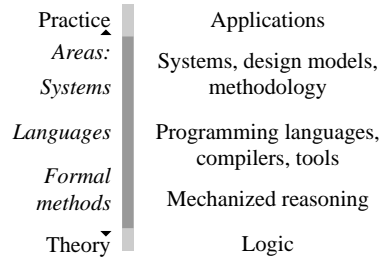


The primary emphasis of my research is on systematic design methods for computational systems, especially methods for achieving fault-tolerance and eliminating errors. There are two main sources of faults and errors in computational systems: errors that arise from *design* flaws; and errors that arise from *external* factors, such as failures in power systems, networks, computational substrates, *etc.* In both cases, the source of errors is rooted in system complexity. Large systems may be composed of parts written by thousands of programmers in separate locations at different times. Runtime environments may be global and heterogeneous, requiring the interaction of large numbers of processes and devices operating in distinct administrative domains. In a digital system, the effects of a single defect can be catastrophic, leading to system shutdown or other unintended or malicious behavior.

The methods for addressing errors fall into two categories. *Proactive* methods (my terminology) are used to simplify and clarify code, reduce code size, facilitate collaboration, and render code amenable to mechanized reasoning. *Reactive* methods include algorithms and protocols designed to detect and recover from faults at run time, especially faults due to external factors. Proactive methods are closely related to software engineering—methods for reducing errors also result in better software practice and enhanced productivity.

My research spans both of these categories. The foundation of my work is rooted in mathematical logic; the research goals are to develop languages, models, and methodologies for constructing reliable programs. A diagram of the research layering as is shown at the right. Taking a broad brush, the focus is on computational infrastructure, in between the areas of pure logic and specific applications, in the fields of *formal methods*, *programming languages/compilers*, and *systems*.



The goals of formal methods are primarily in the proactive category, to develop models and methods for simplifying and analyzing programs. The methods I use are based on mechanized theorem proving using constructive, foundational methods. I am widely known as the leader behind the MetaPRL [HNC+03] logical framework, an LCF-style [GMW79] second-order theorem prover (for higher-order logics), where second-order means that relations over logics can be stated and proved.

Generally speaking, my research in formal methods is outward directed. That is, I believe that the field is mature enough that efforts should now be directed at solving problems elsewhere, particularly in fields like programming languages, compilers, and systems.

Let's start at the foundations, with the the topic of reflection in logics and programming languages.

1 Reflection

When proposing new programming models and methods, the first choice for representing the concepts is often in a programming language (and perhaps an associated logic). The immediate issue that arises is: what are the meta-properties of the new language/logic—for example, is it sound? There is much effort underway [ABF⁺05] to show that these arguments can be mechanized, at least on a case-by-case basis. I believe that the correct way to approach this problem is to find *uniform* arguments that can be applied across a broad range of languages, and the key to uniformity is formal *reflection*.

Reflection has a long and colorful history, starting with Gödel’s use of reflection in his incompleteness theorem [Göd31]. Logically speaking, reflection offers the ability for a logic to be “self-aware” in some sense; it is also a valuable tool when using one logic to reason about another. There have been numerous attempts at formalizing reflection (Harrison [Har95] gives an excellent survey), but the best are only partly successful.

The problem starts with syntax. Gödel-numbering of formulas is completely impractical, and any kind of non-trivial transformation has a disastrous consequence: unless reflected syntax preserves the original structure of sentences identically (meaning, at least, that variables are preserved as variables), then it becomes necessary to re-encode the mechanics of the theorem prover in itself, a painful task.

My contribution to this topic is a complete account of syntactic, structure-preserving reflection (formalized and validated in the MetaPRL framework) [HNYK06a, NKYH05, GHNT04]. The result is deceptively simple—the quotation $\ulcorner t \urcorner$ of a formula t quotes the constants in the formula, but nothing else, for example $\ulcorner \lambda x. x+1 \urcorner = \ulcorner \lambda \ulcorner x \urcorner. x \urcorner + \ulcorner 1 \urcorner$. Since binding structure is preserved, mechanized reasoning in a reflected logic is exactly the same as in the original logic; the difference of course being that the reflected system offers principles for proof and structural induction.

This work builds on a sequence of contributions at the logical foundations, most notably 1) the definition of an *image type* [NK06], which allows for the intensional definition of syntax with a structural induction principle, and 2) the formulation of an induction principle over sequents (logical judgments) called *teleportation* [HNYK06b].

