Research Statement

Daniel R. Licata

I study programming languages, with a focus on applications of type theory and logic. I have

- published papers in major conferences in my area (POPL 2012, MFPS 2011, ICFP 2009 and 2010, LICS 2008), as well as in refereed workshops.
- won the 2012 FoLLI E.W. Beth Dissertation Award for my thesis work on domain-specific logics for specifying and verifying software.
- worked on a new and exciting research topic, called homotopy type theory, an interdisciplinary project between mathematicians and computer scientists.
- advised three undergraduate students on significant projects. Two continued to the PhD program in computer science at Carnegie Mellon, and one is applying to graduate programs this fall.
- co-written two NSF grants, which partially funded my dissertation work and post-doc.

Type theory is a foundational language for programming and mathematics. It is the basis for the automated type systems of modern programming languages, which are effective and scalable lightweight formal methods. It is also the foundation of *proof assistants*, tools for computer-assisted mathematical proofs and full-scale software verification. Recent research has revealed an extremely surprising connection between type theory and the mathematical disciplines of category theory and homotopy theory. During my PhD and post-doc work, I have worked on *homotopy type theory*, a new type theory based on this correspondence; this year, I am a member of the Institute for Advanced Study (IAS), where more than 40 computer scientists and mathematicians have gathered for a year-long special program on this subject. These developments are the most exciting advances in type theory since its invention, with fruitful flows of ideas in both directions. The mathematics explains aspects of type theory that have puzzled researchers for decades, and suggests new type-theoretic principles for programming and proving. Type theory, extended with these new principles, can be used in novel ways for computer-assisted mathematical proofs, and provides new methods for approaching open problems in mathematics. Fields medalist Vladimir Voevodsky, who is organizing the year at IAS, believes that our work will be a turning point in the adoption of proof assistants by working mathematicians.

I plan to continue work on homotopy type theory as an assistant professor, and I firmly believe that this line of inquiry will be a strong component of a computer science department's research profile. First, it is an opportunity for computer science to make a significant interdisciplinary contribution to the practice of mathematics and theoretical physics, just as we currently reach out to other sciences, such as biology. Second, these ideas will also have applications to traditional computer science problems, such as programming and software verification (MFPS 2011). Third, several of the crucial open problems in homotopy type theory are core programming languages questions. For example, it is presently unclear how to actually run a program that uses the new homotopy-inspired principles, or how to provide useful automated theorem proving for the theory. I published the first programming interpretation for a subset of homotopy type theory, by connecting it with the programming languages idea of *generic programming* (POPL 2012). Thus, though many of the motivating applications are on the mathematics side, there are computer science applications as well, and much of the work to be done is core computer science.

In this statement, I will describe (1) the basics of homotopy type theory, (2) applications in mathematics, (3) applications in programming/verification, and (4) my research goals for the next several years.

1 Homotopy Type Theory

Type theory is the basis of the automated type systems of modern programming languages, which rule out an important class of failures and security flaws, by enforcing both language-level and application-level abstractions. Moreover, Martin-Löf's *dependent type theory* is the basis for many modern proof assistants, which are tools for computer-assisted mathematics. Proof assistants allow *computer-checked proofs*—a mathematician can represent a mathematical argument in such a way that the proof assistant can automatically verify its correctness—and more sophisticated interactive theorem proving. This increases confidence in mathematical results—and makes reviewing papers much easier!—and in many cases makes proofs easier to develop, as long as the benefits of automation are not outweighed by the costs of being fully formal. Proof assistants have recently been used to formalize difficult mathematical theorems: the Four-Color theorem about map coloring and the Feit-Thompson theorem in group theory have been fully checked, and significant progress has been made on Hales's proof of the Kepler Conjecture. These tools are also useful for computer-assisted software verification. For example

$$\mathsf{mergesort} : \forall(l : \mathsf{int\ list}).\exists(l' : \mathsf{int\ list}).\mathsf{Sorted}(l')$$

expresses that the function mergesort takes a list l as an argument, and returns a list l' as a result—and that, moreover, the result l' is sorted. Programmers can balance the costs and benefits of verification by choosing what properties to check—one might stop at verifying that l' is sorted, or go on to prove that it is a permutation of l. These tools scale to full-scale program verification, as shown, for example, by Leroy's recent CompCert project, fully verifying the correctness of a C compiler.

Next, I will describe how homotopy type theory extends existing type theories. The starting point of type theory is assertions of the form $e : \tau$, where e is a program and τ is a type (read this as "e has type τ" or "e is a member of τ"). In programming terms, such an assertion is a compile-time prediction about the run-time behavior of e. For example, $e : \mathsf{bool}$ means that e will compute to true or false, while $e : \mathsf{bool} \to \mathsf{bool}$ means that e will compute to a function, that, whenever you apply it to a boolean, will return a boolean. On the mathematical side, these programs are used for constructive reasoning.

