A research career and the role of proofs in Mathematics and Computer Science

Paige Randall North

Utrecht University

24 February 2021

Outline

My experience and advice

Proofs in computer science research

Outline

My experience and advice

Proofs in computer science research

Short CV

- BS in Mathematics from University of Chicago (homotopy theory)

Short CV

- BS in Mathematics from University of Chicago (homotopy theory)
- PhD in Mathematics from University of Cambridge (categorical logic)

Short CV

- BS in Mathematics from University of Chicago (homotopy theory)
- PhD in Mathematics from University of Cambridge (categorical logic)
- Postdoc in Mathematics at Ohio State (applied topology)

Short CV

- BS in Mathematics from University of Chicago (homotopy theory)
- PhD in Mathematics from University of Cambridge (categorical logic)
- Postdoc in Mathematics at Ohio State (applied topology)
- Honorary researcher in Computer Science at University of Birmingham (theory)

Short CV

- BS in Mathematics from University of Chicago (homotopy theory)
- PhD in Mathematics from University of Cambridge (categorical logic)
- Postdoc in Mathematics at Ohio State (applied topology)
- Honorary researcher in Computer Science at University of Birmingham (theory)
- Postdoc in Mathematics and Electrical and Systems Engineering at University of Pennsylvania (applied topology)

Short CV

- BS in Mathematics from University of Chicago (homotopy theory)
- PhD in Mathematics from University of Cambridge (categorical logic)
- Postdoc in Mathematics at Ohio State (applied topology)
- Honorary researcher in Computer Science at University of Birmingham (theory)
- Postdoc in Mathematics and Electrical and Systems Engineering at University of Pennsylvania (applied topology)
- Assistant professor in Mathematics and Computer Science at Utrecht (? and software technology)

Short CV

- BS in Mathematics from University of Chicago (homotopy theory)
- PhD in Mathematics from University of Cambridge (categorical logic)
- Postdoc in Mathematics at Ohio State (applied topology)
- Honorary researcher in Computer Science at University of Birmingham (theory)
- Postdoc in Mathematics and Electrical and Systems Engineering at University of Pennsylvania (applied topology)
- Assistant professor in Mathematics and Computer Science at Utrecht (? and software technology)
- Honorary researcher in Computer Science at TUDelft (programming languages)

Lesson from my career

- Every job (including PhD) I got was through my ‘network’.
  - What you can do (and who knows what you can do) is *much* more important than grades.
- Take every opportunity to meet peers and form close-knit groups.
  - You will learn more than from taking classes or reading papers alone.
- (In my experience.)

My interests

- Understanding the “true nature of things”
- Using mathematics – science of knowing things absolutely
- Interdisciplinarity
- Computer science – artificial and “small” so we can understand it completely

My interests

- Understanding the "true nature of things"
- Using mathematics – science of knowing things absolutely
- Interdisciplinarity
- Computer science – artificial and "small" so we can understand it completely

- So I study computer science using mathematics.
- More generally, the time for interdisciplinarity has come.

Mathematics and computer science

- Several points of intersection:
  - (Functional) programming language theory (denotational semantics)
  - Models of computation (operational semantics)
  - Machine learning & quantum computation
  - Complexity and computability theory
  - Numerical analysis
  - Computer algebra
  - ...
- We focus on (functional) programming language theory

(Functional) programming language theory

- Studying mathematical (denotational) semantics of programming languages

(Functional) programming language theory

- Studying mathematical (denotational) semantics of programming languages
- Creating new programming languages

(Functional) programming language theory

- Studying mathematical (denotational) semantics of programming languages
- Creating new programming languages
- Doing new things in old programming languages

(Functional) programming language theory

- Studying mathematical (denotational) semantics of programming languages
- Creating new programming languages
- Doing new things in old programming languages
- Implementing new programming languages

(Functional) programming language theory

- Studying mathematical (denotational) semantics of programming languages
- Creating new programming languages
- Doing new things in old programming languages
- Implementing new programming languages
- Certifying correctness of programs

Culture in mathematics and computer science

Mathematics	Computer science
Fewer papers	More papers
Big ideas	Incremental
Rigorous	Ideas
Long papers	Short papers
Journals papers	Conference papers
Workshops	Conferences
Seminar culture	Coworking culture
Fewer coauthors	More coauthors
Fewer grants	More grants

