# Research Statement

Shan Shan Huang

ssh@cc.gatech.edu

Modern software applications are disproportionately complex relative to the problems they solve. The essence of a solution is often entangled with large amounts of code dealing with *accidental complexities*: interfacing with external components, multiply defining functionality due to inefficient reuse, appeasing the type system with extraneous structures and annotations, etc. While devising software solutions to an essentially complex problem is intellectually challenging, dealing with accidental complexities is tedious, and often programmed by rote.

The driving force behind my research is *to reduce the accidental complexities in software engineering; to liberate programmers from being "code monkeys" so they can fully engage in the creative and intellectual process of problem solving.*

The key in reducing complexity lies in developing the right abstractions. My research applies programming languages techniques to raise the abstraction level used in program construction: I design language abstraction mechanisms that support better modularity and reuse, and I enable programmers to develop their own domain-specific abstractions through program generation.[1]

***Structural abstraction.*** The complexities in software engineering can be significantly reduced through modular program construction. I developed structural abstraction to support the separate development of orthogonal functionalities, and the construction of software through their composition. Unlike previous approaches such as Aspect-Oriented Programming, my techniques, morphing [4, 6] and static type conditions [5], are the only ones to reach this level of abstraction while maintaining *modular type-safety*: a structurally-abstract piece of code is separately type-checked to guarantee that it is well-typed regardless of its uses.

***High-level abstractions for programming reconfigurable hardware.*** I developed Lime [1], an extension of Java that lifts the abstraction level of hardware programming from gates, wires, and registers, to high-level Object-Orientation, with familiar features such as data encapsulation, subtyping, dynamic dispatch, and automatic memory management. With Lime, the average software engineer can obtain considerable performance gains from programming hardware, while retaining the safety and abstractions afforded by high-level languages.

***Program generation.*** A general purpose language cannot tailor its abstraction level to one suitable for every task. Thus, I developed Meta-AspectJ (MAJ) [7, 8], a program generation tool that allows programmers to develop their own *domain-specific* abstractions. MAJ is an extension of Java that allows the generation of AspectJ (and, by extension, Java) using templates. Using MAJ, domain-specific abstractions can be developed as program generators, with AspectJ as a backend to handle the low-level complexities of program transformation [2]. **This work resulted in a Best Paper Award at GPCE 2004 [8].**

My future research plans include, but are certainly not limited to, 1) investigating flexible program extension mechanisms, 2) developing techniques to ease the complexities induced by type systems while retaining their benefits, 3) developing cross-language semantics so the right language (and thus the right abstractions) can be used to program the right task, yet tasks can meaningfully compose, and 4) evaluating the impact of programming languages' features on software engineering.

---

[1] My research has been conducted in collaboration with my co-authors. I use the first-person herein for brevity.

# Structural Abstraction

Modular program construction—constructing programs by composing separately developed modules of orthogonal functionalities—has been a long-standing goal in software engineering. Previous techniques, such as Aspect-Oriented or Feature-Oriented Programming, allow only the separate implementation of orthogonal functionalities, but not the separate *reasoning* of the implementations. For instance, there is no notion of type-checking an aspect separately from the code it may be composed with. I developed *structural abstraction* as an alternative approach to modular program construction. My techniques, morphing and static type conditions, allow both the separate development *and the separate reasoning* of orthogonal functionalities.

**Morphing.**  Morphing allows code to be declared by statically reflecting over and pattern-matching on the methods or fields of other types—including unknown type parameters [4, 6]. For instance, using MorphJ, the Java extension I developed to support morphing, we can declare a generic class `SynchronizeMe<X>` such that, *for every method of X*, `SynchronizeMe<X>` declares a method with the same signature, but inserts synchronization code in each method. Notice that `SynchronizeMe<X>` is entirely generic to the structure of its input type `X`, and can be instantiated with any type to create a "synchronized" version of that type. This is to say that `SynchronizeMe<X>` is a separate and modular capture of the functionality "synchronization".

The distinguishing contribution of MorphJ over previous approaches of similar expressiveness, is its guarantee of modular type-safety: a MorphJ generic class is type-checked separately from its uses to guarantee that it is well-typed for all possible type-instantiations.

The practical software engineering impact of morphing is demonstrated through the reimplementation of the Java Collections Framework (JCF), the standard data structures library for Java, and DSTM2, a software transactional memory framework. Compared to the originals, the MorphJ versions contain up to *6 times fewer lines of code*, and removed the complexities of traditional specialization techniques (e.g., the duplication and runtime specialization of code).

**Static Type Conditions.**  Static type conditions allow the expression of case-specific code, where case conditions are encoded as subtyping constraints. For instance, in cJ (conditional-Java) [5], the language I developed to support static type conditions in Java, one can declare class `List<X>` to implement interface `Comparable` and method `compareTo(X)` *only if* `X` is a subtype of `Comparable`.

Unlike conditional compilation techniques (e.g., "`#ifdef`" of C/C++), cJ is separately type-checked, and guarantees the consistent uses and definitions of conditionally declared code.

The software engineering benefits of static type conditions are illustrated through a reimplementation of JCF in cJ, which solves the well-known problems with JCF "optional" methods [3].

# High-level Abstractions for Programming Hardware

Software construction has long left the dark ages of low-level assembly programming. Reconfigurable hardware (e.g., FPGAs), however, is still being programmed with languages whose abstractions are at the level of wires, gates, and registers. The computing power of FPGAs has thus been unavailable to all but the relatively few with training in integrated circuit design.

