Targeting the Java Virtual Machine with Genericity, Multiple Inheritance, Assertions and Expanded Types

Dominique COLNET and Olivier ZENDRA

LORIA
UMR 7503 (INRIA – CNRS – University of Nancy)
BP 239
54506 Vandoeuvre-lès-Nancy Cedex
France
`E-mail: {colnet,zendra}@loria.fr`
`http://SmallEiffel.loria.fr`

Abstract. The Java Virtual Machine is now widely available on many architectures and systems, which makes this virtual machine a very appealing target as an execution environment, not only for Java programs but also for Java extensions as well as other high-level languages.
In this paper, we focus on using the current Java Virtual Machine without modification to support assertions, genericity, multiple inheritance and expanded types. An experiment was realized by implementing a new back-end for our compiler, SmallEiffel, The GNU Eiffel compiler.

1 Introduction

One of the main reasons for the Java bytecode [LY96] existence is portability. Thanks to the tremendous momentum gained by Java [GJS96] in the last few years both in the academic and industrial worlds, a number of Java Virtual Machines are now available on a large variety of platforms, ranging from mainframes to smart cards and including personal computers and workstations. This seems to make the JVM bytecode a great target code for programs written not only in Java but also in many other languages, especially high level ones.

It is thus important to know to what extent the Java Virtual Machine makes it possible to support high level language concepts which are not — yet — part of the Java language standard, like genericity and assertions for design by contract.

In this paper, we describe how we target the JVM bytecode to translate Eiffel [Mey94] programs into a set of Java class files. Interesting features of Eiffel, a high-level statically typed object oriented language, comprise generic (or parametric) types, multiple inheritance, powerful assertions allowing programming with design by contract and user-defined expanded (or by value) types and base types which are handled as actual classes. We show in this paper some of the issues that translating Eiffel into Java bytecode may raise and how we have overcome them. We also discuss some of the shortcomings of our first implementation and in which way it can be improved.

The remainder of this paper is organized as follows. Section 2 explains the general principles underlying our compilation method. Section 3 describes how Eiffel types are encoded in the Java bytecode, while section 4 focuses on generic types and multiple inheritance. The implementation of the various feature calls is detailed in section 5, and that of assertions and design by contract in section 6. Section 7 addresses the issue of user-defined expanded objects. Section 8 discusses the relation between our work and some previous one. Finally, section 9 criticizes our current implementation and proposes future work directions, and section 10 concludes.

2 General principles of the translation

In order to translate high-level Eiffel source code to Java bytecode, a separate Java package is assigned to each Eiffel program compiled and, within it, each concrete live type is compiled into a specific Java bytecode file.

Early in our compilation process, a *static analysis* of the whole system is performed [CCZ97]. Starting from the system's root, that is the entry point composed of a root class and root method (equivalent to Java's `main` method), the whole *live code* is computed by recursively following object instantiations and method invocations. This allows the compiler to know about all live concrete types, including generic ones, and to know the inheritance hierarchy used. This first part of the compilation process is exactly similar to the one performed in order to target ANSI C code. In fact, this stage of the compilation is common to our Eiffel to C and our Eiffel to Java bytecode compilers.

The next phase of our compilation algorithm consists in *customizing* methods in each leaf class, including inherited methods, and possibly renamed and/or redefined ones, in what we like to call the "falling down" process [ZCC97]. Hence, a method defined in some ancestor is duplicated in each descendant and customized in the descendant's context. The inheritance graph is thus "flattened", since each generated class file contains everything pertaining to the corresponding Eiffel file and all its ancestors.

Our "flattened" generated classes inherit from a common ancestor, which we name `_any` and which contains utilities such as the manifest string initializers, the dispatch routines (see section 5.3), etc. Except the fact it is inherited by all classes, `_any` has no relation to either the ANY Eiffel class or the `Object` Java class.

