Research Statement

Kathryn E. Gray

My research involves the design and implementation of programming languages to solve problems in programming. Instead of designing new languages, I modify the design of existing languages to improve particular aspects of programming using that language. In implementing these designs, it is sometimes necessary to devise new techniques in compilation or interpretation.

Additionally, I work to improve introductory computer science education through targeted language design and implementation (as well as environment interaction). This process involves observing students using a language (or tool), designing an improved system based on these observations, and then observing new interactions and evolving the design as necessary.

To date, I have completed two related research projects that modify Java to improve introductory education and multi-language interoperability. Both projects are implemented in my Java to Scheme compiler, ProfessorJ, and are a part of the DrScheme development environment.

Modifying Java for Education

The Java programming language and its compilers were designed for professionals. From the perspective of novice programmers, the language contains too much flexibility, and compilers report mistakes in unfamiliar jargon. Novices become frustrated when the must learn extra syntax and jargon up front. The problem lies in using professional tools too soon. To avoid this problem, I developed subsets of Java designed for novice programmers. These subsets are supported by my compiler, ProfessorJ. This approach follows the same philosophy as DrScheme’s pedagogic subsets of Scheme.

The ProfessorJ compiler presents three languages to students, each a limited subset of the full Java programming language. The Beginner language presents objects and methods, with a limited number of statements; the Intermediate language adds mutation, object polymorphism, and inheritance; the Advanced language adds loops, arrays, and library access.

Due to the restricted languages, error messages can avoid referring to concepts that the students do not know, and the environment can direct students away from common mistakes that produce valid programs in professional Java. For example, introductory students often forget the else keyword of an if statement without forgetting the else clause. This omission causes incorrect execution, where few students consider that the mistake is syntactic – most believe that a compiling program is syntactically correct. In the Beginner and Intermediate languages, such an omission is illegal. When the students move to the Advanced language, the students consider the benefits of an if with no else clause and also consider the negative effects of accidentally omitting else. If they encounter the error in the future, they do consider checking the syntax of the program.

I developed the initial subset designs based on observations of students using full Java, and using knowledge of the desired curriculum. Over time, the content of the Java subsets has changed as a result of ongoing observations.

Modifying Java for Interoperability

Traditional interoperability connects multiple languages through an API. Using this approach, accessing a function written in another language appears in the program as a call into the API with a string argument specifying the function, obscuring the actual function call; data moving across the language boundary must be converted and checked, often manually. This creates a high overhead for using multiple languages in
one application. My interoperability work specifically focuses on Java and Scheme, but provides a general foundation for interoperability between other languages.

Instead of presenting interoperability through an API, my work extends Java with a new primitive type, \texttt{dynamic} and the ability to import libraries directly from Scheme. Values provided by the Scheme library use the same syntax as Java values. A value with the \texttt{dynamic} type can be used as a value with any type. Any requirements on the value necessary to preserve type safety are checked during program execution. All values imported from a Scheme library automatically have \texttt{dynamic} type.

Runtime checks of values with type \texttt{dynamic} are determined by the uses of the value within the program. For example in the following program:

```java
dynamic progressOk;
if (progressOk)
    ...
else
    progressOk.error();
```

the value of progressOk must be a boolean during the execution of the \texttt{if} test, and must be an object with an \texttt{error} method during the execution of the else branch. The appropriate checks are automatically inserted during compilation, and a runtime error will occur if the value does not meet the requirement.

In addition to the immediate checks depicted above, object values passed to Scheme functions must be permanently guarded to prevent misuse (i.e. that a method expecting integers is not called with boolean values). These guards store information regarding the source location where the Java value is passed to a Scheme function. If the value is misused within the Scheme program, the resulting error message will report the location. This information assists the programmer in tracking the nature of the bug.

Supporting easy interoperability between Java and Scheme benefits my educational projects. Teachers, wishing to enhance their course with pedagogic libraries, will need access to Scheme and Java libraries and have limited time and programming resources. Additionally, my interoperability technique is applicable to the several Scheme systems that contain connections to Java as well as to connections between other languages (with additional language specific work).

Overall, this system allows programmers to write and debug programs that combine Java and Scheme with greater ease than was possible before.

**Future research directions**

In future work, I intend to continue investigating language improvements targeted at introductory students, as well as to evaluate the benefits of these improvements on education. Additionally, I intend to pursue a specification language allowing others to create their own pedagogic Java subsets. At present, there are two sets of pedagogic languages for Java, mine and those provided by the DrJava pedagogic environment. Not all users of these systems are satisfied with the restrictions provided. I believe it is possible to extend my compiler so that a declarative language specification could guide a program in connecting the pieces of the compiler into a compiler for the specified language.