The Liquid Metal project at IBM Research aims to make reconfigurable hardware available for software programmers, through a single high-level language, Lime [1]. Lime is an extension of Java, offering all of Java's high-level Object-Oriented abstractions. At the same time, Lime extends Java with features that expose Lime programs to bit-level analysis and parallelism. These are essential for synthesis down to FPGAs in a space-efficient and performant manner—though equally beneficial for optimization in the JVM.

I implemented the DES encryption algorithm in Lime, and executed it in an environment comprising a JVM running on PowerPC 405, and Xilinx FPGAs. The results showed that synthesis from a high-level language is possible, and yields reasonable performance improvements.

# Program Generation

Powerful program generation enables programmers to develop domain-specific abstractions that are highly beneficial for specific tasks, though not suitable for inclusion in a general purpose language. Meta-AspectJ (MAJ) [7] is a program generation tool that allows the generation of AspectJ (and, by extension, Java) using code templates. MAJ advocates the approach that, in combination with program generation, AspectJ can be used as an "assembly" language for developing domain-specific abstractions [2]. A mature language implementation is easily achieved since AspectJ handles the low-level issues of interfacing with the base Java language.

MAJ is a structured meta-programming tool—it guarantees that a well-typed meta-program always generates syntactically correct programs. MAJ is also a mature meta-programming tool: it minimizes the number of meta-programming (quote/unquote) operators and uses type inference to reduce the need to remember type names for syntactic entities.

# Future Research Directions

**On program extension mechanisms.** Every programming language imposes an organizational structure on its programs. While these structures help programmers organize code; they can also be restrictive in how code can be extended. I am interested in investigating how to reduce the restrictions imposed by these static structures.

In the immediate future, I want to experiment with two specific ideas. First, I plan to investigate the power of *principled reflection*. Reflection is an important technique for introspecting and abstracting over a program's rigid structures. Reflection as it is implemented in mainstream languages (e.g., Java, C#), however, confuses meta-level entities, e.g., types and names, with program-level values, e.g., strings. This leads to undisciplined use of reflection, and paradoxically, limits its power at the same time. In principled reflection, types and names stay at the meta-level, empowering programmers and the compiler to use and reason about them as such.

Secondly, I plan to investigate alternative methods for code sharing. Code sharing through inheritance, as is the case in mainstream Object-Oriented languages, imposes an extraneous layer of static structure on programs, confusing the concept of modeling with reuse. One possible alternative is a clone-based approach, where code is *conceptually* "cloned" before being extended.

**On complexities induced by type systems.** Typing annotations are an extra layer of computation that programmers have to implement for the benefit of catching some errors at compile time. Types are not essential to the problems being solved, thus the accidental complexities they induce must be proportional to the benefits they provide. I am interested in developing techniques to remove overly complex type system features while retaining their benefits.

In the immediate future, I plan to attack the complexities introduced by variance in Object-Oriented generics. Use-site variance (e.g., Java wildcards) and definition-site variance (e.g., Scala-style variance) each impose different kinds of complexities on programs. This complexity can be removed by combining use-site and definition-site variance through a modular type-inference. Furthermore, inference can allow even more flexible variant types that are currently not expressible in either Java or Scala.

**On cross-language interactions.** The right abstractions, and thus the right languages, should be used to program the right tasks. The age of building applications using one monolithic general purpose language is coming to an end. We already see the beginning of this trend in industry, where an increasing number of applications are built using a combination of heavyweight languages like Java or C#, along with scripting languages to add dynamic behavior, XML to specify system configuration, SQL to query databases, etc.

I am interested in investigating the interactions at the boundaries of these languages. Is there a general framework that can describe how the semantics of one language carries over to another? How can the safety properties that are guaranteed and expected by a language be preserved across language boundaries? What are the useful abstractions for the "glue" languages that connect components written in different languages? These are important questions that must be answered in order for components written in different languages to compose in a meaningful way, and in order for their composition process to not become another accidental complexity.

**On the impact of language features on software engineering.** Throughout my Ph.D. career, I have grown increasingly interested in how programmers actually use language features. Many academic debates on what features are "useful" are not substantiated by evidence from practitioners. I am interested in unearthing some hard data that can guide the research in programming languages and software engineering. Open source repositories such as SourceForge and GoogleCode provide us with a multitude of data. However, software engineering being such a social process, the real challenge is to find a way to remove human bias from this data. I am interested in collaborating with researchers in software mining, and in drawing expertise from areas that have developed mature methods in removing human bias, such as epidemiology.

# References

[1] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric M. Rabbah. Liquid Metal: Object-oriented programming across the hardware/software boundary. In Jan Vitek, editor, *ECOOP '08: Proceedings of the 22nd European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 76–103. Springer, 2008.

[2] Shan Shan Huang and Yannis Smaragdakis. Easy language extension with Meta-AspectJ. In *ICSE '06: Proceedings of International Conference on Software Engineering*, pages 865–868, New York, NY, USA, May 2006. ACM.

[3] Shan Shan Huang and Yannis Smaragdakis. Building scalable libraries with cJ. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 45–46, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. In *PLDI '08: SIGPLAN Conference on Programming Language Design and Implementation*, volume 43, pages 79–89, New York, NY, 2008. ACM.

[5] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cJ: Enhancing Java with safe type conditions. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, New York, NY, 2007.

[6] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *ECOOP '07: Proceedings of the European Conference on Object-Oriented Programming*, pages 399–424. Springer-Verlag, August 2007.

[7] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Domain-specific languages and program generation with Meta-AspectJ. *TOSEM: ACM Transactions on Software Engineering and Methodologies*, 18(2):1–32, 2008.

[8] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*. Springer-Verlag, 2004.