Towards An Extensible Virtual Machine

Chandrasekhar Boyapati
MIT Laboratory for Computer Science, Cambridge, MA 02139
chandra@lcs.mit.edu
Area Exam Report: January 2002

Abstract

The Java Virtual Machine Language (JVML) is rapidly emerging as the de-facto standard for representing portable code and Java Virtual Machines (JVMs) are increasingly being used as standard platforms for running applications. But how suitable are the JVMs and the associated JVML for this purpose? This paper argues that JVML has a serious drawback in that it is not extensible enough. This lack of extensibility hinders the process of deploying new innovations in applications that run on the JVM platform.

This paper also describes how a standard for portable code can be designed to be significantly more extensible than JVML. The paper explores the issues involved in designing such an extensible virtual machine (EVM), and reviews some recent research on formal techniques like type systems, logic frameworks, and analysis algorithms to show how an EVM can safely execute untrusted code.

1 Introduction

Standardized protocols and languages are essential in any heterogeneous networked environment. For example, TCP/IP has long been the standard way of exchanging bytes and has been fundamental to the growth of the Internet. More recently, XML is increasingly being accepted as the standard way shipping data across the web. Similarly, the Java bytecodes, or the Java Virtual Machine Language (JVML) [30] is becoming the de-facto standard for shipping computation and Java Virtual Machines (JVMs) are increasingly being used as a standard platform for running applications—so much so that they are beginning to resemble operating systems in the scope of their functionality.

The idea of standardized machine independent program representations is quite old. The first intermediate language UNCOL (UNiversal Computer Oriented Language) [46] was proposed in 1961 for use in compilers to reduce the development effort of compiling many different languages to many different architectures.

There are many reasons why portable code is useful in networked environments. In terms of efficiency, when repeated interactions with a remote site are needed, it can be more effective to send the computation to the remote site and to interact locally. In terms of extensibility, mobile code supports a far more flexible programming model compared to the classic client-server paradigm. And if heterogeneous sites have to exchange code, then the sites have to have a standard format for representing code.

The Java Virtual Machine Language (JVML) is a portable intermediate code representation designed at Sun Microsystems in conjunction with the Java [20] programming language. JVML runs on top of an abstract computing machine known as the Java Virtual Machine (JVM) [30]. JVMs have been deployed on a wide range of architectures and operating systems and are available on most modern machines.

JVML has been carefully designed with portability in mind. Unlike C and C++, there are no implementation dependent aspects of the specification. For example, uninitialized fields are always set to their default values. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them. An `int` always means a signed two's complement 32 bit integer, and a `float` always means a 32-bit IEEE 754 floating point number. The order of evaluation of sub-expressions in a complex expression is well-defined. The JVM platform specification also includes a set of standard libraries that provide, among other things, general purpose data structures, support for graphical user interface, and access to network communication. Programs written using these libraries are supposed to run unchanged on any standard JVM.

In addition to portability, another significant feature of the JVM platform is its security. JVML is a type safe language. Type safety forms the basis behind the security provided by a JVM. A JVM statically verifies the absence of certain errors in code originating from untrusted sources before executing the code. A JVM also uses a technique known as stack inspection [50] to provide fine-grained access control. Code from multiple principals having different access privileges can co-exist in the same address space and call one other. Thus, unlike conventional operating systems like Unix [33] which rely on processes executing under the control of a privileged kernel, a JVM provides security without any hardware support and with little runtime overhead.

But one significant drawback of JVMs is that they are not extensible enough. JVML is a high-level language and therefore it is limited in its flexibility. JVML was designed to closely resemble its source programming language Java. But the goals that guide the design of a source level language like Java are not necessarily the same as the goals of a common intermediate language. For example, making a source level language object-oriented may be a good design choice because of software engineering reasons, but making a common intermediate language object-oriented is not necessarily the right design choice.

Over the last few years, researchers have proposed several extensions to the JVM platform that enhance the JVM functionality in various ways. But because JVML is not sufficiently flexible, many of these extensions require JVM modifications. Unfortunately, the only way any of these extensions can be deployed in the real world is if Sun Microsystems decides to include it as part of the JVM standard.

Ideally, it should be just as easy for a user to download and install any of these extensions as it is to download and run application code. This can be possible if the virtual machines exposed a more extensible interface. Like other researchers have argued for having extensible operating systems [14, 4], I believe that it is necessary to have virtual machines that are more extensible. That way, the virtual machine architecture will impose less barriers on new innovations and experimentations. The primary goal of this paper is to build the case for extensible virtual machines.

