The Dynamic Virtual Machine: A Platform for Dynamic Language Implementation

Gregory T. Sullivan

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

http://www.ai.mit.edu

The Problem: Software produced with current technology is typically opaque and brittle. Applications adapt poorly to changes in environment (OS, hardware, data profile) or intended use, scale poorly, and are difficult to debug, maintain, and enhance. As software gets larger and more complex, especially in the artificial intelligence research domains, software needs the ability to adapt to complex, dynamic environments.

Motivation: We are developing an infrastructure to support the addition of advanced language features to the toolset available to programmers. We are focusing on **reflection**, **dynamism**, and **metaprogramming**.

Reflection refers to the ability of an application to inspect, at runtime, its internal state, both data and control. Examples include examining the class hierarchy at runtime, or using runtime profile information to find performance bottlenecks. Based on analysis of data gathered using runtime reflection, an application may want to dynamically adapt its behavior or internal structure – this is what we refer to as *dynamism*. Reflection and dynamism enable a form of programming called *metaprogramming*: writing programs that reason about and modify the behavior of themselves and/or other programs.

Previous Work: The most successful attempt at adding a high degree of reflection and dynamism to the programming infrastructure is the metaobject protocol (MOP) of the Common Lisp Object System (CLOS). As far as more mainstream languages go, C++ has very little reflection or dynamism, but Java™ has significant reflective capabilities. Java's reflection API makes Fields, Methods, and Classes first class, though immutable, objects. Java also allows dynamic class loading, providing a somewhat heavyweight mechanism for limited runtime program adaptation. The programming language Dylan provides a superset of the reflection capabilities of Java, making some items, such as the set of methods for a virtual function, mutable.

The work on Aspect Oriented Programming [2] attempts to add more control abstractions to the programming process, and there is an implementation, called AspectJ, on top of Java.

The research area of dynamic optimization (see recent workshops on Feedback-Directed Optimization) is relevant to our research on efficient implementation of dynamic languages.

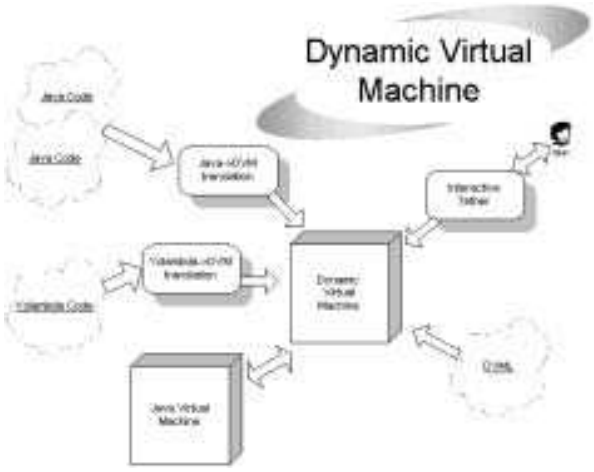
Approach: We have designed and implemented a *Dynamic Virtual Machine* (DVM) that is intended to be an ideal host for dynamic programming languages. The DVM provides all of the features of the Scheme programming language, such as first-class functions and dynamic typing, with the addition of first-class types, support for dynamic dispatch, and a high degree of reflection. The type system is based on *predicate dispatching* [1], which allows arbitrary predicates as a base case.

We have a translation from Java to the DVM, and we are working on translations from other languages. Once running on the DVM, a translated application can take advantage of reflection and dynamism unavailable in the source language (e.g. Java). For example, an application can be dynamically instrumented to track its own performance, and new versions of methods can arrive dynamically over the network. Some of the metaprogramming techniques supported by the DVM are related to Aspect-Oriented Programming.

Coincident with research centered on exposing ever more dynamic features to programmers is our research on *dynamic, optimistic optimization*. A basic tenet of our research is that programs should not pay a performance penalty for dynamic language features they do not use. For example, if a program never redefines the function +, the program should not run slower because it could have. Because very little is known about the behavior of an application before runtime, and because, at least in theory, everything might change while the application

is running, we need to do *optimistic* optimization at runtime. For example, suppose we have a call to + on two arguments that are guaranteed to be integers. We may inline that call to +, but we must instrument all paths that may redefine + to dynamically undo this optimization.

To produce optimized versions of functions while tracking dependencies, we have developed the technique of *dynamic partial evaluation*. Dynamic partial evaluation applies traditionally static partial evaluation techniques at runtime. Dynamic partial evaluation of an expression produces a triple of the result value, an optimized version of the expression, and any dependencies accrued during evaluation of that expression.



Impact: The need for software to reflect on its own structure and performance, as well as to dynamically adapt to changing circumstances, will only continue to increase with time. The static approach to analyzing and optimizing programs is less and less successful in the face of this increased need for dynamism and adaptability.

Also, there is increasing interest in "lightweight" scripting languages, as well as domain-specific languages. The implementations of these languages typically take the following trajectory:

1. Initial introduction, high enthusiasm, rapid adoption,
2. Application to ever larger projects, growing, broad user base,
3. Addition of more sophisticated language features, while trying to improve performance of legacy "quick and dirty" implementations,
4. Implementations hit performance and maintenance walls.

A very high-level target machine such as the DVM provides an attractive language implementation alternative, as it starts with sophisticated language features and the dynamism that is so difficult to add on later.

Future Work: During the next year, we will focus on dynamic optimization as well as interoperability with Java and other languages.

Research Support: This research is supported in part by DARPA under the Express contract F30602-97-2-0013.

References:

[1] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211. Springer, 1998.

[2] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.