J. Garrett Morris | Research Statement

My research is centered on the role of types and type systems in usable, expressive programming languages with strong safety guarantees. My goal is to create programming languages and type systems that support correct-by-construction programming in areas such as concurrency, systems, and low-level programming. I begin with *typed functional programming*. Typed functional programming offers numerous benefits, including usability, expressiveness, and correctness guarantees. However, domains such as concurrency, systems, and low-level programming are traditionally outside the bounds of functional type systems. To bridge this gap, my work explores the use of *linear types* in a functional setting: linearity can be used to capture correctness and efficiency guarantees for concurrent and stateful programming, while functional programming provides powerful type-directed abstraction mechanisms to make linearity more expressive and easier to use.

Typed functional programming is one of the great successes of theoretical computer science. It is the first paradigm in which programs can be correct by construction, relying on strong typing and decidable type inference to guarantee that programs are well-typed, and so are free of certain classes of errors. Higher-order typed functional programming has led to the development of new forms of type-directed abstraction. For example, monads encapsulate models of computation, such as stateful or non-deterministic computation. Ideas from functional programming have been successfully adopted in other programming paradigms. C#, Java, and C++ all now support anonymous functions (lambda expressions). Other language features that originated in functional languages include comprehensions, as now appear in Python, and explicit option types, as now appear in languages like Swift and as nullable types in C#. Monads, a relatively abstract idea, have inspired language features like Language-Integrated Query in the .Net framework languages [20].

Many areas, such as concurrent and low-level programming, are still largely outside the bounds of traditional functional type systems. This makes showing correctness in these domains harder. Proving the correctness of systems programs, for example, has involved heroic verification efforts, requiring hundreds of thousands of lines of proof script to show the correctness of only hundreds of lines of C code [15]. *Linear types* provide a mechanism to capture stateful safety properties using type systems, and have been used for assuring the safety of pointers [1] and copy-free message passing [3]. Existing linear calculi are inexpressive, however, limiting their adoption in practical settings.

I discuss three areas of my current research. The first is *instance chains*, an improvement to ideas that underlie the type systems of Haskell and other languages. The second is my work on *session types*, which describe and enforce communication protocols using types. The third is *Quill*, a linear functional calculus which provides the usability and expressiveness of languages like Haskell while guaranteeing safety for concurrent, stateful, and low-level programming tasks. I then give an example of how these ideas can be used to provide better safety guarantees for programming with file I/O. Finally, I propose a program of further research, building on these ideas to achieve practical programming with linear types. This work sits at the intersection of current research trends in linear types and the use of high-level languages for low-level software, and offers synergies with the systems and verification communities.

1 Instance Chains

My graduate work developed *instance chains* [21, 24], a generalization of type classes. Type classes [33] are a programming language feature that provide sound, static typing for type-directed overloading. Their key contribution is the introduction of *qualified types*, which use logical predicates to restrict polymorphic (generic) types. Type classes were introduced in Haskell, but have since been adopted in other language, including Agda, Coq, and Isabelle, and are closely related to the ideas of concepts in C++ and traits, as seen

in languages like Scala and Rust. While they were initially used for relatively simple ad hoc overloading, such as the application of the addition operator at both integral and fractional types, type classes have proven to be extremely expressive. They have been used, among other things, to encode extensible records (i.e., objects) [14], and to capture units of measure in types [5, 26].

Instance chains extend the expressiveness of type classes, building on their logical underpinnings. They extend the language of predicates to include negation. This can be used to capture semantic properties, such as equality being inapplicable to functions. It can also capture safety properties, such as prohibiting bitwise manipulation of pointers. Instance chains also provide a more flexible means of defining type classes. For example, they can be used to refine overloaded definitions in a modular way. These two aspects of instance chains combine to provide a significant additional increase in expressive power. For example, I have developed an encoding of extensible variants (i.e., extensible data types) using instance chains [21] that avoids many limitations present in other such encodings [29]. Since I developed this technology, a related feature, closed type families [8], has been implemented in the Glasgow Haskell Compiler. However, instance chains provide significant expressiveness benefits even compared to closed type families [22].

2 Session Types and Linear Logic

My postdoctoral work has focused on *session types* [10] and their connections to classical linear logic. Session types capture protocols in the types of communication channels, and guarantee that well-typed processes follow those protocols. A central notion in session typing is compatibility (also called duality) between protocols; for example, if one process expects to send an integer, the other process should expect to receive an integer. Recent work has identified a propositions-as-types correspondence between session-typed process calculi and cut elimination in both the intuitionistic and classical linear logics [6, 32].

