Research Statement

Ningning Xie

Software has been the cornerstone of the modern digital world. It is gradually taking charge of all aspects of our lives: finance, healthcare, transportation, communication, education, entertainment, to name a few. Yet, despite tremendous advances over decades, building reliable software systems remains challenging: developers can easily write programs that produce wrong results, corrupt data, leak privacy, or crash. **My research applies theoretical rigor to programming language design that offers strong correctness and efficiency guarantees, with the aim of statically eradicating error-prone programs, and making it easy to build reliable, robust, and efficient software systems.**

A good language design must deliver simplicity, expressiveness, correctness, and efficiency.

- *Simplicity*: the language should be easy to use, and at its core it should be built on top of a relatively small set of well-studied language features, avoiding ad-hoc complicated language extensions.
- *Expressiveness*: the language should allow developers to write code in a concise manner, and the code should be easy to understand, extend, and reason about.
- *Correctness*: the language should help developers write correct code, by imposing static checks that maintain semantic invariants and provide safety guarantees.
- *Efficiency*: developers should be able to write efficient code that lowers costs of computational resources, time, and energy, while not sacrificing other criteria.

In my research, I design and develop accessible, expressive, efficient, and provably correct language features and tools that solve practical challenges, using *functional programming* and *type theory*. Type theory is a foundational principle that assigns properties and establishes semantic guarantees about programs. Its basic form ensures that an *ill-typed* expression like `1 + True` will be rejected. Advanced type theory can achieve stronger guarantees including data abstraction, memory safety, performance, etc., benefiting both developers and language implementors.

I design, build and implement three key aspects of languages and systems:

1. **Language design for developers to build programs easily**, by providing type-theoretical design and formalization of language features that provide correctness guarantees such as type safety, and guide the continued evolution of language implementations. (Part I)
2. **Compiler and runtime systems to run programs efficiently**, including a logic-based language design for *garbage collection*, as well as a type-safe *generative programming* technique, to allow programmers to express high-level programs that generate efficient low-level code. (Part II)
3. **Applications of programming language techniques to broader areas**. In particular, I explored utilization of programming language techniques to machine learning systems to increase reasonability, correctness and efficiency of machine learning programs. (Part III)

My work has brought broader impacts to both academia and industry. **I have received, as first authors, two Distinguished Paper awards at the premier conferences in programming languages**: POPL [10] and PLDI [7]. To apply my research results in real-world contexts, I have successfully established productive and ongoing collaborations with research teams at Microsoft Research [9, 6, 11, 7], DeepMind [1], and Google Research [4]. **My design and implementations have been integrated into widely-used open-source software** like the industry-leading Haskell Compiler GHC [16, 10, 3]. I have also worked actively on bringing emerging languages with new fundamental principles to the community, including Koka language [9, 6, 7] with effect types, and Dex language [4] with typed array programming. My research has also resulted in a number of open-source Haskell implementations and libraries [11, 6], as well as formalizations and proofs in the Coq theorem prover [17, 15, 13, 8, 12].

Echoing the insightful slogan "*well-typed programs cannot go wrong*" *(Robin Milner, 1978)*, **I envision a future where "well-typed software cannot go wrong"**, in the sense that language advances derived from functional programming and type theory can help developers build reliable and robust software systems more easily. To drive my vision forward, I am actively working on advanced programming language techniques, and I am keen to apply them to broader areas such as machine learning, data science, and many other areas that can benefit from type-theoretical programming abstraction with strong correctness and efficiency guarantees.

Research Contributions

I. Language Features: Type-theoretical Design and Formalization

Most programming languages, whether in academia like Haskell, or in industry like Java, are typed. Types have provided great expressiveness with strong static guarantees for languages. For example, *parametric polymorphism*, also called *generics*, is essential to maximize code reuse while preserving type safety.

I have a strong track record in type-based language design and formalization, to name a few: (1) For simplicity, I formally established the comparison among common object-oriented type features [12], promising an economy of both theory and implementation. (2) For expressiveness, I have shown that *intersection types* support *compositional programming* [13], allowing a system to be built by composing smaller subsystems. (3) For correctness, the work on *type classes* [14], a structural approach to overloading, for the first time establishes its resolution coherence; and the work on *union types* [2] can encode nullable types and enforce null safety.