One notion that comes up immediately in mathematical proof and software verification is *equality*—when are two programs equal? A key idea in type theory, especially the NuPRL type theory developed by Constable and collaborators, is that *each type comes with a notion of equality for its members*:

- For basic data types, such as booleans, integers, and lists of integers, programs are equal if they evaluate to structurally equal results. For example, programs e and e' of type int list are equal if they both evaluate to the same list value, say $[1, 2, 3, 4, 5]$.
- For function types, the right notion is *extensional equality*: two functions f and g are equal if they return the same result on all arguments. Similarly, for potentially-infinite interactive processes, such as a Web server, the right notion of equality is *bisimulation* (a relation between states that is preserved by transitions), as studied in model checking. Using these definitions of equality, one can verify a piece of code by proving that it is equal to (behaves the same as) a specification or reference implementation.
- Using *quotient types*, programmers can equip a type with a problem-specific notion of equality. For example, when working with a type binary_search_tree of binary search trees, it is profitable to equate different tree layouts that represent the same dictionary, so that trees that differ by order-preserving rotations are considered equal.

The force of these definitions comes from the fact that equality has a special status in type theory:

all type-theoretic constructions respect equality. This means, first, that a function $f : \tau \to \sigma$ must take equal arguments in τ to equal results in σ. For example, this captures the invariant that a function lookup of type $(\mathsf{binary\_search\_tree} \times \mathsf{key}) \to \mathsf{value}$ must return the same value when applied to dictionaries that differ by a tree rotation. Another manifestation is that, if a predicate $P(e)$ is true, then $P(e')$ must also be true whenever e is equal to e'. For example, if a reference implementation of a process satisfies a mathematical specification (it doesn't crash, it never leaks sensitive information, ...), and the actual implementation is bisimilar to the reference implementation, then the actual implementation satisfies the specification as well.

With this background in hand, we can discuss the key new insight in homotopy type theory:

type theory is compatible with computationally relevant notions of equality

Rather than thinking of equality $e = e'$ as a mere proposition, we think of it constructively as a full-fledged type, where a program p of type $(e = e')$ computes *evidence* that the programs e and e' are equal. This evidence p itself can fruitfully be used in other programs. Combined with the fact that all type-theoretic constructions respect equality, this notion of computationally relevant equalities is a very powerful principle, with many applications.

2 Applications in Mathematics

Homotopy type theory brings formal mathematics much closer to informal practice, reducing the costs of computer-assisted proofs, and suggests new type-theoretic methods for mathematics.

As an example of the first point, we can *equate isomorphic structures*. Often, there are two or more equivalent definitions of a mathematical concept—for example, real numbers can be defined either as Dedekind cuts or Cauchy sequences. Moreover, a single proof will usually rely on different definitions at different points, because one definition or another may be more useful for certain arguments. In informal mathematical practice, it is common to prove that the definitions are equivalent once, and then tacitly convert back and forth between them without comment. Homotopy type theory allows mathematicians to mimic this informal practice when writing fully formal computer-checked proofs, by taking isomorphic types to be equal. Formally, this is accomplished by taking isomorphisms[1] as computationally relevant evidence for type equality. Because of the general principle that all constructions respect equality, this results in a formal theory in which equivalent definitions can be freely interchanged, as in informal practice.

Voevodsky's univalence axiom generalizes this application, replacing isomorphism with a homotopy-inspired notion of equivalence of types. This generalization leads to exceedingly clean formalizations of category theory and homotopy theory. For example, the central notions of category theory can be defined in such a way that common informal conventions about what should be equated (isomorphic objects, equivalent categories, naturally isomorphic functors) become formal equalities in a proof assistant.

As an example of the second point, we can do homotopy theory in a type theoretic way. *Higher-dimensional inductive definitions* (a generalization of quotient types to computationally relevant equalities) enable direct, combinatorial definitions of topological spaces (up to homotopy) inside a proof assistant. For example the circle is defined to be an inductive type with a member $(\mathsf{base} : \mathsf{Circle})$ and a loop, which is represented by computationally relevant evidence for an equality $(\mathsf{loop} : \mathsf{base} = \mathsf{base})$. Then, we can apply type theoretic methods to problems in homotopy theory. A first example of this kind is the fact that the fundamental group of the circle is the integers. After reading a mathematician's proof of this, I developed a much simpler proof by applying some type theoretic principles (a paper on this result is in preparation). These ideas may be applicable to open problems, such as computing the higher homotopy groups of spheres.

