My mission as a programming languages researcher is to build automated and mathematically principled program analyses for continuous computations. Continuous computations comprise any program that uses continuous mathematics like probability theory or calculus. These computations pervade computer science: from machine learning (ML) to embedded systems to scientific computing and beyond. However, programmers must ensure the programs that implement these continuous computations remain efficient and robust. For example, how can a scientist check that for a range of physical conditions, a computational climate model is not overly sensitive? Likewise, how can an auditor ensure that a ML hiring model *monotonically* chooses more qualified candidates over less qualified ones? By harnessing the power of continuous mathematics, we can reframe these questions as properties about *certified bounds on derivatives*. However, without advanced mathematical training, programmers may lack the expertise needed to formalize and verify these properties. An opportunity arises. Can programming languages research automate these tasks with principled program analyses?

My research answers this question by developing automated program analyses in two popular programming language paradigms that expose continuous computations: **Differentiable Programming** and **Probabilistic Programming**. These analyses focus on certifying robustness [POPL22, OOPSLA22, OOPSLA23, AURA-SUB*, ICLR23] of continuous computations that require trustworthiness and improving performance [ESOP20, DAC21, RV21, DATE23, STTT23] of continuous computations that tolerate approximation. While decades of program analyses used foundations built atop discrete mathematics, these analyses are inapplicable when programs use continuous mathematics. By leaning *into* continuity, my research capitalizes on the continuous mathematical structure that precludes earlier techniques. These insights enable my work to bridge the gap between the capabilities of yesterday's program analyses and the needs of tomorrow's programmers. My core research contributions include:

1. The first abstract interpretation of Automatic Differentiation whose generality supports non-differentiable functions [POPL22] and higher derivatives [OOPSLA22], whose precision is optimized [OOPSLA23] and whose scalability extends to programs with hundreds of thousands of derivatives.

2. The first approach to apply continuous relaxations to probabilistic programs [ESOP20] and compile these programs to low precision hardware [DAC21, DATE23] to make inference up to 27× faster.

In addition to my research appearing in top conferences in programming languages (POPL, OOPSLA, ESOP), my work has appeared in top conferences in embedded systems/design automation (DAC, DATE) and machine learning (ICLR, CVPR). My near-term ambitions are to build general, precise and scalable program analyses for continuous computations in emerging domains by forging connections with other communities like machine learning and scientific computing. My long-term ambitions are to harness the power of continuous mathematics to expand the foundations of program analysis.

Differentiable Programming and Automatic Differentiation

Differentiable Programming which includes Automatic Differentiation (AD), serves as the backbone for machine learning and simultaneously pervades many other domains including graphics and scientific computing. Despite AD's ubiquity, automated formal reasoning about the derivatives AD computes has lagged. This absence of formal reasoning about AD code is problematic. For instance, in TinyML, when computing gradients in low precision datatypes, one must ensure the gradient's range does not overflow past the datatype's dynamic range, since overflows would ruin a computation's result. Additionally, in high-stakes social settings, ML systems may automate hiring decisions. Thus one must ensure that if two people are similarly qualified, but one has more experience, the more experienced person should be hired, otherwise that system would spread bias. Fairness in this setting is formalized as a monotonicity condition on the experience level, which equivalently means all derivatives are strictly positive.

Thus practitioners across multiple domains including both TinyML and trustworthy ML need answers to questions like "*is the largest possible gradient always less than some threshold?*" or "*are all the computed derivatives strictly positive?*". Despite the need, these questions are severely understudied.

My work answers these questions by combining abstract interpretation with AD. Abstract interpretation [Cousot] is a scalable and general program analysis framework that allows programmers to automatically reason about *sets* of program inputs instead of a single input. By merging abstract interpretation and AD, my work helps programmers automate the verification of formal properties like monotonicity that are defined over *sets* of gradients. Hence, my research builds a unified automated framework that gives programmers these formal guarantees and removes the burden of needing to know all the intricacies of calculus. Further, as in the case of monotonicity, by analyzing and verifying properties over a program's set of gradients, my work can verify properties about the original program itself.