This paper also describes how a standard for portable code can be designed to be significantly more extensible than JVML. The paper explores the issues involved in designing such an extensible virtual machine (EVM), and reviews some recent research on formal techniques like type systems, logic frameworks, and analysis algorithms to show how an EVM can safely execute untrusted code.

The rest of this paper is organized as follows. Section 2 lists several research proposals that require JVM modifications, motivating the need for having extensible virtual machines. Section 3 describes how an EVM can be designed starting from first principles. Section 4 examines other prominent intermediate languages proposed in literature. Finally, Section 5 presents our conclusions.

2 Motivating Examples

This section builds the case for extensible virtual machines. It presents several research proposals that enhance the JVM platform in various ways. In each case, the best way to implement the extension requires JVM modifications. Other JVM compatible implementations either compromise on the semantics of the system or involve significant performance penalties or both.

2.1 Parameterized Types for Java

Parametric polymorphism [37, 1, 8, 10, 49] is recognized as a key language mechanism for augmenting the expressiveness and safety of a programming language. It provides the ability to abstract a piece of code from one or more types, making the code reusable in many different contexts. Many JVM compatible approaches have been proposed to add parameterization to Java. These approaches can be broadly classified as follows.

1. Type erasure [8, 10]: It is the most common technique and is based on the idea of deleting the type parameters (so $\mathsf{Stack}\langle\mathsf{T}\rangle$ erases to Stack). Casts are inserted to recover the erased type information wherever it is needed in the transformed program text. GJ [8], the candidate solution for future releases of Java, relies on this technique. The primary disadvantage of type erasure is that it limits the expressiveness of the language. Since objects of parametric types do not carry all the type information at runtime, type erasure cannot support some type dependent primitive operations such as `new T[...]`.

2. Code duplication [1]: Polymorphism is supported by creating specialized classes/methods, each supporting a different instantiation of a parametric class/method. This technique supports a more expressive form of parameterized classes than type erasure, but it often leads to an unacceptable footprint in memory and disk space.

3. Type passing [37, 49]: In this technique, information on type parameters is explicitly passed to code requiring them. This technique involves significant space and time overhead at runtime.

The lesson in each case appears that if the virtual machine does not support polymorphism, the end result will suffer. While there are known techniques for supporting parametric polymorphism efficiently and without language restrictions by modifying the virtual machine [37, 22], these techniques cannot be deployed in the real world unless they become part of the JVM standard.

2.2 Persistent Java

Persistent object systems offer a simple yet powerful programming model that allows applications to safely share objects both in space and time. Such systems have distinguished objects known as persistent roots. All objects reachable from persistent roots are automatically stored in persistent storage by the system—the rest of the objects are garbage-collected.

Since Java has no persistence model built into it, many research systems have been proposed that add persistence to Java. These include PJama [3], JPS [6], GemStone/J [17] and PSEJ [42]. Of these, only PSEJ is JVM compatible. PSEJ uses the technique of bytecode rewriting. Bytecode rewriting has become an established technique for extending Java. But PSEJ has severe problems both in terms of its semantics and its performance. For example, PSEJ does not support garbage collection of persistent objects.

To support the persistent programming model, an object-oriented virtual machine requires the ability to traverse the object graph, ability to install hooks into method calls and field accesses, and the ability to replace objects with stub objects, among other things. These can be efficiently implemented in a modified JVM. In fact, all the other systems mentioned above—PJama, JPS, and GemStone/J—use modified JVMs to support persistence. These systems provide the natural semantics for persistence, and are fairly efficient as well. This once again suggests that the JVM interface is too rigid to support many extensions.

2.3 Software Evolution in Persistent Java

Software systems evolve over time to meet new demands. Persistent object systems contain long lived objects, so any practical persistent object system must provide a mechanism for upgrading the persistent objects. These upgrades involve changes to the code implementing the persistent objects, as well as changes to the persistent objects themselves.

Much research has been done on software evolution in persistent object systems. PJama [3, 12], JPS [6, 31] and GemStone [17, 40] support some form of software evolution using modified JVMs. Software evolution cannot be implemented on unmodified JVMs.

2.4 Reflective Interface for Java

Metaobject protocols [23] offer a principled way of extending the behavior of programs. Metaobjects can be used to transparently implement non-functional requirements such as fault tolerance, security and distribution [52]. The Java Reflection package java.lang.reflect only provides the ability to introspect a program but not to alter program behavior.