- Ideal strategy:
  - publish conference (CS) paper that emphasizes ideas first
  - expanded journal (math) paper that emphasizes proofs second

Life in academia beyond masters

Aim for a research career if

- there is a topic you are obsessed with
  - Don't become too obsessed
- you find science communication fulfilling
- you want to spend a lot of time in other countries and have most of your network in other countries
- (you want to do it all)

Being a minority in math and cs

- Experienced nothing overt
- It is much more difficult to form a network of trusted, close colleagues
  - Work on becoming very close to one or a few people, who can connect you to others
- Minorities are much more likely to be invited to give talks, etc
  - Good and bad

Outline

My experience and advice

Proofs in computer science research

Curry-Howard correspondence

Statement

Proofs are programs.

Statements are program specifications.

Curry-Howard correspondence

Statement

Proofs are programs.

Statements are program specifications.

(In constructive mathematics.)

Curry-Howard correspondence

Example

Statement: For every natural number n, there is another number p which is prime and greater than n.

Proof: ...

Types

Haskell

We have types and type formers:

- $\mathbb{N}$
- $\mathbb{L}\mathsf{ist}(A)$
- $A \times B$
- $A \to B$

Types

Haskell

We have types and type formers:

- $\mathbb{N}$
- $\mathbb{L}\text{ist}(A)$
- $A \times B$
- $A \to B$

You can prove:

- There is an addition $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Types

Haskell

We have types and type formers:

- $\mathbb{N}$
- $\mathbb{L}\mathsf{ist}(A)$
- $A \times B$
- $A \to B$

You can prove:

- There is an addition $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

You can't prove:

- This addition is associative.
- You can only prove that externally.

Software verification

There are two main ways to check that a program behaves correctly:

- test it
- prove that is adheres to a specification

Software verification

There are two main ways to check that a program behaves correctly:

- test it
- prove that is adheres to a specification
    - externally
        - Hoare logic
        - model checking
    - internally
        - type theory: a programming language with extra features so that you can prove things

Software verification

Why do we want to prove correctness internally?

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
- The computer checks correctness of the *proof*

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
- The computer checks correctness of the *proof*

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
- The computer checks correctness of the *proof*

Pitfalls

- The specification has to be ‘correct’

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
- The computer checks correctness of the *proof*

Pitfalls

- The specification has to be ‘correct’
- In practice (industry), two programs are sometimes created: an efficient one and a correct one

Verification

What do we need to prove correctness (or anything)?

Verification

What do we need to prove correctness (or anything)?

Consider the statement $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

- We can construct a value $+ :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Verification

What do we need to prove correctness (or anything)?

Consider the statement $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

- We can construct a value $+ :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Consider the statement $a + b = b + a$

- We want to construct a $p :: a + b = b + a$

Verification

What do we need to prove correctness (or anything)?

Consider the statement $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

- We can construct a value $+ :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Consider the statement $a + b = b + a$

- We want to construct a $p :: a + b = b + a$
- First, we need a *equality type* that takes two values x, y of the same type A and creates a new type $x = y$

Verification

What do we need to prove correctness (or anything)?

Consider the statement $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

- We can construct a value $+ :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Consider the statement $a + b = b + a$

- We want to construct a $p :: a + b = b + a$
- First, we need a *equality type* that takes two values x, y of the same type A and creates a new type $x = y$

Verification

What do we need to prove correctness (or anything)?

Consider the statement $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

- We can construct a value $+ :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Consider the statement $a + b = b + a$

- We want to construct a $p :: a + b = b + a$
- First, we need a *equality type* that takes two values x, y of the same type A and creates a new type $x = y$

Actually we want to prove this for all $a, b :: \mathbb{N}$

- We need something like $\forall_{a,b::\mathbb{N}}\ a + b = b + a$

Verification

What do we need to prove correctness (or anything)?

Consider the statement $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

- We can construct a value $+ :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Consider the statement $a + b = b + a$

- We want to construct a $p :: a + b = b + a$
- First, we need a *equality type* that takes two values x, y of the same type A and creates a new type $x = y$

Actually we want to prove this for all $a, b :: \mathbb{N}$

- We need something like $\forall_{a,b::\mathbb{N}}\ a + b = b + a$
- We use $\Pi_{a,b::\mathbb{N}}\ a + b = b + a$

Dependent types

We want $\Pi_{(a,b)::\mathbb{N}\times\mathbb{N}}\ a+b=b+a$.