Despite the apparent simplicity in applying existing numerical abstract interpreters to AD, the fundamental challenges of program analysis emerge: *how does one make sure the analysis is general, precise, and scalable?* In a sequence of three papers [POPL22, OOPSLA22, OOPSLA23] my research directly tackles these challenges and sets the foundation of static analysis of AD.

Generality. Formal, compositional reasoning about the semantics of differentiable programs presents challenges because computer programs are often *non-differentiable*. These points of non-differentiability stem from branch statements in the program. These mathematical pathologies in the program mean one thing: to prove formal guarantees about AD code, more generalized types of derivatives are needed.

To generalize abstract interpretation of AD to support non-differentiability, I built DeepJ [POPL22]. DeepJ grapples with non-differentiability by defining the first abstract semantics based on Clarke Generalized Jacobians. This generality allows DeepJ to reason about gradient properties for both differentiable and non-differentiable (but Lipschitz continuous) functions. This generality also means DeepJ is the first to obtain Lispchitz robustness guarantees on deep neural networks (DNNs) that are adversarially perturbed by non-differentiable perturbations like image rotations, a threat model no prior work addressed.

The need for generality also extends to *higher* derivatives and *richer* abstract domains. Formal properties are often defined over higher derivatives. For instance, the convexity of a function (e.g., a DNN), is a formal property defined over second derivatives. Previously, a programmer would have to define an AD semantics for the desired order of derivative and then prove the corresponding abstract semantics sound for a chosen abstract domain. To compute a different order of derivative or use a different abstract domain, all proofs would need to redone which puts a heavy burden on the programmer. To lift this burden, I developed the first general construction for abstract interpretation of higher-order AD [OOPSLA22]. My work creates a general framework for building sound abstract interpreters for arbitrary orders of derivatives and general classes of abstract domains. This approach removes the programmer's burden of reformalizing their abstract semantics each time they want to use a different abstract domain or compute a different derivative. Instead, programmers only specify a small set of abstract transformers for primitive functions (e.g. $\exp(x)$) and the highest desired derivative to obtain both a sound concrete and a sound abstract semantics which abstractly interprets all derivatives up to that chosen order.

Precision. The general construction I developed in [OOPSLA22] revealed the crucial need to abstractly interpret AD *precisely*. Obtaining precision is challenging since AD computations are highly nonlinear. For instance the quotient rule of calculus $\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2}$ involves four nonlinear operations: three multiplications and a division. However, decades of abstract interpretation literature focused on linear functions. Due to this lack of precise abstractions for nonlinearities, a naive abstract interpretation of AD might fail to verify properties like monotonicity due to imprecision. A fundamental question arises: *how does one tame the vast nonlinearity in AD that plagues abstract interpretation?*

To answer this question, I developed Pasado [OOPSLA23], an automated technique to synthesize precise static analyzers tailored to the structure of AD. By formulating abstract interpretation as a tractable optimization problem, Pasado optimally solves for precise abstractions of groups of multiple nonlinear operations corresponding to the chain rule, product rule, and quotient rule. Grouping nonlinear operations together prevents imprecision from compounding. Since these rules underlie forward and reverse mode AD, Pasado abstractly interprets both modes. Compared to the prior state of the art, this precision allows Pasado to compute local Lipschitz constant bounds that are over 2,000× more precise.

Scalability. Since derivative computations in AD typically have 2×-5× more operations than the original function that was differentiated, scalability becomes a primary concern. My work on Pasado addresses this concern by mathematically reducing challenging high dimensional optimization problems to 1D optimization subproblems that are efficiently solvable. Furthermore, I showed how these subproblems can be efficiently parallelized across GPUs. Thus, Pasado's optimization-based AD abstraction scales to large convolutional networks which require hundreds of thousands of derivative computations.