There are a number of extensions to Java that address this limitation. Many of these extensions including AspectJ [24], Kava [52], Dylang [53], and Javassist [11] use the technique of code rewriting to be compatible with standard JVMs. However, these extensions have limited reflective capabilities. In particular, they are limited in their ability to dynamically alter the behavior of a program. Other implementations like Guarana [39] and MetaXa [19] provide a more flexible reflective interface, but they use customized JVMs to do so.

2.5 Region Based Memory Management

Many software systems such as file servers and database servers require fine-grained control over data representations and memory management. High level, type safe languages like Java fail to give programmers the control needed to build low level systems. Low level languages like C avoid these drawbacks, but they admit a wide class of safety violations such as buffer overruns, dangling pointer dereferences, and memory leaks.

Region based memory management [48] offers an alternative to the above systems. Languages like Cyclone [21] use regions to offer programmers significant control over memory management without violating memory safety. Cyclone is a type safe language, yet Cyclone programs cannot be translated to JVML. This is because JVML is a high level language with automatic memory management and garbage collection, and it provides no way to get around that.

2.6 Consistent Java RMI Semantics

Java supports remote method invocation (RMI) [43], which is a form of RPC and is based on the Modula-3 network objects [5]. But the Java RMI semantics for argument passing are inconsistent. While arguments to local methods calls are passed by reference, arguments to remote methods are sometimes passed by reference and sometimes by value.

Ideally, all arguments to methods should be passed by reference to provide consistent semantics for method calls. But suppose an object x residing on one machine is passed by reference as an argument to a remote method on a second machine. If the remote method repeatedly uses x, then it will generate a lot of network traffic. One way to reduce the amount of network traffic is to migrate x to the second machine. But this cannot be implemented on standard JVMs because standard JVMs do not support object migration. This is yet another example where the inflexibility of the JVM interface prevents developers from implementing their ideas on standard JVMs.

2.7 Partial Classfile Loading

Sometimes, distributed applications may run on low bandwidth networks. To reduce the amount of code transferred on such systems, some researchers have proposed splitting classfiles into frequently used hot portions and infrequently used cold portions. The cold portions are shipped only when required [26]. But the bytecode verifier in a JVM verifies code on a per-class basis and the JVM loads code on a per-class basis, making it difficult to implement this optimization on standard JVMs.

2.8 JVM Optimizations

Besides projects that enhance the JVM interface, there are numerous other projects that focus on improving JVM performance. Many of them use static analysis to perform optimizations such as removal of redundant null pointer checks, array bounds checks and runtime typecasts. Projects such as Jalapeno [9], Marmot [15], and Flex [16] use customized JVMs to implement such optimizations.

Unfortunately, application vendors shipping JVML cannot use the optimization techniques mentioned above to optimize their code. This is because there is no way to represent the optimized code in JVML. For example, there is no way to indicate in JVML that a particular array access is safe and does not need a runtime bounds check.

3 Design of an Extensible VM

This section describes how a portable and secure intermediate language (and an associated virtual machine) can be designed starting from first principles, while keeping the language as extensible as possible.

3.1 Instruction Set Architecture

The essence of the problem with JVML as a general purpose instruction set is that it is a CISC—a high level, highly encoded instruction set that is carefully tuned to the demands of the Java language. The problem with CISC instruction sets is that they are brittle encodings. If a computation fits the instruction set exactly, things are good. The computation can be encoded compactly and executed efficiently. But if a computation is just slightly different, there is no simple, efficient encoding. The numerous examples presented in Section 2 illustrate this point.

I therefore believe that a standard for representing portable code should consist of a low level, RISC-like instruction set. The resulting extensible virtual machine (EVM) platform will give application developers tremendous flexibility in implementing their ideas and innovations. Compilers at the code producer side will have more room for maneuver when producing optimized code for the EVM platform. In fact, all the projects presented in Section 2 that required JVM modifications can be expressed efficiently in a low level instruction set.

The main problem with low level code is that it is hard for the VM to ensure any form of security. JVML, on the other hand, is a type safe language. JVMs statically type check programs before running them, and type correct JVML programs cannot cause certain security violations. The next few sections will explore how an EVM can also provide a secure execution platform.

3.2 Mechanisms for Providing Security

Traditional operating systems like Unix rely on processes executing under the control of a privileged kernel to provide security. But this approach requires special hardware which may not be available on many platforms. Therefore, this approach cannot be used for portable code. The rest of this section discusses the different security mechanisms that are suitable for a portable platform for running code.