I consider this to be a major accomplishment, and I believe it will take some years for the consequences to be known and understood. Formal reflection is an area where constructions are delicate, and where missteps are likely to tumble the construction like a house of cards. This result comes from a four-year effort (including several failed attempts). The internals are highly technical, but the end result is not. It is appropriate to view the reflection mechanism as a compiler that takes logic or language specification as an input, and automatically produces an embedding within a meta-logic as output. To end-users, it is important that the embedding be predictable, but the intricacy of the argument that justifies the embedding is unlikely to be a major concern.

2 Compilers

Moving up a level, I am interested in *compilation*; that is, the translation of source programs to machine code. Correctness of compilation is of course critical to program correctness, but compilers are themselves large, complex systems that are difficult to design correctly. See for example, Hoare and Misra's Verified Software challenge [HM05].

My recent contributions have been in the area of formalization, or *formal compilers*, where the emphasis is on design methods for clarifying and isolating the trusted parts of the compilation path [HN06b, HNGA03, GṪNH03, GH02]. By the term trusted, I mean that a component is trusted if the correctness of the compiler depends on it. Trusted code is undesirable, yet in traditional compiler implementations, nearly all of the code must be trusted.

We have shown that, although the total size of a compiler does not necessarily decrease when it is formalized, the amount of *trusted* code can be reduced to about 1% of the total size. Furthermore, the trusted transformations are defined as a set of small-step declarative rules that can be verified for soundness separately; none is more than 10 lines of code. Some of the larger, more complex transformations (register allocation in particular), need not be trusted.

For demonstration, I have taken two-phase approach, where the objective in the first phase is to compile a fairly large, fairly realistic ML-style source language to machine code, demonstrating the algorithms and methods I have described. The second phase is to validate it. Slind *et al.* [SLO06, GIOS05] have a closely related approach, where they start with a smaller, still realistic, source language (a fragment of HOL) and perform a complete compiler verification.

This work built on my earlier research on the *Mojave compiler collection* (MCC) [HSA⁺02, Smi03], where the goals were broad support for a range of languages using a common formal infrastructure. MCC served as a compiler and programming language testbed that supported full ANSI C and fairly large fragments of Java, Python, and OCaml. The key accomplishments included 1) a *safe* runtime for ANSI C, similar to Morrisett's Cyclone [JMG⁺02] and Necula's C-Cured [NCH⁺05], 2) process migration and speculations, two topics I will talk more of in the next section. With the advent of the Microsoft CLI in 2002/2003, I reorganized the project, taking a formal approach to compilers and a systems approach to process migration and speculation.

3 Systems

At the highest level, my research focuses on models and methods for large systems, specifically distributed systems. The reason for choosing distributed systems is because they are complex, heterogeneous, and faults are unavoidable. The methods are both proactive and reactive, to develop models that not only simplify distributed programs, but also allow transparent recovery from faults.

Two specific models are *process migration* and *speculations*. Process migration is the ability to move a process from one computer to another, perhaps in response to faults. Speculations are like transactions, but they specifically omit the *isolation*

property to allow multiple processes to cooperate during the transaction. Migration and speculation are closely related; both are based on a form of checkpointing. It should be noted that these are not new concepts to distributed systems, there is prior work going back to the 1980s. However, the models are effective and simple to grasp. My research aims at providing a firm semantical foundation, especially with respect to consistency in the presence of faults.

My contributions can be placed in three categories: system *mechanisms*, group *communication*, and *filesystems*.

Originally, the snapshot/checkpointing mechanism (used for both migration and speculations) was provided through the MCC compiler and runtime [S†H07], where the compiler annotated the program with hooks for managing snapshots. More recently, we have reduced the mechanism to a pure kernel-level implementation [†H06, †H05a, †SH03], yielding *transparency*—applications need not be aware of faults and fault recovery. This is especially important for deployment on the DOE ASC platforms, and I judge it to be an important step forward. Checkpoints are based on differences, rather than logs (see for example Grossman’s work on log-based methods [HG06]). During a transaction, communication establishes dependencies with neighboring processes; again, this is handled transparently by the implementation. From a language point of view, the definition of speculations is precise [†H07, †ap06, †H05b].