Probabilistic Programming

Probabilistic Programming has emerged as a powerful paradigm that lets programmers statistically model uncertainty with intuitive programs. Programmers can write statements like `x := Gaussian(m,s);` `observe(x==1.2);` without needing to know the internals of how to implement Bayesian inference. Nonetheless, difficulties arise when automating inference through the programming language. Indeed, one reason why the adoption of probabilistic modelling has lagged other ML techniques is that inference code is often prohibitively slow. To improve inference speed, faster approximate inference methods have been developed. However, it remains difficult to ensure that these approximations preserve accuracy. Additionally, one must ensure that the probabilistic programming language generates efficient code for these inference routines. One may also require *exact* instead of approximate inference to certify formal

guarantees on the probabilistic program's behavior. The situation for exact inference is even worse: it is often intractable for continuous distributions due to the need to solve complicated integrals.

To tackle these challenges, my research developed the following solutions: 1) automated program transformations that apply continuous relaxations to probabilistic programs to improve inference speed, 2) principled compilation of approximate inference to reduced precision arithmetic for generating efficient code on resource constrained devices, and 3) abstract interpretation of probabilistic programs.

Continuous Relaxations of Probabilistic Programs. Inference with continuous distributions is often more efficient than with discrete ones. This idea led me to develop Leios [ESOP20]. Leios automatically applies continuous approximations to discrete probabilistic programs to improve inference speed. By leveraging ideas from continuous mathematics, Leios replaces all discrete probability distributions with continuous approximations such as approximating a Binomial distribution with a Gaussian distribution. Leios then corrects the conditionals to account for the approximation. This insight improved the speed of Bayesian inference by 4.8× on average, with negligible accuracy loss.

Reduced Precision Probabilistic Programming on Embedded Devices. The effectiveness of Leios led me to ask how mathematical approximations can optimize the *lower* levels of the computational stack to generate fast inference code. Thus, my subsequent work Statheros [DAC21] developed a compiler for reduced precision, fixed-point arithmetic probabilistic programming. Statheros was the first probabilistic programming work to target low-resource embedded systems. By generating fixed-point instead of floating point versions of MCMC inference routines, Statheros' compilation technique improved the speed of inference by up to 27× on embedded devices without hardware floating point units.

Building upon both Statheros and my work in differentiable programming, I then developed the first compiler for reduced precision fixed-point *variational inference* on embedded systems with ViX [DATE23]. Performing variational inference in reduced precision requires computing all gradients in low precision datatypes. To avoid errors, one must ensure the gradient's range does not overflow past the datatype's dynamic range: an assurance that I provided with the AD static analysis techniques I developed in DeepJ [POPL22]. By selecting low-precision types that also avoid overflows, ViX achieved speedups of 8.15× and 22.67× over 32-bit and 64-bit floating point types respectively.

By enabling efficient Bayesian inference on embedded devices, my work opens up the potential to support fast probabilistic inference in applications like TinyML, robotics or cyber-physical systems.

Abstract Interpretation of Probabilistic Programs. Often, one needs *exact* inference results to formally verify properties of probabilistic programs. However, exact inference scales poorly, hence practitioners settle for lower and upper bounds on probabilities for verification. Moreover, these bounds can be too loose to be useful. Additionally, previous works only verify properties of a *single* probability distribution, but cannot verify properties for *sets* of distributions. This limitation prevented all prior works from analyzing the effects of dataset perturbations on probabilistic programs.

To address these challenges, in work currently under submission [AURA-SUB*], I co-developed AURA, the first abstract interpretation for probabilistic programs subject to data perturbations. By leveraging several abstract interpretation insights of [OOPSLA23], AURA reformulates the problem of certifying posterior bounds as a tractable optimization problem. This reformulation leverages insights from continuous optimization, namely pseudoconcavity, to provably find the tightest posterior bounds. AURA's generality further supports the new setting where the data inside `observe()` statements can be perturbed. Hence AURA is the first work to prove robustness for an infinite set of posterior distributions.