- We have a type $a+b=b+a$ that *depends on* $(a,b)::\mathbb{N}\times\mathbb{N}$

Dependent types

We want $\Pi_{(a,b)::\mathbb{N}\times\mathbb{N}}\ a + b = b + a$.

- We have a type $a + b = b + a$ that *depends on* $(a, b) :: \mathbb{N} \times \mathbb{N}$
- We call a type that depends on values of another type a *dependent type*

Dependent types

We want $\Pi_{(a,b)::\mathbb{N}\times\mathbb{N}}\ a + b = b + a$.

- We have a type $a + b = b + a$ that *depends on* $(a, b) :: \mathbb{N} \times \mathbb{N}$
- We call a type that depends on values of another type a *dependent type*
- Now Π ('for all') is a type former that takes a dependent type like $a + b = b + a$ and produces a new type $\Pi_{(a,b)::\mathbb{N}\times\mathbb{N}}\ a + b = b + a$

Dependent types

We want $\Pi_{(a,b)::\mathbb{N}\times\mathbb{N}}\ a + b = b + a$.

- We have a type $a + b = b + a$ that *depends on* $(a, b) :: \mathbb{N} \times \mathbb{N}$
- We call a type that depends on values of another type a *dependent type*
- Now Π ('for all') is a type former that takes a dependent type like $a + b = b + a$ and produces a new type $\Pi_{(a,b)::\mathbb{N}\times\mathbb{N}}\ a + b = b + a$
- It is the type of functions that take in a value $(a, b) :: \mathbb{N} \times \mathbb{N}$ and return some value in $a + b = b + a$

Dependent types

We want $\Pi_{(a,b)::\mathbb{N}\times\mathbb{N}}\ a+b=b+a$.

- We have a type $a+b=b+a$ that *depends on* $(a,b)::\mathbb{N}\times\mathbb{N}$
- We call a type that depends on values of another type a *dependent type*
- Now Π ('for all') is a type former that takes a dependent type like $a+b=b+a$ and produces a new type $\Pi_{(a,b)::\mathbb{N}\times\mathbb{N}}\ a+b=b+a$
- It is the type of functions that take in a value $(a,b)::\mathbb{N}\times\mathbb{N}$ and return some value in $a+b=b+a$
- We call this a type of 'dependent functions'

Dependent pairs

Now consider the statement:

For every natural number n, there is another number p which is prime and greater than n.

- $isPrime(p) := p > 1 \times \Pi_{x,y:\mathbb{N}}\ (xy = p) \to (x \leqslant y) \to (x = 1)$
- Now we can write $\Pi_{n::\mathbb{N}}$ there is $p :: \mathbb{N}$ such that $isPrime(p) \times (p > n)$
- Again $isPrime(p) \times (p > n)$ depends on p (and n).
- Is $\Pi_{p::\mathbb{N}} isPrime(p) \times (p > n)$ what we mean?
- Instead we want $\exists_{p::\mathbb{N}} isPrime(p) \times (p > n)$
- We write $\Sigma_{p::\mathbb{N}} isPrime(p) \times (p > n)$
- It is the type of pairs of a $p :: \mathbb{N}$ and a $q :: isPrime(p) \times (p > n)$
- Now we can write $\Pi_{n::\mathbb{N}} \Sigma_{p::\mathbb{N}} isPrime(p) \times (p > n)$
- A value of this type is a program that produce a p from an n together with a proof that it is prime and bigger than n

Role of this verification

- Used somewhat widely in industry
  - Mostly on very critical and fundamental software/hardware
  - Supported by governments/militaries or in research groups
  - Very costly
- Automated theorem proving helps
- Gaining importance in mathematics

Current research

What does current research look like?

- Introducing the identity type creates interesting behavior $\rightarrow$ homotopy type theory
- Domain specific languages (logics) that incorporate specific reasoning principles
- Establish that these languages can be implemented (normalization proofs or algorithms)
- Papers in this field are usually 'math papers' (i.e. Definition - Theorem - Proof) but are often accompanied by code - the formalized proofs

Thank you!