[1] An isomorphism between two types consists of functions back and forth between the two types whose composites are the identity—this is essentially the same idea as a bijection of sets.

Moreover, these techniques are simple enough that undergraduates can get involved in this research: this past summer, I advised an undergraduate student who learned the proof assistant and made significant progress on a proof that the torus is homotopy-equivalent to the product of two circles.

Generalizing from the specifics of algebraic topology, there has recently been an enormous amount of activity in higher-dimensional category theory, which is bearing fruit in many different fields of mathematics and theoretical physics. Homotopy type theory is the *internal language* of certain structures in higher-dimensional category theory (∞-toposes). Mathematicians using these structures have found that homotopy type theory is is often a better way to work, even informally, than any previously-known techniques.

3 Applications in Programming

After learning about Voevodsky's univalence axiom, I began to investigate what these new homotopy-theoretic principles mean for programming. Homotopy type theory was designed for mathematical (rather than computational) goals, and whether and how one can execute a program that uses homotopy-inspired principles is currently an open problem. I have solved a special case of this question, for a restricted subset of homotopy type theory (POPL 2012). This is an important question to answer for several reasons: Philosophically, it justifies the constructivity of homotopy type theory. Practically, a programming interpretation allows programs to be extracted from proofs, facilitates proof automation, and enables computer science applications of these principles. In this section, I will describe the programming interpretation of homotopy type theory that I have developed so far, as well as some programming/verification applications of it.

The essence of the programming interpretation is the computational meaning of the aforementioned principle that all type-theoretic constructions respect equality. For example, when p is evidence for an equality $(e = e')$, the principle that a predicate C respects equality—that $C(e)$ implies $C(e')$—is witnessed by a function $\mathsf{transport}_C(p) : C(e) \to C(e')$. However, the traditional evaluation rules for transport in type theory do not account for situations when p is computationally relevant—e.g. if it is an isomorphism. I have solved this problem, in a special case, by viewing transport and other similar operations as *generic programs*: All of the types in type theory have certain programs (transport and a few others) associated with them, but these programs have gone unnoticed until now, because it is only in the presence of computationally relevant equalities that we can put them to work. In this case, each predicate C is equipped with a function $\mathsf{transport}_C$, where the evaluation behavior of this function is different for different predicates C.

What can one do with these generic programs? One application that I have investigated is *code reuse*. Many programming problems involve equipping a type with some operations. For example, a dictionary module consists of a type dict equipped with operations such as

$$\mathsf{insert} : \mathsf{dict} * (\mathsf{key} * \mathsf{value}) \to \mathsf{dict}$$
$$\mathsf{lookup} : \mathsf{dict} * \mathsf{key} \to \mathsf{value}$$

Consider two isomorphic types, such as key-value association lists (sorted by key) and balanced binary trees. If a programmer implements the dictionary operations for one of these types, then transport provides a way to automatically implement the dictionary operations for the other. The reason is that the presence of the dictionary operations (insert, lookup) can be seen as a predicate on a type dict, all predicates respect equality, and equality of types is isomorphism. Thus, the principle that isomorphic types are always interchangable, described above as a way to simplify mathematical proofs, *writes code for you*—it takes an implementation of dictionaries as binary search trees, say, and creates one as association lists.

Now, there is no magic: the isomorphism provides conversion functions between search trees and association lists, and transport computes by inserting these conversions in the all necessary places (insert on

association lists will convert to a search tree, do the insertion there, and then convert back). However, in some cases, particularly isomorphic types arising from using *type refinements* to track data structure invariants, automatable optimizations can simplify the code derived from `transport` to the efficient code one would have written out by hand. Additionally, I realized that the same ideas can be used to give simple *specifications for abstract types*: `transport` enables a very concise description of how an efficient implementation corresponds to a reference implementation, so that one can reason in terms of the reference implementation but run the efficient code. In this case, efficiency of `transport` is not an issue, because the conversions are done only in specifications/proofs, not in the actual running code.

Domain-Specific Logics. Finally, I will describe the application that motivated my study of homotopy type theory. This research is from my dissertation, which won the FoLLI E.W. Beth Dissertation award.