Type-theoretical study of language features also provides a semantic foundation for language implementations. My work has uncovered and resolved flaws in real-world language design and implementations. For example, *algebraic datatypes*, used pervasively as a structured way to define sum and product types, had its *kinding* (i.e., finding the type of datatypes) result wrong when the notion of *complete user-supplied kind signature* was introduced in the Haskell compiler GHC (see its bug report #16609). I provided a type-theoretical formalization of the kinding algorithm that **is the first known, detailed account of a kinding algorithm** [10] supporting modern type features including a *dependently-typed* kind language.

This work has seen impacts in the general community. My paper on the kinding algorithm [10] received a **distinguished paper award at POPL, ACM's premier conference in programming languages**. The paper made a direct impact on GHC: beyond bug fixes, I explored alternative design decisions that increase program reasonability (e.g., I proposed *quantification checks* that use dependency analysis to reject types that fail to generalize), and **GHC later adopted my design**. The work also inspired an independent implementation of the kinding algorithm in PureScript (see its merged PR #3779), a strongly-typed functional programming language that compiles to JavaScript. As put by the implementor in a public thread, "*I found the implementation based on [10] to be straightforward. The paper was very clear and I enjoyed working through it.*"

Other type-theoretical design and formalization also serve real-world problems. My work on polymorphic *gradual typing* [15] provides a foundation for expressing polymorphic type hints and thus increases reasonability for dynamically typed languages like Python. The work was **selected for a special issue of TOPLAS, ACM's premier journal in programming languages** [8]. Moreover, existing languages often bake a fixed set of effects like exceptions and concurrency into the runtime system, and my work on *algebraic effects* [9, 6, 5] makes it possible for users to add effects independently and in a unified framework without needing special runtime support. The techniques **have been integrated into Microsoft's Koka language to generate efficient C code** [6].

II. Efficiency: Compiler and Runtime Systems

Producing performant code has been a long-standing non-trivial challenge for languages. With functional programming and type theory, we reap benefits beyond type safety: **my research has provided language designs with strong compile-time guarantees to generate fast runtime code, achieving high-level abstraction while preserving low-level efficiency**.

One critical part of program performance in the compiler and runtime system is *garbage collection*. Garbage collection is a form of automatic memory management that relieves programmers from manual memory management, but it can also have significant overhead on performance. To enable efficient garbage collection, **I designed a novel *linear resource calculus*, inspired by *linear logic*, to track resource liveness, which guides the compiler to emit precise and *garbage-free* reference counting instructions**, where a (non-cyclic) reference is dropped as soon as possible [7]. The formalization enables many optimizations that lead to competitive performance. One critical optimization is *reuse analysis*, which updates (immutable) data in-place when possible, based on the *resurrection hypothesis*: objects often die just before creating an object of the same kind. Reuse analysis leads to a new programming paradigm called *functional but in-place*: just as tail-call optimization lets programmers write loops with regular function calls, reuse analysis lets programmers write in-place mutating algorithms in a purely functional

way. The implementation of the reference counting algorithm in the KOKA compiler demonstrates that the precise reference counting technique competes with state-of-the-art memory collectors. This work has been well received by the community: our paper received a **distinguished paper award at PLDI, ACM's premier conference in programming language design and implementation**. As put by one reviewer, "*This paper reports on impressive work... and makes a reader think: why hasn't this been done before?... I think this paper has potential to become a 'classic'.*"

On the other hand, even though compilers can already optimize programs effectively, programmers, who carry more knowledge about the intent of programs, have little control over the generated programs. To give programmer fine-grained control over code generation that helps compilers to produce code with predictable performance, **I have formalized the first type-safe multi-staging programming in the context of Haskell, known as Typed Template Haskell, which allows programmers to write *staging annotations* that guide the compiler to generate efficient code**, by instructing the compiler to generate code in one stage of compilation to be used in another, effectively eliminating abstraction overhead [3]. To guarantee type safety, Typed Template Haskell implements *typed code quotations*, where code quotations are type-checked. The key challenge that threats type safety here is that multi-staging can interact with type classes in an unsound way, where *splicing* a well-typed code value can raise a type error. My work establishes a type sound semantics for Typed Template Haskell which is easy to implement and reason about, and it is expected that Typed Template Haskell will be implemented in GHC to echo the development in this formalization. **The impact of this work also reaches other language communities**: Scala3 suffers from a similar unsoundness problem with typed code quotations and *implicit parameters*; similarly, OCaml will soon face this problem as it is acquiring support for both typed code quotations and implicit parameters. **This work will help to guide the integration of these features and avoid the unsoundness problem from the outset**.