Other Research Contributions

In addition to differentiable and probabilistic programming, my focus on continuous computations has led to additional research contributions in both approximate computing and computer vision.

Program Analysis for Approximate Computing. Continuous computations represent an ideal application for approximate computing since coarser numerical approximations can improve performance. Nonetheless, ensuring that these approximated programs remain provably robust is critical yet challenging. To address this difficulty, I co-developed Diamont, a runtime system for enabling aggressive program optimizations [RV21] by dynamically tracking certified bounds over approximate numeric expressions. Additionally, I extended this work to verify algorithmic fairness at runtime [STTT23].

Computer Vision. I have also published in computer vision (CV) [CVPR17] which gave me useful domain knowledge to inform my work in programming languages. My CV knowledge even led me to incorporate many vision neural networks as benchmarks in my AD research [POPL22, OOPSLA23]. Applying program analysis ideas to CV also led me to supervise an undergraduate student on a project that built fast verifiers to certify geometric robustness of vision networks [ICLR23]. Armed with abstract interpretation ideas from PL, we built the first scalable verifier for general geometric perturbations like

image rotations. In addition to substantial precision improvements, this work obtained up to $42,600\times$ faster verification times than prior work, which led to orders of magnitude better scalability. These results led this paper to **win a notable designation (top 25% of papers)** at ICLR 2023.

Future Research Program

The program analyses I built for both probabilistic and differentiable computations have proven that continuous mathematics can be successfully integrated into foundational programming languages research. Looking ahead, continuous computations proliferate in emerging applications. Between the rising tides of AI, the growing importance of scientific computing (e.g., climate change modeling), and the increasing human reliance on autonomous and cyber-physical systems, programs that need continuous mathematics are inescapable. In light of these trends, my work is ideally positioned to provide new analyses and abstractions to give programmers the performance and assurances they need.

Additionally, between the NSF and DOE funding new differentiable programming initiatives within the past year and Google investing heavily in the JAX AD framework, current trends showcase how AD research already attracts funding from many sources. My research has even led to me assisting my advisor Sasa Misailovic and Professor Gagandeep Singh with the submission of a NSF Core Medium grant proposal this past October on analyzing gradients for trustworthy AI.

Building upon the skills I cultivated over my past work and the emerging needs of programmers, my future research agenda aims to address the following questions:

Program Analyses for Differentiable and Probabilistic Scientific Computing. Many of the most impactful uses of continuous computations come from scientific models, such as those used to study the spread of pandemics or the impacts of climate change. Interestingly, AD was developed by the same scientific computing community to help them differentiate the source code of their models! Hence, scientific computing including scientific ML, represents an attractive target for the program analyses I developed to ensure robust AD. I have already started to explore this intersection. My recent work Pasado [OOPSLA23] performed robust AD-based sensitivity analysis of numerical ODE solutions of both climate and chemistry models to prove monotonicity with respect to physical parameters. However, a gap exists between the PL community and scientific computing. I aim to bridge this gap by applying PL insights like verification or even program synthesis to AD programs in scientific computing. Beyond collaborations with scientific computing faculty, I aim to leverage my experience working at NASA Langley to build collaborations with government labs which represent another fertile ground for scientific computing.

Usability of Differentiable Programming. While many AD tools exist, there is limited work that studies the end user experience of AD. Indeed, many user errors of AD are under-studied [Hückelheim et al.]. One avenue for collaboration with software engineering researchers, is to customize the AD analyses I developed to try to check for these user errors and to try to find bugs in AD programs.

Differentiable and Probabilistic Analyses for Cyber-Physical Systems. Robotics and cyber-physical systems, including both Edge and IoT devices, regularly compute with continuous quantities that model the physical world. Many of these continuous computations involve computing probabilities (e.g. Kalman filters) but also derivatives for system dynamics. Ensuring the safe autonomy of these systems reduces to obtaining formal guarantees on these continuous computations. I aim to specialize the techniques I developed for ensuring robust AD and robust Bayesian inference to the specific computational patterns found in cyber-physical systems. Furthermore, the embedded processors running on cyber-physical systems and robotics platforms are often resource-constrained. Thus my research on low precision probabilistic programming [DAC21, DATE23] serves as a springboard to exploit reduced precision for other continuous computations in embedded systems. To realize this ambition, I plan to forge collaborations with the embedded/cyber-physical systems, edge computing and robotics communities.

