **propagate** works in conjunction with **deeply** to determine what nested types must be present in the type being checked for deep conformance. See Section 3.3.3 and the appropriate JL specification [5] for details.

When used as a modifier on non-private constructors, **propagate** designates which constructors will participate in constructor propagation [5]. In the class being compiled, the constructors marked with **propagate** will interact with the constructors marked with **propagate** in the superclass. If the superclass's class file is annotated, the JL compiler extracts the propagation information and uses it to insert new constructors into the AST.

3.3.5 Stripping the Source Code

All **propagate** and **deeply** keywords are stripped from the AST. In addition, all nested parametric types are stripped out of the AST. Some **propagate** and nested parametric type information is maintained by the compiler for use in the final compilation step.

3.3.6 Compiling the Stripped Source Code

The stripped AST represents Java source code and is now compiled by a standard Java compiler.

3.3.7 Embedding Semantics in Class Files

Two types of information are embedded in the class files that result from the compilation in the previous step. Types in which the **propagate** keyword appears will have propagation information embedded as an attribute in their bytecode. Types originally defined with nested parametric types will have this information embedded in their bytecode. All compilers other than the JL compiler and all loaders will ignore this embedded information.

3.4 Generating Instantiations

Instantiations of a parametric type consist of the parametric type followed by a list of actual type parameters enclosed in angular brackets. All instantiations require an actual type parameter for each formal type parameter; JL does not support partial instantiation. The JL compiler can encounter an instantiation expression wherever a type is legal in Java. This includes as the supertype in an extends or implements clause, as the type in a variable or array definition, and as the type in an allocation, cast or instanceof expression. Instantiations can also appear as constraint types in formal type parameter clauses (Section 3.3.3).

To generate an instantiation, the JL compiler loads and parses the JL template file that represents the parametric type (Section 3.2). The immediate result of this parse is the creation of the parametric type's AST, which is distinct from the defining type's AST in which the instantiation is specified. The JL compiler next loads each actual type parameter specified in the instantiation. These actual types can themselves be instantiations, so this process is naturally recursive.

The instantiation of a nested parametric type requires that all type parameters in all enclosing parametric types be bound before the nested parametric type is instantiated. This binding information is passed to the nested parametric type during instantiation so that all type parameters within the definition of the nested parametric type can be bound. This binding information is also used to uniquely name the generated type.

Once all the actual type parameters have been loaded, they are validated against the constraints encoded in the JL template header. Constraint validation can include **deeply** processing (Section 3.3.3) and constructor propagation (Section 3.3.4).

Both **deeply** and **propagate** processing can result in dynamic additions to the parametric type's AST. These additions take the form of new constructors or new nested types that depend on the actual types specified for the instantiation. Since more precise type information is available at instantiation-time than is available when a parametric type is compiled into its template, instantiation-time additions are a superset of compile-time additions. For example, parametric type `P` below would not have any nested types propagated at compile-time, since `Object` does not have nested types. At instantiation-time, however, a class that has public nested classes could be specified for type parameter `T` and these nested classes would be propagated into the instantiated type.

```
class P<T deeply extends Object> {}
```

Assuming all processing described above succeeds, all occurrences of formal type parameters in the parametric type's AST are replaced by their actual types. The AST is then written to a file as Java source code. The name of the file is a mangling of the parametric type name, the names of all actual type parameters and, under some circumstances, the name of the **This** type. The Java compiler then compiles the source into a class file, which JL compiler annotates with propagation and nested parametric type information if necessary. The mangled type name is then substituted for the instantiation expression in the defining type's AST.

3.5 Input/Output Model

This section describes the high-level input/output model of the JL compiler by way of an example. For the purpose of illustration, we represent the type names of instantiations with the same representation that a programmer would use. The JL compiler, however, will mangle these type names in conformance with Java naming conventions. Our example shows the results of compiling and instantiating a nested parametric type. Consider the following class definition:

```
class A<T> {
    class B {
        class C<U> {
        }
    }
}
```

Figure 2 - Compiling Nested Types

In Figure 2, class `A<T>` contains a non-parametric nested class `B`, which contains a parametric class `C<U>`.

```
jlc A.jl ↪ A.jlt, A$B$C.jlt
```

The result of invoking the JL compiler with the file that contains this definition, `A.jl`, is shown above. JL compilation results in a template file for each parametric type contained in the input file. Template files end with the `jlt` suffix and type parameters are not represented in the name. The use of `$` in nested types follows the Java convention.

```
A<String> ↪ A<String>.class, A<String>$B.class
```

Once a template file exists for a parametric type, instantiations can be generated. The result of instantiating parametric type `A` with the actual type parameter `String` is shown above. In this case, two class files are generated, one for each class in which all type parameters in `A.jlt` can be bound. When appropriate, these class files will contain JL annotations as described in the previous sections of this document.

```
A<String>.$B.C<Date> ↪ A<String>$B$C<Date>.class
```

To see how a nested parametric type can be instantiated, we show the result of instantiating parametric type `C` with the actual parameters `String` and `Date` above. In this case, template `ABC.jlt` is used. The actual type parameter for class `A`, `String`, is available for use during the instantiation of nested parametric type `C`.