My work with Sam Lindley [17, 18] explores this correspondence, and the associated concurrency, in a functional setting. We have developed a core functional language with session-typed concurrency, and shown that is has a semantics-preserving correspondence with Wadler's linear-logic inspired process calculus. We have also characterized deadlock freedom for this language, and shown why well-typed processes do not deadlock. This program of work makes two further contributions. First, it allows us to relate session-typed concurrency to other notions of computation. We have shown that continuation-passing style, a standard technique to express explicitly sequential control in pure λ-calculi, is sufficient to encode session-typed concurrency as well. Second, it allows us to extend our core language without losing deadlock freedom. One example is our formulation of recursion in session types. We identify both recursive and corecursive patterns of communication, where previous work has identified only the corecursive pattern [30]. We have repaired a defect in most existing formulations of duality for recursive session types [2, 11]. Finally, we have shown that we preserve deadlock freedom, even in the presence of non-termination.

3 Linear Types in Functional Languages

Linear types assure that values are used exactly once. Session typing depends on linearity to require that processes neither repeat previous protocol steps nor skip future protocol steps. Linearity has also been used to guarantee safety invariants of stateful programming, such as the correct use of handles, permissions, or type-changing updates [25]. Finally, linearity can assure performance, such as by allowing safe in-place updates of large in-memory data structures in a pure functional language [7]. However, existing linear calculi are less expressive than their non-linear equivalents, particularly in higher-order programming, limiting the adoption of linear types despite their wealth of applications.

I have recently proposed a new calculus, called *Quill*, which combines linear types and functional programming [23]. Its key contribution is in the treatment of higher-order functions. Higher-order functions

are central to the expressiveness of functional programming, and foundational for defining new abstractions, such as functors and monads in Haskell. Quill is the first linear calculus to provide a fully-general treatment of higher-order functions. It does so by combining a standard linear functional calculus with a novel use of qualified types [13]. Quill preserves both the usability and expressiveness of modern functional languages. Terms in Quill have principal (most general) types, type inference is decidable, and well-typed terms in a simple (non-linear) functional calculus are typable in Quill, with types that include their linear uses.

One example of Quill's expressiveness (described in detail in my paper [23]) arises from monads, which encapsulate models of computation. Existing linear calculi [19, 31] cannot express a general notion of monads; instead, they treat monads in which subcomputations execute linearly (such as for state) separately from those in which subcomputations may execute repeatedly or not at all (such as for exceptions). Many combinators can be defined for all monads. One example is the `sequence` function, which combines a list of monadic computations into a single monadic computation. Existing linear calculi would need at least two distinct variants of the `sequence` combinator. Quill has one `sequence` combinator, usable in all monads.

Example: Using Types to Enforce Correct Handle Usage

This example demonstrates the use of instance chains and linear types in assuring the correct usage of C-like file I/O operations. While this API is quite simple, it still admits a number of programmer errors. The programmer may attempt to write to a file opened only in read mode, or read from a file opened only in write mode; either results in runtime failure. I will show that instance chains can be used to enforce that modes are well-formed, and that usage of handles corresponds to their modes. The programmer must also ensure that file handles are closed after use. This can be particularly error-prone when a single file handle is shared among different functions (or objects), which must then "agree" on which function is responsible for closing the handle. I will show that linear types can enforce the lifetimes of handles. In this example I use the syntax of Habit, an experimental Haskell-like language that includes instance chains and linearity. However, I hope the syntax will be intuitive even to readers without familiarity with Haskell.

Mode strings (instance chains). The first challenge is describing valid mode strings, and associating mode strings with valid operations on file handles. We restrict our attention to opening files for reading, writing, and appending (that is, the mode strings `"r"`, `"w"`, `"w+"`, `"a"`, and `"a+"`). Rather than expecting programmers to manipulate mode strings directly, we introduce (abstract) constants, such as `readMode`, `writeMode`, and `appendMode`, to describe mode strings. The collection of mode strings available on modern platforms is extensive, and can be arcane and platform specific. Rather than introduce different constants for each possible combination, we provide a generic operator `/\` to combine basic modes. For example, the term `readMode /\ writeMode` would correspond to the mode string `"w+"`.