III. Applications: Programming Languages for Machine Learning Systems

Programming language research has brought insights into many areas: software-defined networks, hardware design, etc. I am broadly interested in general applications of programming languages, and recently, I am exploring how to apply programming language techniques to *machine learning* systems.

First, language design principles help to improve machine learning programming. *Array programming*, including languages like Matlab and libraries like NumPy, is the dominant programming paradigm in machine learning. While existing languages and libraries have been used widely, programmers have suffered badly from *shape errors* caused by, e.g., multiplying two matrices with unmatchable dimensions. In collaboration with Google Research, I have worked on the design of the DEX language. Dex is a functional array programming language with parallel *automatic differentiation*, which also applies *dependent types*, an advanced type feature providing strong type-level guarantees, to array dimensions: **the type system statically checks array dimensions, making shape errors easier to detect and repair**. My work aims to integrate Dex with user-defined computational effects while preserving type safety and parallelism [4]. We have shown that such design can encode practical parallel effects, including parallel accumulation, and parallel exceptions where one computation exception does not interrupt other computations. There has been an ongoing collaboration with Google Research to enrich the formalization with other type features in Dex to prepare a theoretical foundation for future implementations.

Beyond correctness, programming language techniques can also speed up machine learning systems. As another exploration, **I designed a *program synthesis* framework that accelerates deep learning in distributed machine learning training on hardware platforms** [1]. In particular, to facilitate efficient training of large-scale deep learning models, numerous parallelism techniques have been successfully employed. However, while forms of parallelism have greatly improved training throughput, they may incur significant communication overhead. My framework reduces such communication cost, by synthesizing the optimal mapping from parallelism to devices, and mapping-aware *reduction strategies* which reduce between devices using *collective operations*. **The syntax-directed synthesis framework is built on top of programming language techniques**: (1) the semantic correctness of the collective operations is established closely based on *Hoare logic*; and (2) the reduction strategies are modeled as a *domain-specific language*. Benchmark results have shown the effectiveness of the synthesis framework: a synthesized reduction strategy outperforms the default implementation with up to **2.04**$\times$ speedup. **The insights from the work have also been used to to guide a decomposition pass of XLA collectives for GPU systems and to demonstrate speed ups on training workloads at DeepMind**.

Future Directions

My past research has demonstrated great promises in applying type-theoretical principles to designing reliable and efficient programming languages and tools. Looking forward, I envision a future where programming language theory and tools are scaled to more realistic languages, applied more widely, and become more accessible. To achieve this vision, below I outline a few future directions I plan to pursue.

Programming language theory scaled to the real world. Formalization of language features usually focuses on a simplistic setting that leave out many aspects of real world implementations. This gap between theory and practice raises a serious concern: whether a theoretical result obtained is applicable to practice, and to what extent. My future research aims to address this concern by developing richer formalizations of type systems. In particular, a realistic type system must handle properties beyond whether or not an expression is well-typed: efficiency, identification of the source of type errors and production of helpful error messages, the ability to continue type-checking an ill-typed program in order to report on further errors to save programmers from multiple compilations. Following my experience in type theory [10, 17], I plan to formalize those practical aspects in type systems, bridging the gap between theory and practice. This requires to reason about more efficient data structures (e.g., the type inference engine in the Haskell compiler GHC uses the notion of *inert sets* to track type constraints rather than ordered lists as in many literatures, which is more efficient but also threatens termination), types that carry program source location (e.g., which part of the program to blame for type errors?), and better diagnosis of type errors (e.g., an earlier type error should not be re-discovered, or prevent discovery of later ones).

Programming languages for all. I believe programming abstractions, semantics, paradigms, and tools will have a significant impact on other communities, and I am excited to realize the great potential of programming language techniques to tackle challenges in a broader domain.

Machine learning programming. Machine learning has achieved remarkable successes in numerous areas, such as self-driving cars, facial identity systems, and so on. These successes also pose more serious concerns due to the lack of reasonability and correctness guarantees, causing unpredictable failures (e.g., failure to recognize a stop sign). Another line of my future research is designing and building programming abstractions and tools for machine learning programs to increase their reasonability, modularity, and efficiency. I will continue my close collaboration with leading AI companies like Google Research and DeepMind to apply my results in real-world contexts. In particular, I would like to answer the following questions: *What principles can we use for language design for machine learning programming (e.g., how to balance expressiveness and accessibility since most developers are non-experts in programming languages)? What tools can we build to reason about machine learning systems, and what guarantees can we provide (e.g., whether an implementation of a machine learning model meets its specification?)* I will start by improving existing systems (e.g., Dex [4]), both through better language design and integration with practical features (e.g., type inference for array indices). My ultimate goal is to design the next-generation machine learning programming languages and tools, and use them to improve the machine learning ecosystem.

