

May 22, 2026

University of Pennsylvania

New Jersey Programming Languages and Systems Seminar

9:00	<i>Breakfast and sign-in</i>	
9:55	Welcome!	
10:00	Principal Gradual Type Inference	David Van Horn, <i>UMD</i>
10:20	Paso: Specifying Hardware Communication as Programs	Ernest Ng, <i>Cornell</i>
10:40	Morphosyntactic Programming: Case, Mood, and Type-Directed Disambiguation for Turkish-Like Syntax	Joomy Korkut <i>Bloomberg</i>
11:00	<i>Coffee 1</i>	
11:30	Is this program doing what we want?	Matt Davis, <i>CMU</i>
11:50	Teaching Property Based Testing in a CS1 Classroom	Xiaorui Liu, <i>UPenn</i>
12:10	On the Reliability of Code Comprehension Proxies	Erfan Arvan, <i>NJIT</i>
12:30	<i>Lunch</i>	
13:50	Strata: A Unified Foundation for Code Reasoning Tools	Josh Cohen, <i>AWS</i>
14:10	Algebraic Factorization of Languages for User-Defined Program Synthesis	Oliver Daidis <i>Cornell</i>
14:30	Staging Module Functors	Maite Kramarz, <i>U. Toronto</i>
14:50	<i>Coffee 2</i>	
15:30	Semantics, Operations, and Properties of P3109 Floating-Point Representations in Lean	Tung-Che Chang <i>Rutgers</i>
15:50	Incr: Faster Re-execution via Bolt-on Incrementalization	Vagos Lamprou, <i>Brown</i>
15:10	Leveraging Types and Typestate for Peripheral Cost Analysis	Sai Divvela, <i>UMD</i>
16:30	Pointers in OCaml	Ryan Tjoa, <i>Jane Street</i>
16:50	Closing remarks	

Wu & Chen Auditorium, Levine Hall
3330 Walnut Street, Philadelphia, PA 19104

Abstracts

Morning Session 1

Session chair: Daniel Sainati

Principal Gradual Type Inference

David Van Horn

University of Maryland

Gradual typing aims to combine the benefits of static typing with the flexibility of dynamic typing via the distinguished "unknown" type (\star). Type inference, however, has long resisted a satisfactory solution: principal types are known not to exist, and existing approaches typically rely on heuristics, sacrifice completeness, or alter program semantics. In this talk, I will highlight some of the core challenges and survey prior approaches to this and closely related problems.

I then present a new approach to type inference for gradual types. The algorithm follows a standard constraint-based structure, but relies on a novel solver for consistency constraints, obtained via a reduction to a classic graph partitioning problem: multiway cut. This gives a clean and complete solution for GTLC, and appears to extend smoothly to richer features including let-polymorphism, existential types, recursive types, and subtyping.

I also propose a refined formulation of gradual type inference in which assigning \star incurs a cost. Under cost policies that strictly distinguish the use of \star at different program points, we recover a meaningful notion of principality relative to cost. The min-cut formulation of our solver naturally yields relatively principal type assignments for any given cost policy.

Finally, I show how the same machinery can be applied to gradual migration, producing precise type annotations that preserve program behavior while addressing several limitations of prior approaches.

Paso: Specifying Hardware Communication as Programs

Ernest Ng

Cornell University

Testing and debugging hardware designs is notoriously challenging, in part due to a mismatch in abstraction levels. Testbenches operate at the level of individual clock cycles and signals, but a hardware module's desired I/O behavior is best articulated in terms of high-level communication protocols.

This mismatch affects two parts of a hardware testing workflow. First, a driver program translates high-level operations into multi-cycle signal interactions on the module under test. Second, a monitor program recognizes high-level transactions from a signal-level execution trace. These two tools are typically implemented separately, entailing duplicated engineering effort and risking inconsistencies.

We advocate an alternative approach: for any given hardware protocol, write one program to generate both the driver and the monitor. We propose Paso, a DSL in which users specify hardware components' communication interfaces as succinct imperative programs. The same Paso program both drives hardware modules for testing, and is used to automatically infer high-level transactions from signal traces. We

demonstrate how our tools infer high-level transactions and uncover bugs in open-source implementations of the AXI-Stream and Wishbone protocols.

