The role of proofs in computer science research

Paige Randall North

Utrecht University

25 February 2025

Curry-Howard correspondence

(Constructive) mathematics	$\leftrightarrow$	(Functional) programming
Proofs	$\leftrightarrow$	Programs
Statements	$\leftrightarrow$	Program specifications

Curry-Howard correspondence

Example

Statement: For every natural number n, there is another natural 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:

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

You can program/prove (in Haskell):

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

Types

Haskell

We have types and type formers:

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

You can program/prove (in Haskell):

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

You can't program/prove (in Haskell):

- This addition is associative.

Types

Haskell

We have types and type formers:

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

You can program/prove (in Haskell):

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

You can't program/prove (in Haskell):

- 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 (*formal verification*)

Software verification

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

- test it
- prove that is adheres to a specification (*formal verification*)
    - 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
  - We produce a value of the type that represents the specification

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
  - We produce a value of the type that represents the specification
- The computer (type checker) checks correctness

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
  - We produce a value of the type that represents the specification
- The computer (type checker) checks correctness

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
  - We produce a value of the type that represents the specification
- The computer (type checker) checks correctness

Pitfalls

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
  - We produce a value of the type that represents the specification
- The computer (type checker) checks correctness

Pitfalls

- The specification has to be ‘correct’

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
  - We produce a value of the type that represents the specification
- The computer (type checker) checks correctness

Pitfalls

- The specification has to be 'correct'
- The correct-by-construction program is often not efficient

Software verification

Why do we want to prove correctness internally?

- Correct-by-construction
  - We produce a value of the type that represents the specification
- The computer (type checker) checks correctness

Pitfalls

- The specification has to be 'correct'
- The correct-by-construction program is often not efficient
  - 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 specification/statement $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

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

Verification

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

Consider the specification/statement $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$.

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

Consider the specification/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 specification/statement $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

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

Consider the specification/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 specification/statement $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

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

Consider the specification/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 specification/statement $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

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

Consider the specification/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 specification/statement $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

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

Consider the specification/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 introduce another type former $\Pi_{a,b::\mathbb{N}}\ a + b = b + a$

Dependent types

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

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

Dependent types

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

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

Dependent types

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

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

$$\Pi_{a,b::\mathbb{N}}\ a + b = b + a$$

Dependent types

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

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

$$\Pi_{a,b::\mathbb{N}}\ a + b = b + a$$

- It is the type of functions that take in values $a, b :: \mathbb{N}$ and return some value in $a + b = b + a$, so in some languages you write

$$(a, b :: \mathbb{N}) \to\ a + b = b + a$$

Dependent types

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

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

$$\Pi_{a,b::\mathbb{N}}\ a+b=b+a$$

- It is the type of functions that take in values $a, b :: \mathbb{N}$ and return some value in $a+b=b+a$, so in some languages you write

$$(a, b :: \mathbb{N}) \to\ a+b=b+a$$

- 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.

- $\mathit{isPrime}(p) :=$
  $(p > 1) \times (\Pi_{x,y:\mathbb{N}}\ (xy = p) \to (x \leqslant y) \to (x = 1))$

Dependent pairs

Now consider the statement:

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

- $isPrime(p) :=$
  $(p > 1) \times (\Pi_{x,y:\mathbb{N}}\ (xy = p) \to (x \leqslant y) \to (x = 1))$
- $p > 1 :=$ there exists $n : \mathbb{N}$ such that $1 + succ(n) = p$

Dependent pairs

Now consider the statement:

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

- $isPrime(p) :=$
  $(p > 1) \times (\Pi_{x,y:\mathbb{N}}\ (xy = p) \to (x \leqslant y) \to (x = 1))$
- $p > 1 :=$ there exists $n : \mathbb{N}$ such that $1 + succ(n) = p$
- Again "$1 + succ(n) = p$" depends on $n :: \mathbb{N}$.