3.6 Class Flattening Optimization

Class flattening [5] is a compiler optimization that can be invoked as a post-processor to the JL compiler. When the JL compiler is used to statically generate instantiations, optimizations based on IBM JAX can be used to collapse the class hierarchy generated by JL.

3.7 Unsupported Features

The following features will not be supported in the initial implementation of the compiler:

1. An explicit type equation construct.

Standard instantiation syntax is the only way to use parametric types; the explicit type equation construct is superfluous and will not be implemented.

2. Semantic checking based on pattern matching.

This JL language feature is postponed to a later version.

3. Primitive types used as type parameters.

Only reference types and literals are allowed as parameter to parametric types (as in the previous version of JL).

4. Parametric methods.

This is a language design issue that needs further investigation. The object-oriented merits of such a construct can be argued from both sides.

4 Required Technologies

JL uses the following technologies in its implementation:

1. JavaCC/JJTree
2. Java Reflection
3. JL Loader
4. Bytecode Editor

The use of JavaCC, JJTree and Java Reflection as already been mentioned. The JL Loader allows the JL compiler to load template or class files by resolving their names in the context of the code currently being compiled and the `CLASSPATH` environment variable. The JL Loader determines the fully qualified name of all types it loads according the rules described in the Java Language Specification [9]. The JL compiler also needs to edit class files in order to embed JL information as attributes. These attributes will be ignored by standard Java compilers, but recognized by the JL compiler. There are a number of tools being considered to aid in this task [6,7,8].

5 References

1. O. Agesen, S. Freund and J. Mitchell. *Adding Type Parameterization to the Java Language*. OOPSLA 1997.

2. B. Bokowski and M. Dalm. *Poor Man's Genericity for Java*. Java Informations Tage, Nov. 12, 1998, Frankfurt, Germany.
3. G. Bracha, M. Odersky, D. Stoutamire and P. Walder. *Making the future safe for the past: Adding Genericity to the Java Programming Language*. OOPSLA 1998.
4. R. Cardone and C. Lin. *Comparing Frameworks and Layered Refinement*. Submitted for publication, <http://www.cs.utexas.edu/users/richcar/JavaLayers.html>.
5. R. Cardone and C. Lin. JL home page: <http://www.cs.utexas.edu/users/richcar/JavaLayers.html>
6. G. Cohen, J. Chase and D. Kaminsky. *Automatic Program Transformation with JOIE*. USENIX Technical Symposium, 1998.
7. M. Dahm. *Byte Code Engineering*. Java Informations Tage, Sept. 20, 1999, Dusseldorf, Germany.
8. Jikes Bytecode Toolkit. IBM AlphaWorks, <http://www.alphaworks.ibm.com/tech/jikesbt>.
9. J. Gosling, B. Joy and G. Steele. *The Java Language Specification, 2nd Edition*. Addison-Wesley, 2000.
10. F. Tip, C. Laffra, P. Sweeney and D. Streeter. *Practical Experience with an Application Abstractor for Java*. OOPSLA 1999.

6 Notes

1. Nested Parametric Type Template Generation

The method of generating templates for nested parametric described in this document takes place at compile-time. An alternative method would generate these templates every time the enclosing parametric type is instantiated. The benefits of the instantiation-time approach is that the bindings of type parameters in the enclosing parametric type are available at the time the nested type's template is created. Thus, instantiation is simplified because all type parameters that must be bound have their explicitly actual types specified (though the *This*-binding is an exception). One major drawback of the instantiation-time approach is that a template file is generated for each nested parametric type for each instantiation of an enclosing parametric type.

2. Name Mangling

JL's actual name mangling algorithm will not be trivial in that it must support:

- The optional inclusion of the *This*-binding type.
- The fully qualified type names for all type parameter actuals.
- Nested parametric type naming.
- Compliance with Java's naming restrictions.
- Co-existence with Java's package and nested type naming schemes.
- A search algorithm that can efficiently locate source, template and bytecode files based on their names as specified in source code.