Cryptographic Signatures: The use of cryptographic signatures is one mechanism to provide security. In this approach, the code producer (or some trusted authority) signs the code certifying that the code obeys certain security policies. Code consumers run the code only if they trust the signing authority. The problem with this approach is that there are too many application vendors, and hence too many authorities to be trusted. Moreover, this approach only protects code consumers from malicious code. Signatures do not offer any protection against unintended bugs.

Dynamic Checks: Another way a system can provide security is by the use of dynamic checks. Dynamic checking can be implemented either by inserting code around unsafe instructions, or by hard coding the VM implementation to perform these checks. The problem with dynamic checking is that it degrades performance because of the runtime overhead involved.

Static Checks: A system can also provide security through static checking. In this approach, the VM statically verifies the absence of certain errors before running code. This approach avoids the above-mentioned problems associated with signatures and dynamic checks, and is ideally suited for portable code. The problem, however, is that it is often hard to come up with statically verifiable proofs of safety properties for low level code. The next section describes the state of the art in static checking of low level code.

3.3 Static Checking of Low Level Code

Much recent work in static checking of low level code focuses on using types and logics to reason about the safety of programs. This section explores some of the prominent research projects in this area.

Typed Assembly Language [36] : Typed assembly language (TAL) is designed for environments where untrusted low level code must be checked for safety before being executed. TAL is based on a generic RISC assembly language. Its static type system provides support for enforcing high level language abstractions. TAL is powerful enough to automatically generate well-typed code from high level languages like ML. The typing constructs in TAL also admit program transformations like CPS and closure conversion as well as most conventional low level compiler optimizations such as register allocation, copy propagation, constant folding, and dead code elimination.

FLINT [45, 28]: The FLINT project provides another framework for generating typed low level object code from high level languages. The FLINT type system is general enough to support multiple high level source languages (including object oriented languages like Java and languages with higher order functions like ML) as well as multiple low level target languages. FLINT can not only generate proofs for memory safety like TAL and PCC (discussed below), but FLINT can also generate proofs of advanced properties such as for type and effect systems [32].

Proof-Carrying Code [38]: Proof-carrying code (PCC) is another framework for verifying safety properties in low level programs. PCC encodes the relevant operational content of simple type systems using extensions to first order predicate logic. Because PCC uses a general logic framework, PCC can encode complex security properties that cannot be expressed in TAL or FLINT type systems. On the other hand, the proofs generated by PCC are often longer than proofs in TAL or FLINT.

Foundational Proof-Carrying Code [2]: Foundational proof-carrying code (FPCC) is designed to minimize the size of the trusted computing base of systems that run low level code from untrusted sources. In FPCC, the operational semantics of low level code is defined in a logic that is suitably expressive to serve as a foundation of mathematics. FPCC uses higher order logic with a few axioms of arithmetic, from which it is possible to build up most of modern mathematics. FPCC is general enough to verify any property expressible in Church's higher order logic.

As we can see from the previous discussion, the current technology is not limited by the expressiveness of frameworks for representing proofs. Moreover, for programs written in high level languages like Java or ML, technology already exists to generate proofs that encode the safety properties expressed in the type systems of those languages. It is interesting to note that all the JVM extensions presented in Section 2 are memory-safe extensions. It is thus only a matter of engineering to produce safety proofs for programs written using these extensions.

The current technology is, however, limited by our ability to automatically generate proofs for some complex program properties. For example, both the Java and ML runtime systems include trusted code that performs garbage collection. If a garbage collector is to be shipped as part of the application code in the EVM framework, then the garbage collector must contain a proof of safety. Generating safety proofs for something like a garbage collector reduces to statically proving complex heap properties in programs with destructive updates. The next section explores state of the art in this area.

3.4 Static Checking of Heap Properties

This section explores some recent research in the area of statically proving heap properties.

Three-Valued-Logic Analyzer [29, 44]: The three-valued-logic analyzer (TVLA) deals with proving complex heap properties in languages with destructive updates. For certain programs with pre- and post-conditions, TVLA is able to to determine such properties as when the input to the program is a list (or tree), the output is also a list (or tree). TVLA has been used to analyze programs that manipulate doubly linked lists and circular lists, as well as some sorting programs.

Pointer Assertion Logic Engine [34]: The pointer assertion logic engine (PALE) is another framework for checking data structure invariants in programs that can contain pre- and post-conditions as well as loop invariants. PALE can verify a large class of data structures, namely all those that can be expressed as graph types [25]. Graph types consist of data structures that must be represented by a spanning tree backbone, with possibly additional pointers that do not add extra information. Graph types include data structures like doubly linked lists, trees with parent pointers, and threaded trees.