3 Type mapping

3.1 Mapping of simple types

Simple basic Eiffel types are not an issue because each basic Eiffel type has its Java equivalent. For example, the Eiffel INTEGER type can be mapped as a Java `int`. Regardless the fact that basic Eiffel types are true classes – extra methods can even be added to the INTEGER type – mapping basic Eiffel types is easy.

Simple, these basic Eiffel types are directly mapped to the corresponding types in the Java bytecode, according to the following correspondence table:

Eiffel type	JVM bytecode type	Java type
INTEGER	I	int
REAL	F	float
DOUBLE	D	double
CHARACTER	B	byte
BOOLEAN	Z	boolean
POINTER	Ljava/lang/Object;	Object

Note that POINTER in Eiffel is a special basic type used to manipulate external entities. In this case, external entities are Java objects, hence the type "reference to `Object`".

Furthermore, since basic Eiffel classes are genuine classes, it is necessary to generate an associated Java class file, which will hold the operations that exist in the Eiffel type but not in the corresponding Java bytecode (primitive) type. For example, class INTEGER features an Eiffel instance method `to_string`, which does not exist on type `int` at the bytecode level, hence the need for a `my_package/integer.class` wrapper file.

3.2 Mapping of reference types

Normal Eiffel types are reference types, as in Java.

Their mapping is quite straightforward: each Eiffel reference type is mapped to one specific Java class file, included in the package corresponding to the compiled system. A class named HELLO_WORLD is thus coded in an `hello_world.class` file, in, say, an `hello_world` directory. Similarly, the Eiffel library type STRING is mapped to a `my_package/string.class` file, which is absolutely not related to `java.lang.String`.

Quite simply, Eiffel attributes are coded as instance variables in the Java bytecode object.

4 Genericity and multiple inheritance

Analysing of the whole system allows the compiler to know exactly which classes are used and, for generic types, which derivations are actually live and used by the system.

Indeed, since the generated code is customized as much as possible in order to get the best performance, all concrete derivations of a generic type are considered to be distinct types. For example, class ARRAY[G] is an Eiffel generic class, parameterized by the type of its elements. If a system uses both ARRAY[INTEGER] and ARRAY[STRING], two separate customized types are generated in the Java bytecode: `array0integerF` and `array0stringF`. Note the name mangling scheme we use, which makes it possible to have arbitrary nesting of type parameters as well as any number of type parameters.

Eiffel type	JVM bytecode type name
ARRAY[INTEGER]	arrayOintegerF
ARRAY[STRING]	arrayOstringF
ARRAY[ARRAY[ARRAY[INTEGER]]]	arrayOarrayOarrayOintegerFFF
DICTIONARY[STRING,INTEGER]	dictionaryOstring_integerF
DICTIONARY[ARRAY[STRING],INTEGER]	dictionaryOarrayOstringF_integerF

Multiple inheritance is also handled gracefully thanks to the system analysis and code customization performed.

Basically, multiple inheritance in Eiffel is solved entirely in the inheritance clause of the class[1], that is statically, at compilation time. Because all the generated code is "flattened" and customized, it is able to always work with accurate types (possibly polymorphic, see section 5.3) and can thus avoid problems present for example in C++ [Str86], like offset computation. There is thus no extra cost associated with the use of multiple inheritance compared to single inheritance, be it in terms of execution time or memory usage.

5 Feature calls

In Eiffel, feature calls consist both in method calls and field accesses. Indeed, in Eiffel method calls and attribute accesses are uniform: calling a method without argument relies on the same syntax as accessing an attribute of the object. The compiler is in charge of generating the appropriate code for these two cases.

Thanks to the results of the whole system analysis, our compiler is able to distinguish three kinds of feature calls in the Eiffel code: calls on `Current` (the Eiffel equivalent of `this` in Java), monomorphic calls on an object other than `Current`, polymorphic calls on an object other than `Current`.