Next, we must associate mode strings with allowed operations on file handles. We do this by defining a type-level representation of permissions; in this case, we can rely on a very simple language including only `R` for reading, `W` for writing, and `p ⊗ q` for the combination of permission `p` and permission `q`. We then associate the mode constants with the corresponding permissions, so `readMode` would have the type `Mode R` while `appendMode` would have the type `Mode W`. Finally, the mode combining operator `/\` must also combine permissions, so we would give it the type `Mode p → Mode q → Mode (p & q)`. Here `p` and `q` range over arbitrary permissions, and `&` is a type-level mapping from permissions `p` and `q` to a combined permission. This mapping may be partial in some cases; for example, by not combining `W` permissions, we can rule out nonsensical combinations like `writeMode /\ appendMode`. The `/\` operator and `&` mapping can be defined simultaneously using instance chains, in a straightforward manner.

Opening and closing (linearity). Now we can consider the interface to files. We will leave the inner details of file handles abstract, and give the type signatures for the primitive operations. Opening a file

requires a file mode, and returns a file with the corresponding permissions. (The `IO` type is used to describe all computations that can perform input or output, including file, network, and console I/O.)

```
open   :: String → Mode p → IO (File p)
```

The type of file handles, `File p` is linear; that is, handle must be used exactly once. Re-opening a file consumes a handle and a mode string, and returns a handle with the new permissions. Linearity plays a central role here: if the old handle were not consumed by `reopen`, then it could be used with the now incorrect permissions.

```
reopen :: File p → Mode p' → IO (File p')
```

Finally, closing a file consumes the file handle.

```
close  :: File p → IO ()
```

Reading and writing (linearity and instance chains). When reading from or writing to files, we need to check whether the handle has the appropriate permissions. We can do so using type classes. We introduce a type class predicate `Has p q` that holds when permission `p` includes permission `q`. For example, we would expect predicates `Has R R` and `Has (R ⊗ W) R` to hold, but not predicate `Has W R`. With this predicate, we can now give the types of the file I/O operations:

```
read   :: Has p R ⇒ File p → Int → IO ([Byte], File p)
write  :: Has p W ⇒ File p → [Byte] → IO (File p)
```

Because file handles are linear, each I/O operation returns a "new" copy of the handle to be used afterward.

The predicate `Has` itself can be defined using instance chains. Here is one possible implementation:

```
1  instance Has p p
2  else     Has (p ⊗ p') q if Has p q
3  else     Has (p ⊗ p') q if Has p' q
4  else     Has p q fails
```

Line 1 provides the base case; lines 2–3 recursively check conjuncts in a `p ⊗ q` permission, while line 4 explicitly eliminates cases like `Has R W`. This code is straightforward and intuitive, despite defining a type-level predicate.

Putting it all together. Here is a small example of buggy file I/O code.

```
1  do hr      ← open "sample" writeMode
2     (x, hr) ← read hr 10
3     hw      ← reopen hr writeMode
4     (y, hr) ← read hr 10
5     hw      ← write hw (fun x y)
6     close hr
```

The intention is to open the file `sample` for reading, read two 10-byte values, re-open the file for writing, and write the result of applying some function `fun` to the two read values. However, this code is erroneous in several respects. In line 1, the file is opened for writing instead of for reading. This will be detected by the type system in line 2, where it will indicate that the handle `hr` does not have the `R` permission. Lines 3 and 4 are swapped: the second value should be read before the file is reopened. In this case, the type system will catch that the value `hr` is used twice, whereas (because it is linear) it is only allowed to be used once. Finally, line 6 has a typo: it closes handle `hr`, where it should close `hw`. Again, this will be detected because of the linearity: the result `hw` of `write` is used no times, whereas it must be used once.

Research Proposal: Toward Practical Linear Programming

I propose a program of further research towards *practical functional programming with linear types*. This will build on Quill to provide the core typing discipline, instance chains to provide expressive abstractions on top of Quill typing, and can be set in the context of my experience with applications of linear types. The objective of this work is to show that linear higher-order programming is beneficial for practical programming problems. This will help bridge the gap between functional and systems programming languages, and provide stronger foundations for correct-by-construction low-level programming.

Potential impact. This research will have significant impact, academically and in industry. Academically, it sits at the interaction of two trends in current research and funding. The first is on the development and use of linear type systems; recent examples include work on linear functional programming [19, 31] and session types [6, 32]. The second is the use of high-level languages for low-level programming; recent examples include work using Rust [16] and Cogent [27]. Funding in this area includes the DARPA HACMS project, which seeks to use type systems and verification to assure the safety of cyber-physical systems. This research can have significant synergies with work in the verification community, providing much stronger guarantees than can be provided by current type systems, and so simplifying and reducing manual verification efforts. Finally, linear type systems have growing industrial uptake, including uses of Rust at Mozilla and the Singularity project at Microsoft.

Research objectives. This work has three primary technical objectives.