Role Analysis [27]: While TVLA and PALE are primarily intraprocedural, Role Analysis supports compositional interprocedural analysis and verifies similar properties. Roles capture the notion that the type of an object should depend not only on the fields and methods of the object, but also on the data structures in which the object participates. As objects move between data structures, their types change to reflect their changing relationship with other objects.

But the technology for static verification of heap properties is still in early stages. For example, we still cannot statically verify the correctness of a practical garbage collector, even though some recent advances offer partial solutions to the problem [35, 51].

3.5 Architecture of a Secure EVM

To summarize the discussion so far, this paper argues that the interface to a portable EVM should consist of a low level instruction set and an expressive logic framework for representing safety proofs. Programs to be run on EVMs must be accompanied by safety proofs. An EVM will statically verify the corresponding proof before executing a program. The EVM framework will provide application developers with tremendous flexibility to implement their ideas and innovations. But the EVM framework will also place a responsibility on code producers to produce safety proofs for the low level EVM code they generate.

For programs written in high level languages like Java or ML, technology already exists to generate proofs that encode the safety properties expressed in the type systems of those languages. But the technology for static verification of heap properties is still in early stages. An EVM must therefore also have the ability to run code signed by trusted authorities. For program components such as a practical garbage collector that cannot be proved safe with current technology, an EVM can install and run such a component if the component is signed by a trusted authority. As technology for static checking improves, there will be less need to use signatures for ensuring security. Figure 1 provides an illustration of the EVM interface.

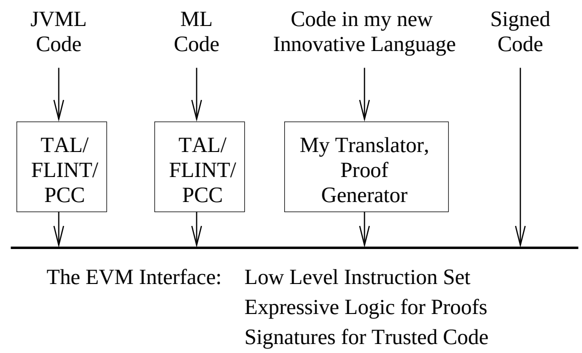

Figure 1: The EVM Interface

4 Other Portable Code Formats

This section reviews some prominent portable intermediate code formats other than JVML.

4.1 Microsoft Intermediate Language

The Microsoft .NET architecture is a new computing environment designed to support a variety of distributed applications. .NET software components are distributed in an intermediate language, Microsoft IL, executed by the Microsoft Common Language Runtime (CLR).

The design of IL is similar in spirit to the design of JVML. Like JVML, IL is a high level language and includes class-based objects, inheritance, garbage collection, and a security mechanism based on type safe execution. There is also on-going work to extend IL to support parametric polymorphism [22] and features like first class functions, closures, and thunks (to make it easier to support functional languages) [47].

But IL suffers the same problems like JVML. Being a high level language, IL is not very flexible. Every time an application writer comes up with an idea that is just slightly different from what is supported, there is no simple way to express the innovation in IL.

4.2 Limbo

Limbo [13] is a programming language designed at Lucent Technologies to be part of the Inferno [41] operating system. Inferno was created for supporting distributed services. In addition to traditional computing systems, Inferno was intended to be used in a variety of network environments, for example, those supporting advanced telephones, hand-held devices, television set-top boxes, and inexpensive network computers. The Inferno system was designed at about the same time as Java, and the project had similar goals as that of Java. However, unlike Java, it failed to become as popular.

Limbo itself is a source programming language. Limbo is type safe. It borrows its expression syntax and control flow from C, but it also includes declarations as in Pascal, abstract data types, typed channels, first class modules, automatic memory management, and preemptive threads. It excludes pointer arithmetic and casts.

Programs written in Limbo are compiled to a portable intermediate format. Unlike JVML, the Limbo intermediate format is low level and is designed to closely match with the modern processor architectures. But unlike JVML, the intermediate code in Inferno is not type checkable. Inferno relies on trusted compilers to sign the intermediate code.

5 Conclusions

The spread of the JVM platform offers the real possibility of realizing a *write once, run everywhere* environment. JVMs provide a portable and secure platform to run code. But unfortunately, JVMs are not sufficiently extensible. This paper presents many examples to illustrate this point.

