

Research Statement

Nathaniel Nystrom

<http://www.nanocow.com/nystrom>

My research focuses on developing programming languages for implementing safe, correct, and efficient systems. Fundamental problems that arise when developing applications for the Internet and for modern architectures include extensibility, security, concurrency, and distribution. My interest is in finding practical solutions to these problems by working out their theoretical foundations and then designing and implementing the programming language features based on these foundations.

My primary research focus has been on extensible languages and tools that enable programmers to express domain-specific knowledge directly in the language through the type system or via domain-specific abstractions. This simplifies software development by reducing the semantic gap between the domain of the application and the implementation language. Compilers and language runtime systems can then take advantage of this domain-specific knowledge to better analyze and optimize code. I believe compiler and runtime support for domain-specific abstractions is the key to developing correct, efficient software on modern architectures.

As parallelism increasingly becomes the standard way to achieve high performance in modern architectures, programmers will be required to develop concurrent software capable of running on hundreds, thousands, or even millions of processors. Developing correct and efficient code for these systems is challenging; concurrency-related errors and performance problems are notoriously difficult to test for and to debug. Programming languages, and type systems in particular, can address these problems by allowing programmers to express program invariants, enabling compilers to rule out errors in programs before they are run and to generate more efficient code.

The aim of my thesis research at Cornell was to develop type-safe programming languages to more easily reuse code [4]. This work was applied in the Polyglot compiler framework [6] to support construction of domain-specific extensions to the Java programming language.

More recently at IBM, I have worked on X10, an object-oriented programming language for high-performance computing that provides abstractions for lightweight concurrency and distribution [2, 10]. Most of my work on X10 has been on extending the language to allow programmers to support domain-specific annotations [8] and to easily express constraints on data [9]. These constraints and annotations are enforced by the compiler or by programmer-supplied compiler extensions to rule out errors and are used to generate efficient code.

Scalable extensibility and Polyglot

Polyglot [6] is an extensible compiler framework for implementing extensions of the Java language. Extensions are implemented by inheriting from a base Java compiler provided by the framework. Polyglot has been used in a more than twenty research projects at Cornell, IBM Research, and elsewhere¹, and I still actively maintain and improve Polyglot.

Polyglot provides *scalable extensibility*: the amount of code required to extend a body of code (in this case, the compiler) with a new feature should be proportional to the size of the feature. Without scalable extensibility, implementing new features can require duplicating code—complicating code maintenance—or writing boilerplate code far larger than the code necessary to support the new feature.

¹See <http://www.cs.cornell.edu/Projects/polyglot> for a partial list.

Nested inheritance and nested intersection

One limitation of Polyglot is that extension code often requires run-time type casts to interact with code inherited from the base compiler, leading to possible run-time type errors. Java's type system is not expressive enough to provide scalable extensibility without sacrificing static type safety. Moreover, the design pattern approach requires that the developer anticipate which features are to be extended, limiting extensibility.

To address these limitations, we developed the language J& ("Jet") [5, 7], which provides type-safe scalable extensibility using *nested inheritance* and is largely backward compatible 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 usually transparent to the programmer, 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 J&; writing Polyglot extensions in J& requires less code and has stronger type safety guarantees than extensions written for the original Java version of Polyglot.

J& also supports software composition via *nested intersection* [7]. Composing two or more packages (or classes) recursively composes the members of those packages as well. Using the J& version of Polyglot, compiler extensions can be easily composed: two compilers adding different, domain-specific features to the base language, Java, can be composed to obtain a compiler for a language that supports both sets of features. I have also applied nested intersection to the composition of peer-to-peer networking systems [4].

Annotations and compiler plugins in X10

After moving to IBM in Fall 2006, I continued my work on extensibility in the context of the X10 programming language [10, 2]. Users of X10 require language features that allow programmers to augment the base language with additional semantic checking or with application-specific or architecture-specific optimizations.

I designed an annotation and compiler plugin system for X10 [8]. The X10 compiler can be extended with compiler plugins that can perform new analyses and optimizations. Annotations provide a way to extend or modify the semantics of X10 without redefining the language itself and without breaking existing implementations. Programmers use annotations to provide additional information to the plugins. For example, plugins can be used to extend the language to support type annotations such as for units of dimension (e.g., to prevent confusion of English and metric units).

Dependent types in X10

Because concurrent code is especially difficult to debug and tune, a design goal of X10 was to rule out a large number of errors at compile time, and to have not only good performance, but also predictable performance. To address these goals, Vijay Saraswat and I designed and implemented a dependent type system for X10. *Dependent types*—types dependent on values—allow programmers to express invariants on the program's data that can be used by the compiler to rule out run-time errors and to generate efficient code.

In X10, dependent types take the form of constraint annotations on object-oriented types [9]. For instance, `Array[int]{rank==n}` is the type of all n -dimensional `int` arrays. This type depends on the variable n . Constraints in X10 are predicates on the program state that can be evaluated symbolically by the compiler.

Using these *constrained types*, the X10 compiler can statically rule out errors such as array bounds errors and many application-specific invariant violations. Constrained types are also used in code generation, allowing the compiler to generate efficient array access patterns based on the shape of the array, taking advantage of data locality to reduce overhead.