And beyond. The same research methodology applies to many other areas. In particular, I ask the question: *what would languages and tools for domain X look like if they had been designed with programming principles in mind?* One concrete instantiation of X is data science, e.g., query language design for classic and emerging databases. For example, can we apply high-level programming abstractions to query languages to make them modular (e.g., compose multiple queries *algebraically*), easy to reason about (e.g., decide query equivalence in terms of *semantics*), and run efficiently (e.g., with multi-staging)? There are many opportunities and challenges to be solved in order to apply programming langauge theory broadly, and I am keen to pursue collaborations with colleagues and domain experts to realize the vision.

Accessible programming languages research. Programming language techniques have been useful, but programming language notations, in particular those used in type theory, are a barrier for people to understand the area, preventing practical adoption and applications of programming language ideas. For example, the standard *typing* judgment $\Gamma \vdash t : T$ makes heavy use of notations: a term (t) *has* (:) some type (T) *under* ($\vdash$) some type context (Γ). How can we communicate research on programming languages better? I believe insights from the programming language community itself can be useful in this space. In particular, there have been tremendous efforts in making programming accessible, and many analogies can be drawn between coding and writing: a piece of code (text) is easier to understand if we can jump to the definition of each function (judgment), read its document (interpretation) and search for its uses in the code base (paper), etc. I plan to explore ways to make programming language ideas (or more broadly, the general academic paper writing/reading process) more accessible.

References

[GHC] https://github.com/ghc/ghc.

[Koka] https://github.com/koka-lang/koka.

[Dex] https://github.com/google-research/dex-lang.

[PureScript] https://www.purescript.org/.

[1] **Ningning Xie**, Tamara Norman, Diminik Grewe, and Dimitrios Vytiniotis. Synthesizing Optimal Parallelism Placement and Reduction Strategies on Hierarchical Systems for Deep Learning. submitted to MLSys 2022. arXiv: 2110.10548 [cs.PL].

[2] Baber Rehman, **Ningning Xie**, Xuejing Huang, and Bruno Oliveira. Union Types with Disjoint Switches. submitted to OOPSLA. 2022.

[3] **Ningning Xie**, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. Staging with Class: A Specification for Typed Template Haskell. POPL. 2022.

[4] **Ningning Xie***, Daniel D. Johnson*, Dougal Maclaurin, and Adam Paszke. Parallel Algebraic Effect Handlers. PEPM. 2022.

[5] **Ningning Xie**, Youyou Cong, and Daan Leijen. First-class Names for Effect Handlers. HOPE. 2021.

[6] **Ningning Xie** and Daan Leijen. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. ICFP. 2021.

[7] **Ningning Xie***, Alex Reinking*, de Moura Leonardo, and Leijen Daan. Perceus: Garbage Free Reference Counting with Reuse. PLDI. **ACM SIGPLAN Distinguished Paper Award**. 2021.

[8] **Ningning Xie**, Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. Consistent Subtyping for All. TOPLAS. 2020.

[9] **Ningning Xie**, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect Handlers, Evidently. ICFP. 2020.

[10] **Ningning Xie**, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. Kind Inference for Datatypes. POPL. **ACM SIGPLAN Distinguished Paper Award**. 2020.

[11] **Ningning Xie** and Daan Leijen. Effect Handlers in Haskell, Evidently. Haskell. 2020.

[12] **Ningning Xie**, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. Row and Bounded Polymorphism via Disjoint Polymorphism. ECOOP. 2020.

[13] Xuan Bi, **Ningning Xie**, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive Disjoint Polymorphism for Compositional Programming. ESOP. 2019.

[14] Gert-Jan Bottu, **Ningning Xie**, Koar Marntirosian, and Tom Schrijvers. Coherence of Type Class Resolution. ICFP. 2019.

[15] **Ningning Xie**, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent Subtyping for All. ESOP. **Selected for TOPLAS Special Issue**. 2018.

[16] **Ningning Xie** and Richard A. Eisenberg. Coercion Quantification. HIW. 2018.

[17] **Ningning Xie** and Bruno C. d. S. Oliveira. Let Arguments Go First. ESOP. 2018.