Research Statement

Thomas Gilray

My research develops high-performance programming-language approaches to reasoning automatically about complex systems. I publish my work at top venues in programming languages (POPL, ICFP, JFP, and PLDI), databases (VLDB), artificial intelligence (AAAI), and high-performance computing (HPDC, ICS, ISC, HiPC, and CLUSTER), aiming to bridge the gap between human-level formal systems (such as natural-deduction abstract interpreters and type systems) and their implementation in terms of lower-level data-parallel constructs (such as relational algebra and sparse linear algebra), addressing key performance and expressivity concerns that span the full system stack. My research focuses particularly on developing rich auto-tunable program analyses, strategies for declarative specification of such analyses, programming-language paradigms for deduction and chain-forward reasoning, and high-performance-computing algorithms to support scaling and parallelizing these tasks. Recently, **I lead a team of seven PIs to successfully win an NSF PPoSS Large** grant (Principles and Practice of Scalable Systems: awarding us \$5M over 5 years—**\$1.05M of which is awarded to my lab**) based on a shared vision of how to enable next-generation automated-reasoning tasks to scale effectively on supercomputers and HPC clouds. **I have also recently won an ARPA-H grant** to study secure deductive databases for electronic health records: awarding us \$10M over 5 years—**\$1.29M of which is awarded to my lab**.

Tunable Static Analysis My original motivation was to understand how to automate verification and optimization tasks in compilers for high-level functional and logical languages. Naturally, the heart of such problems is in how to tune whole-program and whole-component static analyses to achieve the necessary precision for relevant program behaviors while reigning in complexity to an acceptable degree so the analyses are practicable. Simulations of code that aim to approximate all possible program behaviors ahead of runtime are up against the halting problem and the fundamental limitations of computability: no particular style of compromise between precision and complexity will ever guarantee an analysis to be both effective and terminating. Instead, a best-effort approach must permit the analysis to be flexibly tuned, while guaranteeing reasonable bounds on both correctness and complexity. Ideally, it will permit bootstrapping analysis intelligence from simple coarse bounds on behavior that are easy to compute (such as control-flow and basic-type recovery) toward more granular, sophisticated, and context-sensitive program properties (such as shape analysis, refinement types, and symbolic path conditions). My own work has developed a general approach to polyvariant and context-sensitive analysis that is easily tuned, by adjusting a policy for abstract heap allocation, while guaranteeing soundness for all possible tunings. This means analysis sensitivies may be tuned to optimize for effective precision only, as soundness-by-construction is always guaranteed (ICFP 2016). This work also lead to an **invited follow-up** paper in the Journal of

Functional Programming (JFP 2018). I surveyed a broad range of sensitivities to show that instrumenting analyses, to increase available information with which to tune polyvariance, allows me to argue for the approach's generality. In fact, the guarantee of soundness my framework provides permits directly introspective notions of polyvariance, with powerful ramifications.

For example, a long-standing problem in context-sensitive program analyses has been how to model the call stack so-as to ensure precise call-return matching as polyvariance is increased, with more complex approches using Dyck reachability at a substantial increase in complexity, among other proposals. Using my framework for polyvariance, I was able to show that perfect call-return matching can be achieved at no asymptotic increase in analysis complexity (and a practical decrease in runtimes) with a simple introspective choice of continuation allocator (POPL 2016)—with a proof mechanized in Coq. I have also applied my framework in collaborative work on applications modeling faceted execution (CSF 2020), abstract symbolic execution for verifying contracts in stateful programs (POPL 2018), and dynamic monitoring and static verification of function totality using the size-change-termination principle (PLDI 2019).

High-performance Declarative Reasoning Unfortunately, designing a sophisticated analysis on paper is a far cry from the challenges of practical implementations. In implementing program analyses, concerns of software maintainability, correctness, and scalability are all in tension. Achieving high performance with hand-written code typically involves obfuscation of the logical or functional rules one starts with, so although an analysis may be correct on paper, subtle bugs and brittle implementation details are likely inevitable in practice. In the last few years, much of my focus has turned to this problem of designing full-stack declarative systems for reasoning which permit the direct expression of provably-correct analysis rules, to be made high performance automatically by a compiler and runtime. In service of this vision, collaborators and I have developed a data structure for GPU-based sparse linear algebra that is amenable to streaming updates in a fixed-point loop (ISC 2016, **won PRACE-ISC best paper**), an MPI-based system for parallel relational algebra (HiPC 2019, **won best paper**) with dynamic load balancing (ISC 2020, **won Hans Meuer best paper**), and a GPU-based system for relational algebra (ASPLOS 2025 and AAAI 2025) that yields far superior performance on an H100 versus 512 threads on ALCF's Theta supercomputer. Building on these underlying HPC approaches, we have developed several Datalog-based languages and approaches to program analysis. I have applied these approaches in a collaborative project with Galois, inc., to develop scalable whole-component analyses that can serve applications in automating program repair—**work funded via a DARPA VSPELLS grant** (awarding us $8M over 4 years, **$400,000 of which funds my group** through a subcontract). This collaboration has yielded advances in program-analysis design and implementation, and lead to an open-source pointer-analysis framework for LLVM, called Yapall, written in Rust using a procedural-macro-based Datalog. Another approach I am taking is to implement a Datalog that supports first-class facts, via a unified abstraction for facts and data carried down through the full compiler stack, which I show supports higher-order relations via defunctionalization and yields many-orders-of-magnitude speedups in program-analysis applications (VLDB 2025).