Morphosyntactic Programming: Case, Mood, and Type-Directed Disambiguation for Turkish-Like Syntax

Joomy Korkut

Bloomberg

We present Kip, a statically typed functional programming language with a syntactic purity discipline, whose surface syntax is in Turkish and whose type checking is guided by Turkish morphology. Kip uses case suffixes to determine which parameter each argument belongs to, so arguments can be given in any order when the cases uniquely identify them. Kip also carries morphological ambiguity through parsing and resolves it during type checking via type-directed constraint solving. Finally, Kip enforces a lightweight effect discipline via syntax: pure definitions are noun phrases, while effectful computations are infinitival verb phrases, and effectful calls appear in the imperative mood. We also describe the Rocq mechanization of Kip's calculus, which proves progress, preservation, and elaboration correctness. This is joint work with Alperen Keles (University of Maryland, College Park) and Onur Akdemir (Topsort).

Morning Session 2

Session chair: Benjamin C. Pierce

Is this program doing what we want?

Matt Davis

Carnegie Mellon University

Practicing software engineers frequently grapple with a fundamental question: is this program doing what we want? This requires navigating the "we" and the "want" of a specific context to define what "right" actually looks like. However, identifying critical edge cases is often tedious, and software engineers may struggle to conceive of these scenarios without a concrete example to surface them. In this talk, I will present findings from my research into the human factors of property-based testing tools and discuss how automatically-generated examples can elicit deeper thinking about program behavior.

Teaching Property Based Testing in a CS1 Classroom

Xiaorui Liu

University of Pennsylvania

Property-based testing (PBT) is a powerful software testing technique. It is also a natural introduction to thinking about formal specification, making it valuable to introduce early to computer science students. But the details of how to teach PBT effectively at the undergraduate level remain under explored.

We investigated the experience of students learning PBT in a large CS1 course, integrating it into two core assignments and extensively measuring students' experiences and reactions. We found that students can learn to apply PBT quickly and fairly effectively—a promising sign—but that they struggle with some key aspects of its workflow. In particular, students struggle with reasoning about "property efficacy," the ability of a property to discriminate good from buggy implementations, in a structured way.

Our work suggests opportunities for future pedagogy, tooling, and resources to advance both the mastery of program specification in introductory computing and the adoption of PBT.

On the Reliability of Code Comprehension Proxies

Erfan Arvan

NJIT

Developers spend 58–70% of their time understanding code, yet we do not know how to measure code comprehensibility, let alone how to reliably improve it. Existing automatic proxies, such as cyclomatic complexity, have long been used as approximations of comprehensibility, but have been shown to be unreliable.

To find better automatic proxies, we need to validate them against measurements of human performance on comprehension tasks. However, this is difficult: the proxies used in prior human studies—such as Likert-scale ratings or task performance measures—are themselves unreliable and often arbitrary. Worse, there is no established ground truth for comprehensibility against which such proxies can be evaluated.

In this talk, I present a study that addresses this problem by constructing a reference notion of comprehensibility based on expert agreement. We first conducted an expert study with professional software engineers to establish a ranking of code snippets by comprehension difficulty using a Delphi-style protocol. We then conducted a study with 44 student participants on the same snippets, measuring 14 commonly-used proxies from the research literature. Finally, we analyzed how well each proxy aligns with the expert-derived ranking.

The results show a clear pattern: proxies that require reasoning about program behavior—such as answering input-output questions—are the most reliable, especially when measured using response time. In contrast, proxies based on syntactic questions perform poorly, often showing weak or even negative correlation with expert judgments.

These findings have implications for prior work on code comprehensibility. Many studies rely on such proxies to evaluate and draw conclusions about code comprehensibility, such as when comparing alternative implementations or assessing which coding practices improve comprehensibility, including control-flow structures, API designs, and uses of language features. If the proxies are misaligned with actual comprehension, then the conclusions of such studies may be unreliable.

