My primary research interests include functional programming languages, embedded domain-specific languages, type systems, and combinatorics, and ways that these topics can contribute to other areas of computer science. More broadly, running through all of my research are three major themes which shape and direct my efforts:

- I am highly motivated by **beauty** in its many forms. I learn new things because they are beautiful; I strive for beauty in all my creative output; I am motivated to communicate an appreciation for beauty in my teaching. Beauty also correlates with generality: the most beautiful solutions are the ones that have cross-discipline applicability.

- I love the process of designing beautiful, coherent visions and then following through on the details to make those visions into reality, which I call **architecting**. I am not interested in theory just for theory’s sake, nor in just hacking out details, but in a harmonious synthesis of the two.

- Architecting and appreciating beauty always happen in the context of community. I love **communicating** ideas, through both teaching and writing, and see communication and collaboration as a central and necessary aspect of my research. Good research is not a solitary pursuit, but a dialogue among researchers—I am always on the lookout for other researchers and students to collaborate with.

In what follows, I describe several current areas of research, highlighting the ways they fit into the above themes, and elaborate on planned future directions for my research.

**Domain-specific languages and type systems**

Almost any domain can benefit greatly from the principled design of **domain-specific languages** (DSLs) and **type systems**. Such languages enable more economical communication of problems and solutions in the domain, and abstracting away domain-irrelevant detail enables higher-level thinking and new insights. Moreover, the process of constructing domain-specific languages itself often sheds new light on the domain under consideration, since it exposes fundamental questions about the meaning of entities and operations in the domain.

Type systems play an important role in the design of expressive DSLs. Some of my earlier work was concerned with extending Haskell’s type system to allow “promotion” of values to types [4], a feature particularly useful for DSLs embedded in Haskell. That work has made a big impact in the functional programming world, garnering nearly 50 citations over the past two years.

My current work on embedded domain-specific languages focuses on the areas of graphics and animation. For the past six years I have been developing a domain-specific language,
diagrams, for describing vector graphics (http://projects.haskell.org/diagrams). It is embedded in the Haskell programming language, with emphases on expressivity, elegant design, and careful analysis of the underlying domain. This careful analysis often leads to the discovery of mathematical abstractions that elegantly capture the essence of some aspect of the domain. For example, as described in a Haskell symposium paper [3], diagrams are represented using a novel tree structure with both upwards- and downwards-accumulating annotations, based on the theory of monoids and monoid actions.

Future directions

I am especially excited about this area of research since it provides many potential avenues for student involvement, across a wide range of backgrounds and interests. Students can get their feet wet helping with the implementation of a DSL or brainstorming new features, or more advanced students can explore type systems and the mathematical semantics underlying new DSLs.

diagrams currently makes use of a preliminary domain-specific language for constructing animations (and more generally, any time-varying values). In collaboration with Andy Gill at the University of Kansas and Nick Wu at the University of Oxford, I have been working on a complete redesign based on the theory of 2-categories—despite the rather abstract underpinnings, the resulting API promises to be expressive, elegant, and practical. The insights we have gained will likely be relevant for other time-centric domains as well, such as music and signal processing.

Another direction in which I would like to extend the diagrams framework is adding interactivity. Currently, users can describe static or animated diagrams, but there is no easy way to create diagrams which respond to input. What are the underlying semantics of interactivity? What would it look like to have a convenient language in which to describe interactive things? Part of the answer may lie in recent work on functional reactive programming, but finding a natural way to integrate this work with diagrams will likely raise new questions and lead to new insights.

In addition to interactivity in the end result, what about interactivity in the process of constructing diagrams itself? The idea is to develop a bidirectional user interface which allows a user to construct a diagram either by writing code or by interacting directly with the displayed diagram using their mouse, either to edit the existing diagram or to draw new components. In either case, edits to one form (code or image) will be immediately reflected in the other. The new techniques needed to make such a bidirectional user interface work should also be applicable in domains other than drawing.

Finally, I am always on the lookout for other interesting domains which could benefit from this kind of approach, and especially for opportunities to collaborate with experts in those domains.
Combinatorics, computation, and data structures

I am keenly interested in the intersection of combinatorics and computation. In particular, taking results in pure combinatorics and “porting” them to a computational setting yields both practical programming tools and new insight into the underlying mathematics.

My dissertation focuses on the theory of combinatorial species, developed over the past three decades as a unifying framework for enumerative combinatorics. As a purely combinatorial theory, species are relatively well-explored, but there are striking connections to the theory of algebraic data types which remain largely unexplored and unexploited. The core of my dissertation is to port the theory of species to a constructive type theory—specifically, the recently developed homotopy type theory—laying the groundwork to use the theory of species as a foundational basis for data structures in programming languages. One of the theory’s great strengths lies in a coherent, precise description of the relationship between labelled and unlabelled structures, so this results in a theory of labelled data structures which unifies algebraic data types (such as lists or binary trees) and what one would more typically think of as labelled structures (vectors, arrays, finite maps, . . . ), leading to richer expressivity and a conceptual framework for understanding existing algorithms and data structures in new ways. For example, the theory gives an abstract yet precise way to think about issues of memory layout and allocation in matrix algorithms.

Future directions

Another strength of the theory of species is its ability to describe structures with non-trivial symmetries, such as cycles or bags. Computationally, these correspond to data structures where one “abstract” value can have multiple concrete representations which the programmer should not be able to distinguish. Such abstract data types occur frequently, but are typically implemented in an ad-hoc way, with no help from the compiler in ensuring that the implementation is correct or that the abstract interface does not leak representation details. Species may afford a framework for reasoning about such abstract data structures, though the details are far from obvious.

Yet another key feature of the theory of species is its strong connection to the theory of generating functions, which can be used to summarize and extract information of interest about species. This is a rich seam of material with plenty of combinatorics left to “mine” for computational significance. In particular, I suspect that many algorithms of interest over data structures (enumeration, random generation, serialization, deserialization . . . ) can be reformulated using the framework of generating functions, leading to new insights and new algorithms.

References