This paper also presents the design of an extensible virtual machine (EVM) that can be used as a standard platform for portable code. The paper describes how the EVM platform can provide a flexible, efficient, and secure environment for running code.

References

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.

[2] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Principles of Programming Languages (POPL)*, January 2000.

[3] M. P. Atkinson, M. J. Jordan, L. Daynes, and S. Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system. In *Persistent Object Systems (POS)*, May 1996.

[4] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating Systems Principles (SOSP)*, December 1995.

[5] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Symposium on Operating Systems Principles (SOSP)*, December 1993.

[6] C. Boyapati. JPS: A distributed persistent Java system. SM thesis, Massachusetts Institute of Technology, September 1998.

[7] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[9] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *ACM Java Grande Conference*, June 1999.

[10] R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[11] S. Chiba. Load-time structural reflection in Java. In *ECOOP Symposium on Objects and Databases*, June 2000.

[12] M. A. Dmitriev and C. Hamilton. Towards scalable and reliable object evolution for the PJama persistent platform. In *ECOOP Symposium on Objects and Databases*, June 2000.

[13] S. Dorward, P. Winterbottom, and R. Pike. The Limbo programming language, 1997. Available at http://inferno.lucent.com/inferno/limbo.html.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-specific resource management. In *Symposium on Operating Systems Principles (SOSP)*, December 1995.

[15] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. In *Software—Practices and Experience 30(3)*, March 2000.

[16] The FLEX compiler infrastructure. Available at http://www.flex-compiler.lcs.mit.edu.

[17] GemStone/J. Available at http://www.gemstone.com/products/j/main.html.

[18] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Principles of Programming Languages (POPL)*, January 1996.

[19] M. Golm. Design and implementation of a meta architecture for Java. Master's thesis, University of Erlang, January 1997.

[20] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[21] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. CS Technical Report TR2001-1856, Cornell University, 2001.

[22] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[23] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of Metaobject Protocol*. The MIT Press, 1991.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP Symposium on Objects and Databases*, June 2001.

[25] N. Klarlund and M. I. Schwartzbach. Graph types. In *Principles of Programming Languages (POPL)*, January 1993.

[26] C. Krintz, B. Calder, and U. Holzle. Reducing transfer delay using Java class file splitting and prefetching. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1999.

[27] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Principles of Programming Languages (POPL)*, January 2002.

[28] C. League, Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *International Conference on Functional Programming (ICFP)*, September 1999.

[29] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, June 2000.

[30] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[31] B. Liskov, C.-H. Moh, S. Richman, L. Shrira, Y. Zhang, and C. Boyapati. Safe lazy software upgrades in object-oriented databases. Submitted for publication, March 2002.

[32] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*, January 1988.

[33] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quartermen. *The Design and Implementation of the 4.4 BSD UNIX Operating System*. Addison-Wesley, 1996.

[34] A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[35] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[36] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Principles of Programming Languages (POPL)*, January 1998.

[37] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.

[38] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*, January 1997.

[39] A. Oliva and L. E. Buzato. Design and implementation of Guarana. In *Conference on Object-Oriented Technologies and System (COOTS)*, May 1999.

[40] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1987.

[41] R. Pike, D. Presotto, S. Dorward, D. M. Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. *Bell Labs Technical Journal 2(1)*, Winter 1997.

[42] ObjectStore PSE for Java. Available at http://www.odi.com/products/psej.html.

[43] Java Remote Method Invocation. Available at http://java.sun.com/products/jdk/1.2/docs/guide-/rmi.

[44] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Principles of Programming Languages (POPL)*, January 1999.

[45] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Principles of Programming Languages (POPL)*, January 2002.

[46] T. B. Steel. A first version of UNCOL. In *Western Joint Computer Conference*, May 1961.

[47] D. Syme. ILX: Extending the .NET Common IL for functional language interoperability. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, September 2001.

[48] M. Tofte and J.-P. Talpin. Region-based memory management. In *Information and Computation 132(2)*, February 1997.

[49] M. Viroli and A. Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

[50] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *IEEE Symposium on Security and Privacy*, May 1998.

[51] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Principles of Programming Languages (POPL)*, January 2001.

[52] I. Welch and R. J. Stroud. Kava—Using bytecode rewriting to add behavioral reflection to Java. In *Conference on Object-Oriented Technologies and Systems (COOTS)*, June 2001.

[53] I. Welch and R. J. Stroud. Dalang—A reflective extension for Java. Technical Report CS-TR-672, University of Newcastle upon Tyne, September 1999.