Afternoon Session 1

Session chair: Michael Greenberg

Strata: A Unified Foundation for Code Reasoning Tools

Josh Cohen

Amazon Web Services

Automated reasoning tools are achieving success at scale across diverse domains, enabled by a rich ecosystem of analyses (deductive verification, model checking, abstract interpretation, etc.) targeting numerous languages (Java, C, Rust, Python, SystemVerilog, etc.). However, these tools are generally single-purpose, targeting a specific language and/or analysis. This prevents improvements from being reused

across toolchains and results in significant duplicated effort. To address this, AWS is developing Strata, a unified framework for code reasoning that enables diverse reasoning applications across multiple front-ends and back-ends through a common foundation.

Strata shares some goals with existing Intermediate Verification Languages (IVLs) such as Boogie, Viper, and Why3, but extends beyond them in three key ways. First, Strata is designed to support many different program analyses, not just the SMT-based deductive verification offered by traditional IVLs. Second, rather than a single intermediate language, Strata provides a family of intermediate languages for expressing programs at different levels of abstraction, along with transformations among them that enable various analyses. Strata also includes general-purpose tools for defining intermediate languages, including a domain-specific framework that automatically generates language definitions, parsers, and (de-)serializers. Third, Strata is implemented in Lean and includes formal semantics and proofs of correctness for its transformations, enabling foundationally sound reasoning across languages and analyses. This is complemented by property-based and differential testing of Strata's implementation.

Strata is open-source and underpins a key pillar of Lean's in-progress CSLib initiative. In this talk, I will give an overview of the Strata platform and its capabilities, and discuss opportunities for contributing to Strata and its ecosystem.

Algebraic Factorization of Languages for User-Defined Program Synthesis

Oliver Daidis

Cornell University

Often, we want to intermix user-written code in a low-level language and machine-generated code from a high-level language. For example, consider inline assembly in C, autogenerated schedules with partly human-written sections in user-scheduled languages, or LLM-generated code in a project with predominantly human-written code. Ensuring predictable behavior requires defining a boundary between the code-generator and the user. However, this boundary does not necessarily align well with function or module scopes, motivating mechanisms for finer-grained behavioral constraints on code generation.

We introduce a generalizable mechanism for allowing user-definable high-level languages based on user-defined algebraic specifications. The user provides an algebraic description of their high-level goal and the type checker converts expressions of the high-level language into type constraints on the low-level language. Our type system facilitates a simple program synthesis scheme for common cases and has a lightweight, linear-time type-checker for programs with no remaining holes for synthesis. Furthermore, our type system allows constraint by multiple high-level languages addressing possibly disjoint aspects of the semantics of the low-level language. If time allows, we will discuss extensions allowing user-defined code-generators.

Staging Module Functors

Maite Kramarz

University of Toronto

MacoCaml is an extension for OCaml which introduces staging constructs to create and manipulate program fragments. Using these, it becomes possible to write high-level, abstract programs which themselves generate efficient, specialized, and provably type-safe code. While module functors can currently be supported by this system, they come with the requirement that any code generation within functor bodies is delayed until application-time, even if those macros are safe to expand. This restriction makes it impossible to write macros that are shared across various possible instantiations of a functor, and it

is also incongruous with how MacoCaml treats code generation under lambda bodies. In this talk, we will discuss how closure conversion and splitting can be used to overcome this limitation, as well as build up the novel vocabulary of module-level quotation needed to enable it.

Afternoon Session 2

Session chair: Richard Eisenberg

Semantics, Operations, and Properties of P3109 Floating-Point Representations in Lean

Tung-Che Chang

Rutgers University

The upcoming IEEE-P3109 standard for low-precision floating-point arithmetic can become the foundation of future machine learning hardware and software. Unlike the fixed types of IEEE-754, P3109 introduces a parametric framework defined by bitwidth, precision, signedness, and domain. This flexibility results in a vast combinatorial space of formats—some with as little as one bit of precision—alongside novel features such as stochastic rounding and saturation arithmetic. These deviations create a unique verification gap that this paper intends to address.