5.1 Monomorphic feature calls applied to the `Current` object

The first kind of call, on `Current`, has properties which can be taken advantage of. Indeed, `Current` is never `Void` (equivalent of Java `null`) and thanks to customization its concrete type is unambiguously known at compile time. It is thus possible to generate a monomorphic call without testing the receiver existence. The following Eiffel code, written in class FOO:

```
Current.bar;  -- or simply: bar;
```

results in this bytecode:

```
aload_0   (load  Current from local #0)
invokevirtual my_package/foo.bar:()V
```

[1] To be more accurate, this is true except for the new Eiffel language extension `Precursor`, akin to `super` in Java.

Note that here bar is an Eiffel parameterless procedure call. An attribute read would look like some_var := bar;, since in Eiffel the uniform access property makes attribute access and parameterless function calls identical. The generated code for an attribute access would thus look like:

```
aload_0  (load  Current from local #0)
getfield my_package/foo.attribute:I
istore_1  (store in local variable  some_var)
```

5.2 Other monomorphic feature calls

When the call is of the second kind, that is, still monomorphic but not applied to Current, this Eiffel instruction:

```
a_foo.bar;
```

results in the following bytecode:

```
aload_1  (load  a_foo from local #1)
checkcast my_package/foo
invokevirtual my_package/foo.bar:()V
```

Note that the checkcast may be required by the JVM for security reasons (see section 4.8.2 of [LY96]), and is generally not removed by our implementation which currently does perform high-level optimizations but only little bytecode-level optimizations.

An attribute access is coded in a similar way, with getfield.

5.3 Polymorphic feature calls

Finally, in the most general case, when several receiver types are possible, a polymorphic call has to be generated. Let's assume a system with a class FOO and its heir SUB_FOO. Then, the parameterless procedure call:

```
some_object.bar;
```

produces this bytecode, assuming that some_object is of type FOO:

```
  aload_1 (load  some_object from local #1)
  invokestatic my_package/_any.switchFOObar:(Lmy_package/_any;)V
```

This bytecode simply calls a switchFOObar dispatch static routine, whose code is the following:

```
static switchFOObar descriptor: (Lmy_package/_any;)V
  0x00  0   aload_0 (load reference from local #0)
  0x01  1   instanceof my_package/foo
  0x04  4   ifeq 15
```

```
0x07  7   aload_0 (load reference from local #0)
0x08  8   checkcast my_package/foo
0x0B  11  invokevirtual my_package/foo.bar:()V
0x0E  14  return
0x0F  15  aload_0 (load reference from local #0)
0x10  16  checkcast my_package/sub_foo
0x13  19  invokevirtual my_package/sub_foo.bar:()V
0x16  22  return
```

This routine is a static one, which does the message dispatch sequentially, relying of the `instanceof` bytecode. Indeed, it must be kept in mind that in Eiffel it is possible to *rename* inherited routines.

It it thus not possible, in case of renaming, to use the standard JVM dispatch mechanism provided by an `invokevirtual my_package/foo.bar:()V` instruction, which is based on fixed routine names. Using `invokevirtual` to actually do the dispatch would imply using some mangled names, effectively hiding the renaming problem, but also making the bytecode much more difficult to read and debug. Here, `invokevirtual` is simply used inside a dispatch branch for a monomorphic call on an instance, which thus amounts to the second case seen above.

Furthermore, because of the uniform access property, a parameterless function may, in some descendants, be redefined as an attribute. A `getfield` would thus have to replace the `invokevirtual` in some of the branches of a dispatch site.

Also note that the code of the dispatch function could have been inlined, but we did not do so in order to keep the code compact (see [ZCC97]).

We will further discuss these implementations in section 9.

5.4 Feature calls on Eiffel primitive types

As we mentioned, the Eiffel class INTEGER leads to an `integer.class` bytecode file. Obviously, basic operations are directly mapped to basic JVM operations on primitive types. For example, an INTEGER addition results in an `iadd` bytecode instruction. However, more complex routines — i.e. routines which are not basic "bricks" and can be coded in Eiffel — such as `max`, are implemented as static routines in class `integer`.

The same applies to the other basic classes mentioned in section 3.1.