I have had a continuing interest in group communication [Bir97], which provides a clean, abstract communication model with a precise semantics. However, group communication protocols (and consensus protocols in general) do not scale. Currently, my contributions have been directed at global protocols consisting of many small groups. The protocols we develop [†NH06] provide “global total order,” or more precisely, communication is globally serializable. These kinds of protocols are extremely complex; we specify and analyze them formally (this time using model checking) [†N07].

The final part is the construction of a global store, or filesystem, for data sharing. My research in this area is on decentralized file services that achieve fault-tolerance through data replication [†NGH06, †NHW04]. The filesystem uses the group communication described above to achieve global sequential consistency.

4 Tools and languages

Large systems are frequently composed of many kinds of heterogeneous parts, yet *build systems*—the tools that we use to compile and construct our systems—have received inadequate attention. Current software practice is often based on derivatives of `make` [Fel79], which was originally designed for small programs contained within a single directory. Extending it to large projects can be tricky, inefficient, and error-prone.

My contribution in this area is the OMake [HN06a] build system, which retains a fairly traditional syntax, but provides a semantics based on global project analysis. A central goal in OMake is *compositionality*, so that program components can be specified separately and composed without interference. OMake specifications are defined in a functional language with dynamic binding. Functionality provides strong assurance that build specifications can be composed without interference; dynamic binding enhances specification re-use. OMake is publicly released, and has attracted numerous users and contributors.

5 Long range goals

My research is directed at what I believe is the central issue in computer science: compositionality. The systems we study are composed of large numbers of heterogeneous components, and these systems are extremely complex. The problem of compositionality is to find *scalable* methods for constructing and understanding these systems. My research is directed squarely at this problem, approached from the perspectives of formal methods, languages, algorithms, and protocols.

I do not believe that compositionality is “solvable” in a traditional sense, or that someday this research will be “done.” Digital systems are inherently complex. However, I do believe that methods can be *systematic*, and the range of topics of my research is aimed at constructing these systematic methods.

References

- [ABF⁺05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. Available from <http://www.cis.upenn.edu/group/proj/plclub/mmm/>, 2005.
- [Bir97] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, January 1997.
- [Fel79] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software — Practice and Experience*, 9(4):255–265, 1979.
- [GH02] Adam Granicz and Jason Hickey. Phobos: A front-end approach to extensible compilers. In *36th Hawaii International Conference on System Sciences*. IEEE, 2002.
- [GHN[†]04] Nathaniel Gray, Jason Hickey, Aleksey Nogin, and Cristian Țăpuș. Building extensible compilers in a formal framework. A formal framework user’s perspective. In Konrad Slind, editor, *Emerging Trends. Proceedings of the 17th International Conference on Theorem Proving in*

- Higher Order Logics (TPHOLs 2004)*, pages 57–70. University of Utah, 2004.
- [GIOS05] Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. A proof-producing hardware compiler for a subset of higher order logic. In Joe Hurd, editor, *Emerging Trends. Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*. Oxford University, 2005.
- [GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.
- [Göd31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English version in [vH67].
- [GṪNH03] Nathaniel Gray, Cristian Țăpuș, Aleksey Nogin, and Jason Hickey. Building reliable compilers with a formal methods framework. In Dr. Indrakshi Ray, editor, *The 14th International Symposium on Software Reliability Engineering (ISSRE 2003). Supplementary Proceedings*, pages 319–320. Chillarege Press, 2003.
- [Har95] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-53, SRI International, Cambridge Computer Science Research Centre, Millers Yard, Cambridge, UK, February 1995.
- [HG06] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, October 2006.
- [HM05] Tony Hoare and Jay Misra. Verified software: theories, tool, experiments, 2005. Available as an online publication <http://vstte.ethz.ch/pdfs/vstte-hoare-misra.pdf>.
- [HN06a] Jason Hickey and Aleksey Nogin. OMake: Designing a scalable build process. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006*, volume 3922 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2006. An extended version is available as California Institute of Technology technical report CaltechCSTR:2006.001.
- [HN06b] Jason Hickey and Aleksey Nogin. Formal compiler construction in a logical framework. *Higher-Order and Symbolic Computation*, 19(2–3):197–230, September 2006.
- [HNC⁺03] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei

- Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. *MetaPRL* — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.
- [HNGA03] Jason Hickey, Aleksey Nogin, Adam Granicz, and Brian Aydemir. Compiler implementation in a formal logical framework. In *Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding*, pages 1–13. ACM Press, 2003. An extended version of the paper is available as Caltech Technical Report caltechCSTR:2003.002.
- [HNYK06a] Jason Hickey, Aleksey Nogin, Xin Yu, and Alexei Kopylov. Mechanized meta-reasoning using a hybrid HOAS/de Bruijn representation and reflection. In John H. Reppy and Julia L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 172–183. ACM, 2006.
- [HNYK06b] Jason Hickey, Aleksey Nogin, Xin Yu, and Alexei Kopylov. Practical reflection for sequent logics. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP’06)*, Electronic Notes in Theoretical Computer Science, 2006.
- [HSA⁺02] Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Țăpuș. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR 2002.007, California Institute of Technology, Computer Science, July 2002.
- [JMG⁺02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, June 2002.
- [NCH⁺05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), May 2005.
- [NK06] Aleksey Nogin and Alexei Kopylov. Formalizing type operations using the “Image” type constructor. In *Proceedings of the 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006)*, volume 165 of *Electronic Notes in Theoretical Computer Science*, pages 121–132. Elsevier, 2006.
- [NKYH05] Aleksey Nogin, Alexei Kopylov, Xin Yu, and Jason Hickey. A computational approach to reflective meta-reasoning about languages with bindings. In *MERLIN ’05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages

- 2–12. ACM Press, 2005. An extended version is available as California Institute of Technology technical report CaltechCSTR:2005.003.
- [SLO06] Konrad Slind, Guodong Li, and Scott Owens. A proof-producing software compiler for a subset of higher order logic. Draft, 2006.
- [Smi03] Justin D. Smith. Fault tolerance using whole-process migration and speculative execution. Master’s thesis, California Institute of Technology, Department of Computer Science, 2003.
- [SȚH07] Justin D. Smith, Cristian Țăpuș, and Jason Hickey. The mojave compiler: Providing language primitives for whole-process migration and speculation for distributed applications. In *HIPS/TOPMoDeLS workshop (at IPDPS 2007)*, 2007.
- [Țăp06] Cristian Țăpuș. *Distributed Speculations: Providing Fault-tolerance and Improving Performance*. PhD thesis, California Institute of Technology, Pasadena, CA, June 2006.
- [ȚH05a] Cristian Țăpuș and Jason Hickey. Distributed synchronization with shared semaphore sets. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2005), Cardiff, UK, 2005*. Workshop on Distributed Shared Memory on Clusters (DSM).
- [ȚH05b] Cristian Țăpuș and Jason Hickey. Extended operational semantics for simple distributed speculative execution. Technical Report caltechCSTR 2005.002, California Institute of Technology, Computer Science, January 2005.
- [ȚH06] Cristian Țăpuș and Jason Hickey. Speculations: Providing fault-tolerance and recoverability in distributed environments. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep ’06)*, 2006.
- [ȚH07] Cristian Țăpuș and Jason Hickey. A theory of nested speculative execution. In *9th International Conference on Coordination Models and Languages (Coordination 2007)*, 2007.
- [ȚN07] Cristian Țăpuș and David Noblet. Fixd: Fault detection, bug reporting, and recoverability for distributed applications. In *HIPS/TOPMoDeLS workshop (at IPDPS 2007)*, 2007.
- [ȚNGH06] Cristian Țăpuș, David Noblet, Vlad Grama, and Jason Hickey. Mojavefs: Providing sequential consistency in a distributed objects system. In *Proceedings of the The 5th International Symposium on Parallel and Distributed Computing (ISPDC 2006)*, 2006.
- [ȚNH06] Cristian Țăpuș, David Noblet, and Jason Hickey. Mojavecomm: A robust group communication library for grid environments. In *Proceedings of the International Conference on Networking and Services (ICNS’06)*, 2006.

- [T̄NHW04] Cristian T̄ăpuș, Aleksey Nogin, Jason Hickey, and Jerome White. A simple serializability mechanism for a distributed objects system. In David A. Bader and Ashfaq A. Khokhar, editors, *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004)*. International Society for Computers and Their Applications (ISCA), 2004.
- [T̄SH03] Cristian T̄ăpuș, Justin D. Smith, and Jason Hickey. Kernel level speculative DSM. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2003), Tokyo, Japan, 2003*. Workshop on Distributed Shared Memory (DSM).
- [vH67] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.