This paper presents FLoPS, Formalization in Lean of the P3109 Standard, which is the first formal model of P3109 in Lean. Our work serves as a rigorous, machine-checked specification that facilitates deep analysis of the standard. We demonstrate the model's utility by verifying foundational properties and analyzing key algorithms within the P3109 context. Specifically, we reveal that FastTwoSum exhibits a novel property of computing exact "overflow error" under saturation using any rounding mode, whereas previously established properties of the ExtractScalar algorithm fail for formats with one bit of precision. This work provides a verified foundation for reasoning about P3109 and enables formal verification of future numerical software.

Incr: Faster Re-execution via Bolt-on Incrementalization

Vagos Lamprou

Brown University

While most software development is incremental, most execution environments are not: even small program modifications fail to take advantage of prior executions, at worst requiring full re-execution of all computational stages in the modified program. Such full re-execution decelerates software development and debugging, especially in dynamic polyglot environments such as the Unix and Linux shell.

This talk will present Incr, a system that accelerates the re-execution of unmodified shell programs by automatically incrementalizing their execution. Incr analyzes and tracks interdependencies to detect and store key intermediate results, reusing them on subsequent re-executions whenever possible. Incr's effect analysis guarantees correct re-execution even for non-idempotent computations, and several static and dynamic optimizations reduce the runtime and storage overheads of incrementalization. Applied to diverse real-world scenarios, Incr accelerates re-execution by an average of 34.2× and a maximum of 373.3×—all while requiring no developer annotations or code modifications and remaining behaviorally indistinguishable from non-incremental execution.

Leveraging Types and Typestate for Peripheral Cost Analysis

Sai Divvela

University of Maryland

As IoT devices become increasingly prevalent, embedded systems operating in resource-constrained environments must carefully account for their energy requirements. These requirements are typically expressed as worst-case energy consumption (WCEC), the maximum possible energy consumed across any possible execution. One approach to finding this is by performing a static enumeration of all possible program paths. However, this static analysis is complicated by peripherals, which are a critical feature for sensor-driven and reactive embedded platforms, e.g., an IoT wearable. To incorporate peripherals into the analysis, there are three main costs to consider. The first is the energy cost of executing code. The second is the energy cost of using an active peripheral, and the third is the cost from having peripherals remain active in the background. The key challenge, however, is that peripheral devices are often stateful, with their energy consumption varying based on their internal state. This state is typically invisible to both programmers and static analyses, which leads to peripherals being modeled under overly restrictive assumptions or unmodeled altogether, hindering accurate cost analysis for many embedded platforms.

We address this challenge by encoding peripheral state directly into the type system, making the hidden internal state visible at compile time. For the cost analysis, we utilize automatic amortized resource analysis (AARA), a well-known type-based approach that provides cost bounds for programs. However, AARA alone lacks knowledge of the internal peripheral state. To address this, we extend AARA with typestate oriented programming. By combining the two ideas, we are able to reason about the cost of using peripherals in a more precise way. To our knowledge, we present the first type system combining AARA and typestate to provide a unified approach to cost analysis, while extending to asynchronous and interrupt-driven peripherals and enabling robust bounds on a wide variety of embedded platforms.

Pointers in OCaml

Ryan Tjoa

Jane Street

Interior pointers are a cornerstone of systems programming, used to surgically manipulate both data structures and raw bytes, as well as to express low-level APIs. However, unrestricted traditional pointers are not memory-safe in OCaml because the garbage collector moves heap objects and must track their liveness.

This talk discusses how a lightweight language feature—polymorphic indices into data—allows on- and off-heap pointers to be safely and uniformly implemented as a two-word data structure. Indices are typed like a lens (with type parameters corresponding to a base and element type), but are byte offsets at runtime.

We implemented this in OxCaml, an experimental branch of OCaml that also serves as Jane Street's production compiler. We touch on how OxCaml's existing extensions, namely, unboxed types and the mode system, extend the functionality of pointers and indices.