6 Assertions and design by contract

Eiffel offers a powerful assertion mechanism, allowing an easy implementation of design by contract. Java does not feature such a mechanism for the time being, although there is work under way in this direction [SUN99]. Our work is an example of a complete assertion mechanism implemented using the existing Java bytecode. Note that assertions can be turned on or off at compilation time, according to the degree of assertion checking requested, which makes debugging much easier. Different levels of assertion checking will thus result in the production of different bytecode, containing more or less assertion code.

We explain in this section some aspects of the implementation of the three major types of assertions which are useful to implement design by contract: class invariants, method preconditions and method postconditions.

The class invariant is intended to guarantee that only correct instances (objects) of a given type may be created. Invariants are cumulative throughout the inheritance tree: every class inherits the invariants present in its ancestors and adds them to its own in a conjunctive manner (invariants can only be strengthened).

During system analysis, all the invariants pertaining to a specific class and its ancestors are gathered and then generated in the "flattened"corresponding Java bytecode file. These invariant assertions are put in an instance procedure named "invariant", which is called each time there is a qualified call on a feature of an object of the type in which the invariant is defined. Note that, in case of multiple inheritance, invariant clauses may be aggregated that come from several "branches".

The following example illustrates class invariant coding:

```
class FOO
...
feature
   bar is
      do
         ...
      end;

invariant
   boolean_expression1;
   boolean_expression2;
end;

class USE_FOO
...

use_foo is
   do
      ...
      a_foo.bar;
   end;
end;
```

The call to `foo.bar` is translated as a normal call as we have explained in section 5. However, when invariants are on, additional code is generated just before the call in the client code to trigger the invariant check:

```
aload_1   (load  a_foo from local #1)
dup       (duplicate top of stack)
checkcast my_package/foo
invokevirtual my_package/foo.invariant:()V
checkcast my_package/foo
invokevirtual my_package/foo.bar:()V
```

The invariant routine simply implements boolean expressions and check whether they are true or not. In case they are not, a static method dedicated to error handling — `SmallEiffelRuntime.runtime_error` — is called and displays the information related to the violated assertion.

Method preconditions are assertions which are triggered when a method is being called, just before executing the first instruction of its body but after evaluating the method parameters, if any. Preconditions are intended to specify the conditions in which a method may be called, that is, for example, to check the validity of its arguments.

The generation of bytecode to implement preconditions is not unlike the implementation of invariants. Preconditions are accumulated from all the ancestors of the method considered, knowing that their semantics according to the design by contract rules is to be weakened from an ancestor to its heir.

But unlike with invariants, when preconditions generation is on and a method with preconditions is called, the caller is not changed at all. Instead, extra code is inserted at the beginning of the callee's body to check the boolean expressions composing the method preconditions. Assertion violations are handled the same way as for invariants.

Method postconditions are the last important kind of assertions for design by contract programming. They precise what a method can guarantee its callers, and also allow, albeit in a less important way, to control the behavior of the routine body. Postconditions are accumulated from ancestor to heir, each level of the inheritance tree reinforcing the postcondition, in a way similar to what is done for invariants. Like for preconditions, the caller code is not modified, whereas the callee is, by insertion of extra code after the routine body, just before it returns, to check for the method postconditions.

One important problem when enabling assertions is to avoid infinite recursion. For the time being, the solution implemented in out Java bytecode generation is an effective but crude one. It consists in setting and accessing a global flag, `check_flag`, contained by the `_any` class, the common "utility" ancestor. Thus, at the beginning of assertion code — be it invariant, precondition of postcondition code — this flag is checked. If it is not set, the assertion code sets it and executes, otherwise the assertion code is skipped. Obviously, this does avoid any recursion problem, but is a bit too crude since an assertion clause should not be checked only if this very same clause is already being checked. To implement this, it is necessary to implement one flag per assertion clause, that is one for each class invariant clause, one for each method precondition set and one for each method postcondition set. This has not yet been implemented in our Java bytecode generator, although it has been successfully in our C code generator.