Dependent pairs

Now consider the statement:

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

- $isPrime(p) :=$
  $(p > 1) \times (\Pi_{x,y:\mathbb{N}}\ (xy = p) \to (x \leqslant y) \to (x = 1))$
- $p > 1 :=$ there exists $n : \mathbb{N}$ such that $1 + succ(n) = p$
- Again "$1 + succ(n) = p$" depends on $n :: \mathbb{N}$.
- We want something like $\exists_{n::\mathbb{N}}(1 + succ(n) = p)$

Dependent pairs

Now consider the statement:

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

- $isPrime(p) :=$
  $(p > 1) \times (\Pi_{x,y:\mathbb{N}} \ (xy = p) \to (x \leqslant y) \to (x = 1))$
- $p > 1 :=$ there exists $n : \mathbb{N}$ such that $1 + succ(n) = p$
- Again "$1 + succ(n) = p$" depends on $n :: \mathbb{N}$.
- We want something like $\exists_{n::\mathbb{N}}(1 + succ(n) = p)$
- We write $p > 1 := \Sigma_{n::\mathbb{N}}(1 + succ(n) = p)$

Dependent pairs

Now consider the statement:

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

- $isPrime(p) :=$
  $(p > 1) \times (\Pi_{x,y:\mathbb{N}}\ (xy = p) \to (x \leqslant y) \to (x = 1))$
- $p > 1 :=$ there exists $n : \mathbb{N}$ such that $1 + succ(n) = p$
- Again "$1 + succ(n) = p$" depends on $n :: \mathbb{N}$.
- We want something like $\exists_{n::\mathbb{N}}(1 + succ(n) = p)$
- We write $p > 1 := \Sigma_{n::\mathbb{N}}(1 + succ(n) = p)$
- It is the type of pairs of an $n :: \mathbb{N}$ and an $e :: 1 + succ(n) = p$

Dependent pairs

Now consider the statement:

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

- $isPrime(p) :=$
  $(p > 1) \times (\Pi_{x,y:\mathbb{N}}\ (xy = p) \to (x \leqslant y) \to (x = 1))$
- $p > 1 :=$ there exists $n : \mathbb{N}$ such that $1 + succ(n) = p$
- Again "$1 + succ(n) = p$" depends on $n :: \mathbb{N}$.
- We want something like $\exists_{n::\mathbb{N}}(1 + succ(n) = p)$
- We write $p > 1 := \Sigma_{n::\mathbb{N}}(1 + succ(n) = p)$
- It is the type of pairs of an $n :: \mathbb{N}$ and an $e :: 1 + succ(n) = p$
- Now we can write $\Pi_{n::\mathbb{N}}\ \Sigma_{p::\mathbb{N}}\ isPrime(p) \times (p > n)$

Dependent pairs

Now consider the statement:

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

- $isPrime(p) :=$
  $(p > 1) \times (\Pi_{x,y:\mathbb{N}}\ (xy = p) \to (x \leqslant y) \to (x = 1))$
- $p > 1 :=$ there exists $n : \mathbb{N}$ such that $1 + succ(n) = p$
- Again "$1 + succ(n) = p$" depends on $n :: \mathbb{N}$.
- We want something like $\exists_{n::\mathbb{N}}(1 + succ(n) = p)$
- We write $p > 1 := \Sigma_{n::\mathbb{N}}(1 + succ(n) = p)$
- It is the type of pairs of an $n :: \mathbb{N}$ and an $e :: 1 + succ(n) = p$
- 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 increasingly 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 → homotopy type theory
- Domain specific languages that verify specific programs/systems

- Implement these languages (normalization proofs/algorithms)
- Papers in this field are often 'math papers' (i.e. Definition - Theorem - Proof) but are often accompanied by code – the formalized proofs

Current research

What does current research look like?

- Introducing the identity type creates interesting behavior $\rightarrow$ homotopy type theory
- Domain specific languages that verify specific programs/systems
  - DSL for hybrid dynamical systems for verifying physical behavior of robots
- Implement these languages (normalization proofs/algorithms)
- Papers in this field are often 'math papers' (i.e. Definition - Theorem - Proof) but are often accompanied by code – the formalized proofs

Current research

What does current research look like?

- Introducing the identity type creates interesting behavior → homotopy type theory
- Domain specific languages that verify specific programs/systems
  - DSL for hybrid dynamical systems for verifying physical behavior of robots
  - DSL for directed homotopy theory for verifying behavior of concurrent processes

- Implement these languages (normalization proofs/algorithms)
- Papers in this field are often ‘math papers’ (i.e. Definition - Theorem - Proof) but are often accompanied by code – the formalized proofs

Current research

What does current research look like?

- Introducing the identity type creates interesting behavior → homotopy type theory
- Domain specific languages that verify specific programs/systems
  - DSL for hybrid dynamical systems for verifying physical behavior of robots
  - DSL for directed homotopy theory for verifying behavior of concurrent processes
  - DSL for fuzzy logic for verifying behavior of fuzzy control systems
- Implement these languages (normalization proofs/algorithms)
- Papers in this field are often ‘math papers’ (i.e. Definition - Theorem - Proof) but are often accompanied by code – the formalized proofs

Thank you!