Advanced Programming Language Technology for Reflective, Dynamic, Adaptive Software

Gregory T. Sullivan & Jonathan R. Bachrach

Artificial Intelligence Laboratory
Massachusetts Institue Of Technology
Cambridge, Massachusetts 02139

http://www.ai.mit.edu

The Problem: Software produced with current technology is typically opaque and brittle. That is, 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 the scale and complexity of applications attempted by the software development community as a whole and by the artificial intelligence research community in particular grow, software needs to be developed with the ability to adapt to a complex, dynamic environment.

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 introspect, at runtime, on 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.
- **Syntactic extension**: Based on ideas from the Lisp and Dylan programming languages, we are designing a powerful procedural macro facility for Java.

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) [1,2]. As far as more mainstream languages go, C++ has very little reflection or dynamism, but Java™ has added significant reflective capabilities to the language. Java's reflection API makes Fields, Methods, and Classes first class, though immutable, objects. Java also allows dynamic class loading, providing a heavyweight yet effective mechanism for runtime program adaptation. The programming language Dylan [4] provides a superset of the reflection capabilities of Java as well as a MOP that allows dynamic changes such as adding methods to generic functions, adding classes, etc. The MOP of Dylan is not as complete as that for CLOS, largely due to performance considerations. The work on Aspect Oriented Programming [3] attempts to add more control abstractions to the programming process, and there is an implementation, called AspectJ, on top of Java.

C and C++ include a limited syntactic extension facility with simple string substitution macros. Lisp and Scheme have much more sophisticated macro systems, including the notion of "hygiene" which addresses complications related to variable name clashes. Lisp and Scheme are amenable to sophisticated programmer-defined syntax manipulation because the prefix language syntax itself is so simple. Dylan is the most ambitious attempt to date to add modern syntactic extension features to an infix language.

Approach: Providing syntactic abstraction, reflection, and extreme dynamism to the programming substrate motivates two major areas of research: *language design* and *language implementation*. That is, we need to make these advanced programming features usable and practical. From a language design perspective we are trying two approaches:

- Integration of new language features into Java. The macro system for Java is designed to "feel like Java".
- Aspect-Oriented Programming Languages. The work in aspect-oriented programming motivates work in designing domain-specific "mini-metalanguages". Aspects would be written as metaprograms abstracting over the behavior of existing programs.

The challenge for language implementation is to provide powerful abstraction facilities but yet to avoid any performance degradation when these facilities are not used. Furthermore, when reflection and dynamism are used, we want the performance hit to be minimal.

We are applying many innovative techniques to accomplish optimization in the face of reflection and dynamism, including:

- *Optimistic optimization*: With a meta-object protocol, many aspects of the running program can be changed dynamically, but these changes occur rarely in practice. We take advantage of this pattern by assuming the current state, but guarding any optimizations we make under these assumptions so that we may back out the optimization if our assumptions are violated.
- *Partial evaluation*: Partial evaluation is a general technique for optimization with respect to constant values. We use partial evaluation techniques to implement optimistic optimization.
- *Proof-based Optimization*: There are many program transformations that rely on more sophisticated facts about a program than can typically be obtained by mainstream compilers. We are developing an infrastructure for "pluggable optimizers" that prove their transformations correct in a high-level proof language.
- *Dynamic Virtual Machine*: The instruction set of existing virtual machines (or real machines) is typically either too low level or too language-specific to be a reasonable target for compiling highly reflective and dynamic language features. We are developing a very high level virtual machine with direct support for advanced language features such as first class functions, first class predicate types, generic functions and multimethod dispatch, and runtime method redefinition.

The following diagram presents an overview of a running DVM-based system:

[height=2in, width=5in]figures/dvm-diagram.eps

Optimizations are performed safely and opportunistically on the (potentially) running application.

The combination of syntactic extension, reflection, and dynamism provides a powerful substrate for "aspect-oriented programming" as described in [3].

Difficulty: There is a fundamental tension between abstraction and performance. The task of language implementation is to "compile away the abstraction". We aim to drastically increase the abstraction facilities available to the programmer, and we thereby set ourselves a difficult implementation task.

Impact: Abstraction, reflection, and dynamism are powerful tools, and by putting them in the hands of programmers, we hope to increase the adaptability and dynamism of future software.

Future Work: We are actively pursuing all of the subprojects outlined above concurrently.

Research Support: This research is funded in part by DARPA under contract number F30602-97-2-0013.

References:

[1] D. Bobrow and L. DeMichiel and R. Gabriel and S. Keene and G. Kiczales and D. Moon. Common Lisp Object System Specification. *ACM SIGPLAN Notices*, vol. 23, Sep 1988.

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

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, JM. Loingtier, J. Irwin. Aspect-Oriented Programming . In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[4] A. Shalit. The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language. *Addison-Wesley*, 1997.