7 User-defined expanded types

Like C++, Eiffel allows the user to declare objects which are allocated in the stack or inside some other object. Such *expanded* types are useful when runtime indirections are to be avoided, since objects will be accessible directly rather than through references.

Unfortunately, except for primitive types (`char`, `int`, `float`, etc.), all Java objects are handled by reference. It is not possible to allocate a complex object either in the JVM stack or inside a single attribute of another object. Splitting a complex object does not seem to be a realistic solution, for example when some function has to return a complex expanded object. If feasible, it seems to us that such a splitted implementation would be at least tedious.

Fortunately, as mentioned in the [Mey94] Eiffel reference manual, allocation without reference is not mandatory and is still a compiler designer decision, provided the expanded semantics is guaranteed. Thus we have decided to manage expanded types like ordinary ones: they have their own corresponding class file and are allocated in the heap.

To preserve the semantics of Eiffel, each time some user-defined expanded object is passed as an argument of some routine, the corresponding expanded object is *duplicated* in the heap and the new reference is passed to the callee. Obviously, even though we are able to preserve the semantics of expanded types, their actual "expandedness" is not. Thus an important raison d'être of these objects, avoiding reference indirections, is lost. Also, because Java — like Eiffel — rightly prohibits explicit freeing of objects, it is not possible to manually free the memory used by the copied expanded object immediately after the call, which requires more memory. Furthermore, since there is no need to perform dynamic dispatch on expanded types, our C implementation of expanded objects also avoids the extra type identifier field in expanded objects, whereas it is not possible to save this memory in our Java bytecode implementation, where expanded types are created the same way as ordinary types. As a consequence, the garbage collector overhead is certainly important when expanded Eiffel objects are used with our JVM implementation.

The implementation of user-defined expanded types in Java bytecode thus seems bound to being rather inefficient, which contrasts with one of the goals of expanded Eiffel objects as well as with our C implementation.

8 Related work

Although the customization or specialization of object-oriented programs has been a technique known for some time [CU89,CU90], it seems it has lead to very few published work concerning the Java language or Java bytecode [SLCM99]. Our work tends to confirm that customization of high-level languages can be ported from one target low-level code (ANSI C) to another (the Java bytecode) without loosing its interest and potential performance improvements.

Problems raised by the implementation of Eiffel user-defined expanded types are somewhat related to *escape analysis* [PG92], which has recently been subject of much research work [GS98,CGS$^+$99,Bla99,BH99]. Many escape analysis works also rely on whole system analysis to determine which objects are not escaped. A non escaped object, like an (actually) expanded object, can be allocated in the stack in order to speedup execution time and to reduce the workload of the garbage collector. The main goal of escape analysis is to gain efficiency, mostly by removing unnecessary synchronizations on non-escaped objects and by allocating them in the stack. The latter is also an important aspect featured by user-defined expanded types.

A similar effort has been done on [DC98] where "inlined objects" somehow resemble Eiffel expanded objects. Their idea is to allocate (inline) small objects *inside* larger heap-allocated ones, without any reference, thus saving the memory corresponding to the removed pointer and the time needed to dereference it. This is quite conform to the semantics and intent of Eiffel expanded objects. One difference with our implementation is that in Eiffel expanded objects are supposed to be explicitly declared by the user whereas the [DC98] implementation does automatic inlining (or expansion). Another difference, as we previously mentioned, is that currently we use references to implement our expanded objects in a simple way in Java bytecode. Since their system was a C++ one, having access to machine-level instructions, it did not have the same constraints as ours and objects could be implemented in a truly expanded way.

From the language design point of view, the addition of expanded types to the Java language may even be a subject of discussion. Indeed, it can be argued that in most cases expanded types avoid the need for looking for escaped objects. However, an automatic determination of objects which are to be allocated in the stack seems to be a higher-level and more general concept (in the same fashion, both Eiffel and Java agree on the fact that memory handling is done by the garbage collector and not by the programmer).

Even if we do not want to debate this in this paper, we can remark that stack allocation in the JVM seems to be a crucial issue both for expanded objects and efficient implementation of non-escaped object.