Interpretable and Robust Machine Learning. I aim to extend my analyses of derivative properties (e.g., monotonicity) to differentiable ML models beyond neural networks. Specifically, I plan to build upon my work on probabilistic systems to develop new analysis techniques and formalize new properties for differentiable generative models like Normalizing Flows and Variational Autoencoders.

Foundational Program Analyses. In addition to the abstract interpreters I developed, AD can benefit from other verification techniques. For instance, Hoare logic lets programmers *work backwards* and reason about the set of inputs that lead to desired outputs. Thus, by adapting Hoare logic to AD, one could answer questions like "*what is the largest set of inputs for which a neural network is still guaranteed to behave monotonically?*". This effort will likely require incorporating derivative information into solvers. Hence, verification techniques such as Hoare logics, type systems, or SMT solvers offer additional pathways to develop foundational program analysis of continuous computations.

References

[Cousot] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs". In: *Proc. 4th ACM Symp. on Principles of Programming Languages*. 1977.

[CVPR17] Aidean Sharghi, Jacob S Laurel, and Boqing Gong. "Query-focused video summarization: Dataset, evaluation, and a memory network based approach". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.

[ESOP20] Jacob Laurel and Sasa Misailovic. "Continualization of probabilistic programs with correction". In: *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020*. 2020.

[RV21] Vimuth Fernando, Keyur Joshi, Jacob Laurel, and Sasa Misailovic. "Diamont: Dynamic Monitoring of Uncertainty for Distributed Asynchronous Programs". In: *Runtime Verification: 21st International Conference, RV 2021*. 2021.

[DAC21] Jacob Laurel, Rem Yang, Atharva Sehgal, Shubham Ugare, and Sasa Misailovic. "Statheros: Compiler for efficient low-precision probabilistic programming". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021.

[POPL22] Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. "A dual number abstraction for static analysis of Clarke Jacobians". In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022).

[OOPSLA22] Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. "A general construction for abstract interpretation of higher-order automatic differentiation". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022).

[STTT23] Vimuth Fernando, Keyur Joshi, Jacob Laurel, and Sasa Misailovic. "Diamont: dynamic monitoring of uncertainty for distributed asynchronous programs". In: *International Journal on Software Tools for Technology Transfer* (2023).

[AURA-SUB*] Zixin Huang, Jacob Laurel, Saikat Dutta, and Sasa Misailovic. "Precise Abstract Interpretation of Probabilistic Programs with Interval Data Uncertainty". In: *Under Submission* (2023).

[Hückelheim et al.] Jan Hückelheim, Harshitha Menon, William Moses, Bruce Christianson, Paul Hovland, and Laurent Hascoët. "Understanding Automatic Differentiation Pitfalls". In: *arXiv preprint arXiv:2305.07546* (2023).

[OOPSLA23] Jacob Laurel, Siyuan Brant Qian, Gagandeep Singh, and Sasa Misailovic. "Synthesizing Precise Static Analyzers for Automatic Differentiation". In: *Proceedings of the ACM on Programming Languages* OOPSLA2 (2023).

[DATE23] Ashitabh Misra, Jacob Laurel, and Sasa Misailovic. "ViX: Analysis-driven Compiler for Efficient Low-Precision Variational Inference". In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023.

[ICLR23] Rem Yang, Jacob Laurel, Sasa Misailovic, and Gagandeep Singh. "Provable Defense Against Geometric Transformations". In: *The Eleventh International Conference on Learning Representations*. **Designated Notable - Top 25% of Accepted Papers**. 2023.