Research Statement

Nathaniel Nystrom

My research focuses on developing programming language techniques for building safe, correct, and efficient systems. I enjoy solving real-world problems and building systems by working out the theoretical foundations of the problem and developing practical solutions.

Type systems allow programmers to modularly express invariants to be enforced by the compiler. In recent years, type systems have moved beyond enforcing simple safety properties to enforce more and more correctness properties of programs. I have a keen interest in applying developing and applying features of these new expressive type systems to issues that arise when constructing large applications, such as extensibility, security, concurrency, and distribution. I will address each of these areas in turn.

Extensibility

Software is frequently constructed by extending or composing existing code. The aim of my thesis research is to develop type-safe programming languages to more easily reuse code. Extending a large system written in a conventional language, can require either duplicating code—drastically complicating code maintenance—or writing boilerplate or glue code far larger than the code necessary to implement the new functionality. Programming languages should support scalable extensibility: the amount of code required to extend a body of code with a new feature should be proportional to the size of the feature.

Many conventional programming languages support scalable extensibility to some extent, but sacrifice type safety to do so. Run-time type checks are needed to allow extended code and base code to interoperate; thus, the program may fail unexpectedly due to a run-time type error. Since the type systems of these languages are too weak, the onus is on the programmer to ensure these type errors do not occur. In addition, substantial planning is required when developing extensible software to provide the appropriate hooks for extension. If a particular aspect of the software does not have the appropriate hooks, it cannot be extended scalably.

Our work on scalable extensibility grew out of the Polyglot compiler framework [CC 2003], which is written in Java and provides scalable extensibility through a design pattern approach I developed. Polyglot provides a base compiler that parses and type-checks Java source code, and then outputs equivalent Java code. Compiler developers can extend the base compiler to compile extended versions of Java into Java code, or to add new analyses or optimizations to the base Java compiler. Polyglot has been used in a number of research projects both at Cornell, and elsewhere. However, as implied above, Polyglot extension code often requires run-time type casts to interact with code inherited from the base compiler, leading to possible run-time type errors. Moreover, while Polyglot supports scalably extending the base Java compiler with new abstract syntax and new compiler passes, extending the type system often cannot be done scalably because the right hooks are not provided.

To address these limitations, we developed the language Jx [OOPSLA 2004], which provides type-safe scalable extensibility using nested inheritance and is largely backward compatibility with Java. Nested inheritance allows a collection of classes or packages to be extended by inheritance. Members of the extended class or package, including nested classes can be overridden to provide new functionality. Using a dependent type system that is nearly transparent, inherited code can use the new overridden classes safely without requiring the programmer to insert run-time type checks. Nested inheritance thus provides both scalable extensibility and type safety. Polyglot itself has been ported to Jx and we find writing Polyglot extensions in Jx requires less code and has stronger type safety guarantees than extensions written for the original Java version of Polyglot.

We have further extended Jx to allow software composition via multiple nested inheritance [Nystrom et al. 2006]. Composing two or more packages (or classes) recursively composes the members of those packages as well. We have used multiple nested inheritance to construct two composable, extensible
frameworks: a compiler framework based on Polyglot, and a peer-to-peer networking system based on FreePastry. Both frameworks support composition of extensions. For example, two compilers adding different, domain-specific features to Java can be composed to obtain a compiler for a language that supports both sets of features.

Building on this work, I plan to expand on both the theoretical and practical aspects of nested inheritance. For example, we have used multiple nested inheritance to compose compilers for multiple languages; however, a theory needs to be developed to define the semantics of the language implemented by the composed compiler. In particular, when are the feature sets of two languages orthogonal? Or conversely, when do the features of two languages interact in conflicting ways, preventing a meaningful composition.

Integrating this theory into Polyglot would enable creation of a compiler from several tiny compilers, each implementing a single language feature, thus implementing a domain-specific language specifically tailored, at a fine-grained level, for a particular application. Compilers constructed in this way would be ideal for exploration of the interaction of different language features.

I would also like to further explore the usability of nested inheritance by implementing more and larger applications in Jx and by extending the language to provide even more flexible mechanisms for code reuse. One application that particularly interests me is to use nested inheritance to extend a Java virtual machine and bytecode verifier. This would enable execution of extended versions of Java without having to translate first to Java and would enable stronger run-time guarantees. With both an extensible VM and an extensible compiler, the effort to implement new language features can be greatly reduced, enabling these new features to be more rapidly adopted into mainstream languages.

Security

Another area where I have done work is language-based security, which uses type theory to enforce security policies. My colleagues and I developed secure program partitioning [SOSP 2001, TOCS]. The goal of secure program partitioning is to protect the confidentiality and integrity of data owned by principals who do not completely trust each other or the machines on which computation is being performed. Programs are annotated with security policies. The compiler then partitions the code and data of the program so that untrusted hosts cannot access confidential data or subvert the control flow of the program.

I would like to explore applying techniques from secure program partitioning to web application development, allowing creation of web applications using a single language for both the server and the client (browser). The benefits of this approach are two-fold. First, developers would have greater assurance that the applications they create enforce their desired security policies. Second, by eliminating the impedance mismatch between server code—written today using languages such as PHP or Java and with a multitude of frameworks—and client code—written in JavaScript and HTML—web applications would be much easier to write and maintain.

Concurrency and distribution

In the future, I would like to explore applying recent developments in type theory such as dependent types, ownership types, and alias types to concurrent programming. Concurrent and distributed programs are becoming increasingly prevalent; however, the abstractions for concurrent programming are often either too weak or too strong for a given application. Weak abstractions do not adequately enforce correctness of these programs; strong abstractions such as transactions can limit the performance of concurrent programs by enforcing guarantees more restrictive than necessary for correctness. I believe type systems can be used to express the correctness requirements of concurrent programs in an application-specific manner, enabling high performance, but static enforcement of correctness.

Reuse sequential code in a concurrent setting is another area where type systems can be applied. At Sun Labs, I collaborated with Grzegorz Czajkowski and Laurent Daynès on ShMVM [ECOOP 2002], a Java virtual machine that shares state between multiple JVM processes, greatly reducing the overhead of running multiple Java applications on the same computer. However, ShMVM was difficult to implement correctly because the VM’s data structures were not designed to be shared by concurrent processes. The ability to specify application-specific data consistency requirements would have enabled faster implementation of the shared data structures with more confidence in their correctness.
To hide the effects of network latency, distributed systems are often built with optimistic protocols or with weak underlying data consistency models. Programming languages today do not provide adequate support for managing tentative or possibly inconsistent data. I would like to design language constructs for reconciling inconsistent replicas of data. When that data is made consistent. A type system similar to one used for enforcing information-flow policies, as in our secure programming partitioning work described above, would provide a means to tracking dependencies on inconsistent data.

Conclusions

Programming languages can address many of the challenges of building large real-world applications. I look forward to applying recent advances in type theory and programming language design to these applications, particularly in the areas outlined above.

References


[Nystrom et al. 2006] “Software Composition with Multiple Nested Inheritance”, Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. Submitted for publication.