Recently, there has been a proliferation of different logics for reasoning about different styles of programs: Hoare logic and separation logic are used to reason about imperative programs. Temporal logics are used to reason about the evolution of concurrent processes. Authorization logics are used to verify security properties. Differential dynamic logic is used to reason about hybrid (discrete/continuous) cyber-physical systems. In my thesis work, I investigated ways to make it easier to use these domain-specific logics to reason about software. The goal is to give programmers the tools to develop and verify programs with logics, within a dependently typed programming language. Common tasks in this style of software verification include representing the syntax of a logic's propositions and proof rules, giving logical specifications to programs, proving theorems about a logic, and implementing theorem provers. A language can facilitate these tasks by providing two different notions of functions, *derivability functions*, which provide the notion of entailment common to many logics, and *admissibility functions*, which allow recursive functions over syntax and proofs (e.g. for implementing a theorem prover). However, these concepts are somewhat at odds with each other, because admissibility functions can ruin the very properties of entailment that derivability functions are supposed to provide. For this reason, some existing proof assistants provide good support for derivability, while others provide good support for admissibility, but none provide adequate support for both.

In my thesis work, I investigated to what extent these concepts can be mixed in richer ways, to support notions such as side conditions on inference rules. I handled the problematic interactions of derivability and admissibility by relaxing the properties that are required *in all cases* of derivability functions, while using generic programming techniques to recover the desired properties in many specific instances. My first attempt at this (LICS 2008) exploited logical techniques called polarity and focusing, which justify a programming language with some support for generic programming. My second (ICFP 2009) used dependently typed programming, which also provides support for generic programming. However, these approaches worked only for the syntax of propositions, not for inference rules and logical derivations.

As I was attempting to solve the full problem, Voevodsky visited CMU and spoke about homotopy type theory. After investigating the programming interpretation of homotopy type theory, I realized that the generic programs I needed were very similar to the ones in homotopy type theory, except for one issue: In homotopy type theory, we consider computationally relevant equalities $p : e = e'$. To capture the relevant properties of logical entailment, it is necessary to generalize this, equipping each type with a notion of asymmetric *transformations*, $p : e \leq e'$, which means that the program e can always be replaced with e', but not necessarily vice versa. I designed a *directed dependent type theory* based on these ideas and applied it to this problem (MFPS 2011), showing that directed type theory provides a useful framework for understanding the combination of derivability and admissibility. Moreover, I believe that directed type theory will have applications beyond this particular motivating example, and that many familiar programming languages ideas, such as subtyping and relational parametricity, can usefully be cast in this framework.

4 Research Goals

During the course of my career, I would like to do programming languages research that spans theory and practice. The research I have described in this statement is quite far on theoretical end of the spectrum. On the more practical side, I want to make it easier for programmers to write elegant, correct, fast, and evolvable code. Some of my past research has been more immediately application-focused: During an internship at Microsoft Research Cambridge with Simon Peyton Jones, I designed and implemented a new feature called *view patterns* in the GHC Haskell compiler. View patterns extend Haskell's pattern matching syntax by allowing function calls inside patterns; they are particularly useful for pattern-matching on abstract types. I worked with an undergraduate researcher to implement a type system for security within a dependently typed programming language (ICFP 2009). I simplified and generalized the ML5 language for distributed computing, and in the process adapted it to a Haskell-like setting with monadic effects (LFMTP 2010); this work was later used by other researchers to extract Erlang programs that provably manage distributed resources correctly. As an undergrad, I worked on software engineering (ASE 2003) and model checking (ASE 2004). I have taught parallel functional programming, and believe that after some implementation work, it will be an extremely effective method for programming multicore machines and clusters. I am excited by the recent increased use of functional programming in industry, especially in finance, and the proliferation of functional programming ideas in industry-friendly languages such as Scala and C#/F#.

That said, I plan to focus on homotopy type theory in the immediate future, because I believe it is the kind of fundamental theoretical research that will eventually lead to significant practical benefits, even if we do not yet know exactly what they are. Whenever the same ideas are discovered independently in two very different fields of inquiry, it is worth taking notice. This work has the potential to change mathematics: if successful, it will make computer-assisted proofs more practical, which will eventually increase the complexity of results that mathematicians can achieve; and, through the connection with higher category theory, it may lead to new techniques and results in wide areas of mathematics and in physics. On the programming side, though homotopy type theory is in a very preliminary state, my thoughts about code reuse, specifications for abstract types, and domain specific logics suggest that the generic programs inherent in homotopy type theory will have practical applications.

Over the next several years, my first goal is to pursue a programming language interpretation for all of homotopy type theory, generalizing my previous work. This work is part of a larger project, with my collaborators at IAS this year, to make a proof assistant for homotopy type theory—the programming interpretation will be the basis for proof automation. With this technology in place, I plan to continue collaborating with mathematicians to develop applications on the math side, and to investigate applications on the programming side. I expect that, within the next five or so years, I will have developed significant results that demonstrate the usefulness of these ideas.