However, constrained types have some limitations that I hope to address in future work. First, constraint annotations can be verbose or tedious to write. Supporting type inference would allow these annotations to be elided by the programmer, enhancing the expressiveness of the language. Second, to

ensure type-checking is sound and decidable, the language in which constraints can be written is a restricted subset of the expression language of X10. Extending the constraint language, while preserving soundness, would enable more program invariants to be checked statically, ruling out larger classes of errors. Such a type system might resemble a design-by-contract approach combined with type states: methods are annotated with pre- and post-conditions and a summary of the side-effects of the methods. The system should allow the programmer to extend the constraint language with new predicates and to also provide code to be evaluated at compile time to check the constraints.

X10VM

The current X10 compiler translates X10 code to Java and runs on a normal Java virtual machine (VM). However, if X10 is to scale to thousands of processors, a dedicated X10 VM becomes essential. In work just now ramping up, the X10 group at IBM is collaborating with Steve Blackburn at Australia National University on the design and implementation of an X10 virtual machine with better support for X10's concurrency and distribution models. The prototype implementation of the VM will be based on Jikes RVM [1], a Java virtual machine. My intended contribution to the project is to implement an extensibility framework for Jikes RVM by leveraging ideas from my thesis work, and to design extensions of Java bytecode to support X10-specific instructions and types. Extensibility is especially important during the prototyping phase of the project, as the code rapidly evolves.

Thorn

Thorn is a concurrent dynamically-typed object-oriented scripting language I am developing with Jan Vitek and others at Purdue University and with John Field and Bard Bloom at IBM. The kernel of the language is a scripting language similar to JavaScript, Python, Ruby, or Perl. Scripting languages are highly dynamic and adaptable, but often do not perform as well as statically typed, compiled languages. In addition, these languages were not designed with support for concurrency. Thorn aims to have the dynamism of scripting languages, but to support high performance and concurrency, distribution, and security through a component model.

A novel feature of Thorn is support for user-supplied language extensions, static analyses, and code transformations, which facilitate the construction of domain-specific languages. The Thorn language and runtime system are extensible through plugins, written in Thorn itself. Using plugins, software can be rapidly prototyped, and then later, incrementally annotated with types and other specifications that can be checked statically. Using these annotations, the runtime system can more effectively optimize code and can check for concurrency-related errors or security violations. We are exploring the use of constraint-based annotations in Thorn similar to the X10 type system.

One problem with supporting annotations using compiler plugins is that modularity can be lost. For certain annotations (e.g., security annotations like in Jif [3]), if one library is annotated, then clients of the library, and any code that transitively accesses the library—including, unfortunately, third-party code outside the programmer's control—must then also have these annotations. Designing a module system that allows modules compiled using different plugins to interoperate is an essential requirement. This requires developing a theory for specifying what annotations can or cannot be composed and what annotations can or cannot be enforced when composing modules. This theory would extend my earlier work on nested intersection.

Language-based tools like IDEs, debuggers, and performance analysis are another essential requirement for effective programming. However, tool support for domain-specific languages is often inadequate simply because the market for these languages is usually small: tools require a considerable investment to develop. Extending the Thorn infrastructure to support both an extensible compiler and runtime system and an extensible tool chain would lower the development cost for these tools.

Conclusion

The challenges of concurrency and distribution in today's computing environments make it essential for programming languages to support extensibility. These issues will only grow more important as parallelism becomes the way to achieve high performance. By providing a mechanism for programmers to

statically analyze and optimize code using application-specific knowledge, compiler and runtime support for domain-specific abstractions is the key to developing correct and efficient software for modern architectures.

I intend to focus on the research areas outlined above, specifically:

- Investigating type systems based on constraints that enable programmers to specify program invariants in a natural way; extending this work to capture user-defined constraints and constraints on mutable data.
- Exploring runtime system extensibility in the context of a virtual machine for X10.
- Designing an extensible scripting language and runtime system with support for user-supplied languages extensions via compiler and runtime plugins.

References

- [1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal*, 44(2), May 2005.
- [2] Philippe Charles, Christian Grothoff, Christopher Donawa, Kemal Ebcioglu, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 2005 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2005.
- [3] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release, at <http://www.cs.cornell.edu/jif>, July 2001–2009.
- [4] Nathaniel Nystrom. *Programming Languages for Scalable Software Extension and Composition*. PhD thesis, Cornell University, Ithaca, New York, USA, January 2007.
- [5] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 99–115, October 2004.
- [6] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *12th International Conference on Compiler Construction (CC 2003)*, number 2622 in Lecture Notes in Computer Science, pages 128–152, Warsaw, Poland, April 2003. Springer-Verlag.
- [7] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software extension. In *Proceedings of the 2006 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 21–36, Portland, Oregon, October 2006.
- [8] Nathaniel Nystrom and Vijay Saraswat. An annotation and compiler plugin system for X10. Technical Report RC24198, IBM T.J. Watson Research Center, 2007.
- [9] Nathaniel Nystrom Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 2008 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2008.
- [10] Vijay Saraswat, Nathaniel Nystrom, et al. The X10 language specification, version 1.7. <http://x10.sf.net/docs/x10-170.pdf>, September 2008.