Adding genericity to Java has also been the topic of major interest in the last few years [AFM97,MBL97,Tho97,BOSW98,SA98,CJ98,Dug99]. As we have shown with our system, whole system analysis combined with duplication and customization allows a reasonable and efficient implementation [CZ99]. Furthermore, whole system analysis does not appear to be mandatory here because it seems possible to generate dynamically (or in a lazy way) the missing needed derivation. For example, if the generic derivation for ARRAY[INTEGER] is already generated and loaded, it can be used, otherwise a new customized version for ARRAY[INTEGER] has to be compiled before using it. This is possible with the JVM which allows dynamic loading of new class files.

Multiple inheritance seems more difficult to implement without global system analysis. Indeed, thanks to whole system analysis and customization, dynamic dispatch in case of multiple inheritance is not an issue and does not cause more runtime overhead than single-inheritance dispatch. When routines are not customized to each type, it is more difficult, although not impossible, to select an efficient object field implementation [GS99].

9 Discussion and future work

The implementation of feature calls we presented in section 5 seems a rather direct translation in Java bytecode of the concepts present in our source language, Eiffel. Indeed, instance calls such as `foo.bar` are implemented with instance method invocations, such as `invokevirtual my_package/foo.bar:()V`.

This seems the logical thing to do, especially in a first implementation of a compiler targeting the Java virtual machine. However, there are hidden costs inherent to this strategy. Indeed, one must keep in mind that `invokevirtual` is not only a call on an instance, but above all a *dispatched* one. In the cases we presented, there is in fact not need for the JVM to do late binding, either because the call can be solved as a monomorphic one (sections 5.1 and 5.2) or because the code we generate already performs the dispatching "manually" (section 5.3).

It thus seems possible to implement these instance method calls as *static* method calls whose first parameter would be the receiver of the source language (`Current`). In case of a truly polymorphic call, the dispatch code we generate can take care of checking the first parameter type and of calling the corresponding static routine. This is very similar to what we have done before when implementing our Eiffel to C generator [CCZ97,ZCC97]. The advantage of this implementation would be to suppress the redundant dispatch phase currently performed by the JVM in `invokevirtual`, replacing it a by a static call. This is very likely to result in a significant speedup of our generated code.

Another performance bottleneck in the Java bytecode currently generated is the series of sequential `instanceof` tests performed to check the receiver type and do the dispatch (see the example in section 5.3). This is very costly and does not scale well, since it carries a cost which is linear with the number of possible types at the considered call site. It seems possible to also use in this case binary branching code which, as we have shown in previous papers, features very good performances. This would imply adding an extra integer attribute to all the types whose instances are subject to dispatched calls. This attribute would hold the type identifier and would be used to perform the dispatching thanks to the binary branching tree. Since it would imply attribute accesses and tests instead of a sequence of `instanceof`, this code may not be optimum for all dispatch sites. It is thus likely that the most optimum solution would mix several solutions, as in [CC99], that is probably short sequences of `instanceof` tests for dispatch sites where only a few types are possible, and binary branching code for others.

This is work which we plan to address in the near future.

10 Conclusion

We have reported in this paper on our experience on how the Java bytecode — without any modification or extension — could be used as an execution environment for programs written in langages including some high-level features which are not — yet ? — part of the Java language, such as type genericity, multiple inheritance and assertions for design by contract.

We have explained the remaining issues for user-defined expanded types, because of the impossibility in Java bytecode to allocate complex objects directly in the stack. This point is relevant not only for our implementation of user-defined Eiffel expanded objects, but also for performance issues related to escape analysis, as previous works have shown. In that respect, allocation of complex objects into the stack would appear as an significant issue for the JVM to truly become the universal multi language virtual machine.

It is nonetheless clear that the current Java bytecode, being a quite powerful "portable assembly code", is a very good target for many higher level object-oriented languages. When developping our Eiffel to Java bytecode compiler, we were indeed able to reuse most of the front-end and core of our Eiffel to ANSI C translator, adapting almost only the code generation back-end. Future improvements in our work are likely to be focussed on the later stage.