- *Abstracting linearity.* There are numerous proposals for encapsulating and abstracting the use of linear types, including applications of monads [7], fractional permissions [4], and adoption and focus [9]. A common aspect of these proposals is that they are all described as features of the language itself. Using Quill, these ideas can be expressed within the language (i.e., as libraries) instead of as part of the core language. As in my existing work on monads, exploring these ideas in a linear functional setting will give rise to new insights and refinements of existing forms of abstraction.

- *Linearity in practice.* There are several application domains that would benefit from both high-level programming techniques and linear types. One such domain is microkernel and embedded systems programming. There has been significant interest in using higher-level languages for writing low-level code; examples include the pioneering work on Cyclone [12], and more recently efforts to write operating system components using Rust [16] or Cogent [27]. I believe that Quill can make a unique contribution in this space, as it is designed to support the full range of type-directed abstractions provided by higher-order programming languages. For example, monads could be used to sequence and control side effects in security-sensitive code, such as system call implementations.

- *Linearity in compilation and run-time systems.* Linear type systems provide a basis for characterizing deterministic deallocation of linearly-typed values, directly in the source language. Further, because Quill tracks duplication of discard of all values, it may provide a basis for deterministic deallocation of linearly-used values, even if they are not of linear type. Finally, it will be possible to infer that individual programs do not require much of the run-time support traditionally required by functional languages, or that particular segments of code (such as inner loops) avoid allocation and so will not be interrupted by garbage collection. This exploration should show that Quill can provides similar benefits to those arising from uniqueness types [28] or usage analysis.

References

[1] Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 78–91, 2005.

[2] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. On duality relations for session types. In *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers*, pages 51–66, 2014.

[3] Viviana Bono and Luca Padovani. Typing copyless message passing. *Logical Methods in Computer Science*, 8(1), 2012.

[4] John Boyland. Checking interference with fractional permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, pages 55–72, 2003.

[5] Bjorn Buckwalter. The `dimensional` package. http://hackage.haskell.org/package/dimensional, 2016.

[6] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*. Springer, 2010.

[7] Chih-Ping Chen and Paul Hudak. Rolling your own MADT - A connection between linear types and monads. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 54–66, 1997.

[8] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 671–683, San Diego, California, USA, 2014. ACM.

[9] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 13–24, 2002.

[10] Kohei Honda. Types for dyadic interaction. In *CONCUR*. Springer, 1993.

[11] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[12] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yinlang Wang. Cyclone: A safe dialect of C. In *USENIX annual technical conference*, Monterey, CA, 2002.

[13] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[14] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 96–107, Snowbird, Utah, USA, 2004. ACM Press.

[15] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[16] Amit A. Levy, Michael P. Andersen, Bradford Campbell, David E. Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: experiences building an embedded OS in rust. In Shan Lu, editor, *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015, Monterey, California, USA, October 4, 2015*, pages 21–26. ACM, 2015.

[17] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015.

[18] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN international conference on Functional Programming*, ICFP '16, Nara, Japan, 2016. ACM.

[19] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In *TLDI*, 2010.

[20] Erik Meijer. The world according to LINQ. *ACM Queue*, 9(8):60, 2011.

[21] J. Garrett Morris. *Type Classes and Instance Chains: A Relational Approach.* PhD thesis, Portland State University, May 2013.

[22] J. Garrett Morris. Variations on variants. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 71–81, Vancouver, BC, 2015. ACM.

[23] J. Garrett Morris. The best of both worlds: Linear functional programming without compromise. In *Proceedings of the 21st ACM SIGPLAN international conference on Functional Programming*, ICFP '16, Nara, Japan, 2016. ACM.

[24] J. Garrett Morris and Mark P. Jones. Instance chains: Type-class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, Baltimore, MD, 2010. ACM.

[25] Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. L^3: A linear language with locations. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 293–307. Springer, 2005.

[26] Takayuki Muranushi and Richard A. Eisenberg. Experience report: type-checking polymorphic units for astrophysics research in haskell. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 31–38. ACM, 2014.

[27] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby C. Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: bringing down the cost of verification. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 89–102. ACM, 2016.

[28] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Graph Transformations in Computer Science, International Workshop, Dagstuhl Castle, Germany, January 1993, Proceedings*, volume 776 of *Lecture Notes in Computer Science*, pages 358–379. Springer, 1993.

[29] Wouter Swierstra. Data types à la carte. *JFP*, 18(04):423–436, 2008.

[30] Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers*, pages 159–175, 2014.

[31] Jesse A. Tov and Riccardo Pucella. Practical affine types. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 447–458. ACM, 2011.

[32] Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

[33] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, Austin, Texas, USA, 1989. ACM.