References

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32 of *ACM SIGPLAN Notices*, pages 49–65. ACM Press, October 1997.

[BH99] Jeff Bogda and Urs Hölzle. Removing Unnecessary Synchronization in Java. In *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34 of *ACM SIGPLAN Notices*, pages 35–46. ACM Press, October 1999.

[Bla99] Bruno Blanchet. Escape Analysis for Object-Oriented Languages. Application to Java. In *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34 of *ACM SIGPLAN Notices*, pages 20–34. ACM Press, October 1999.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33 of *ACM SIGPLAN Notices*, pages 183–200. ACM Press, October 1998.

[CC99] Craig Chambers and Weimin Chen. Efficient Multiple and Predicate Dispatching. In *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34 of *ACM SIGPLAN Notices*, pages 238–255. ACM Press, October 1999.

[CCZ97] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Joint Modular Languages Conference, JMLC'97*, volume 1204 of *Lecture Notes in Computer Sciences*, pages 67–81. Springer-Verlag, 1997.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34 of *ACM SIGPLAN Notices*, pages 1–19. ACM Press, October 1999.

[CJ98] Robert Cartwright and Guy L. Steele Jr. Compatible Genericity with Run-time Types for the Java Programming Language. In *Proceedings of 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33 of *ACM SIGPLAN Notices*, pages 201–215. ACM Press, October 1998.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Language. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'89)*, volume 24 of *ACM SIGPLAN notices*, pages 146–160, 1989.

[CU90] Craig Chambers and David Ungar. Interactive Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, volume 25 of *ACM SIGPLAN notices*, pages 150–164, 1990.

[CZ99] Dominique Colnet and Olivier Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, pages 341–350. IEEE Computer Society, June 1999.

[DC98] Julian Dolby and Andrew A. Chien. An Evaluation of Automatic Object Inline Allocation Techniques. In *Proceedings of 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33 of *ACM SIGPLAN Notices*, pages 1–20. ACM Press, October 1998.

[Dug99] Dominic Duggan. Modular Type-Based Reverse Engineering of Parameterized Types in Java Code. In *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34 of *ACM SIGPLAN Notices*, pages 97–113. ACM Press, October 1999.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[GS98] David Gay and Bjarne Steensgaard. Stack allocating objects in java. Technical report, Microsoft Research, November 1998.

[GS99] Joseph (Yossi) Gil and Peter F. Sweeney. Space- and Time-Efficient Memory Layout for Multiple Inheritance. In *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34 of *ACM SIGPLAN Notices*, pages 256–275. ACM Press, October 1999.

[LY96] Tim Lindholm and Franck Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison Wesley, 1996.

[MBL97] Andrew C. Myers, Joseph A. Blank, and Barbara Liskov. Parameterized types for Java. In *24th Annual ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages (POPL'97)*, pages 132–145, January 1997.

[Mey94] Bertrand Meyer. *Eiffel, The Language.* Prentice Hall, 1994.

[PG92] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, volume 27 of *ACM SIGPLAN Notices*, pages 117–127, July 1992.

[SA98] Jose H. Solorzano and Suad Alagić. Parametric Polymorphism for Java: A Reflexive Solution. In *Proceedings of 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33 of *ACM SIGPLAN Notices*, pages 216–225. ACM Press, October 1998.

[SLCM99] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards Automatic Specialization of Java Programs. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Sciences*, pages 367–390. Springer-Verlag, 1999.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[SUN99] Proposal: A Simple Assertion Facility For the Java[tm] Programming Language.
`http://java.sun.com/aboutJava/communityprocess/jsr/jsr_041_asrt.html`, November 1999. Sun MicroSystems Inc.

[Tho97] Kresten Krab Thorup. Genericity in Java with Virtual Types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Sciences*, pages 444–471. Springer-Verlag, 1997.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *Proceedings of 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32 of *ACM SIGPLAN Notices*, pages 125–141. ACM Press, 1997.