Formal Verification and Coq

Beck, Calvin
hobbes@ualberta.ca

January 18, 2015

What is this Talk about?

Formal verification, mostly! This presentation hopes to address the following:

- Why should you care / be interested?
- Briefly cover some of the methods.
- Make proof assistants more accessible.
- Give some rough intuitions about how these systems work.

Type Signatures

`x` has type `A`:

```
x :: A
```

Type Signatures

`x` has type `A`:

```
x :: A
```

`x` is an integer, `y` is a list of integers.

```
x :: Integer
y :: [Integer]
```

Type Signatures Continued...

Functions have types too!

```
(+) :: Integer -> Integer -> Integer
```

Type Signatures Continued...

Types can also be polymorphic. The identity function, `id`, may take any type as an argument.

```
id :: a -> a
id x = x
```

Lambda Calculus

Lambda terms:

- Variables: "x" and such
- Lambda abstraction: $(\lambda x.t)$ where t is another lambda term. x is an argument, t is the "body"
- Application: (ts)

Combine as you see fit!

Beta Reduction

Beta reduction is just substitution.

$$\texttt{id} = (\lambda x.x)$$

Substituting x for t...

$$(\lambda x.x)t = t$$

Formal Verification: What it be?

The use of formal methods to prove that programs are correct

Formal Verification: What it be?

The use of formal methods to prove that programs are correct

- Want programs to be correct
  - Almost everything has a computer in it now
  - Incorrect programs can be dangerous
  - Bugs can be expensive

Formal Verification: What it be?

The use of formal methods to prove that programs are correct

- Want programs to be correct
  - Almost everything has a computer in it now
  - Incorrect programs can be dangerous
  - Bugs can be expensive
- Mathematicians want computers to verify their proofs as well.

Some Methods

- Checking by hand... Manual labor :(

Some Methods

- Checking by hand... Manual labor :(
- Model checking
  - Essentially checking every possible state of your program
  - “Proof by exhaustion”
  - This can be computationally expensive
  - Works best on small F.S.M.s.

Some Methods

- Checking by hand... Manual labor :(
- Model checking
  - Essentially checking every possible state of your program
  - “Proof by exhaustion”
  - This can be computationally expensive
  - Works best on small F.S.M.s.
- Type Checking
  - Types provide guarantees about how values behave
  - Most languages do this badly (Java, Python)
  - Some are good, but still limited (Haskell)

Some Methods

- Checking by hand... Manual labor :(
- Model checking
  - Essentially checking every possible state of your program
  - "Proof by exhaustion"
  - This can be computationally expensive
  - Works best on small F.S.M.s.
- Type Checking
  - Types provide guarantees about how values behave
  - Most languages do this badly (Java, Python)
  - Some are good, but still limited (Haskell)
- Theorem Proving
  - Mathematical proofs for great justice
  - Use the computer to check the proofs
  - This actually boils down to extended type checking

Levels of Abstraction

- High Level: Algorithms? Correct implementation?

Levels of Abstraction

- High Level: Algorithms? Correct implementation?
- Low Level: Check machine code?

Levels of Abstraction

- High Level: Algorithms? Correct implementation?
- Low Level: Check machine code?
- Hardware?

Levels of Abstraction

- High Level: Algorithms? Correct implementation?
- Low Level: Check machine code?
- Hardware?
- Mix and match!

Levels of Abstraction

- High Level: Algorithms? Correct implementation?
- Low Level: Check machine code?
- Hardware?
- Mix and match!

We'll focus on high level stuff!

The Coq Proof Assistant

We're going to be looking at one of the staples of the industry, called Coq. So named because of:

- Tradition of naming programming languages after animals (OCaml). Xavier Leroy created both OCaml and Coq

The Coq Proof Assistant

We're going to be looking at one of the staples of the industry, called Coq. So named because of:

- Tradition of naming programming languages after animals (OCaml). Xavier Leroy created both OCaml and Coq
- French. Xavier Leroy is from France

The Coq Proof Assistant

We're going to be looking at one of the staples of the industry, called Coq. So named because of:

- Tradition of naming programming languages after animals (OCaml). Xavier Leroy created both OCaml and Coq
- French. Xavier Leroy is from France
- CoC: Calculus of Constructions

The Coq Proof Assistant

We're going to be looking at one of the staples of the industry, called Coq. So named because of:

- Tradition of naming programming languages after animals (OCaml). Xavier Leroy created both OCaml and Coq
- French. Xavier Leroy is from France
- CoC: Calculus of Constructions
- Thierry Coquand is the creator of CoC

The Coq Proof Assistant

We're going to be looking at one of the staples of the industry, called Coq. So named because of:

- Tradition of naming programming languages after animals (OCaml). Xavier Leroy created both OCaml and Coq
- French. Xavier Leroy is from France
- CoC: Calculus of Constructions
- Thierry Coquand is the creator of CoC
- Basically the universe is trying to make this talk awkward

Examples

MOVING ON TO EXAMPLES!

Theory 'n Stuff

Coq is basically just a type-checker!

Theory 'n Stuff

- What does type checking have to do with proving theorems?

Theory 'n Stuff

- What does type checking have to do with proving theorems?
  - EVERYTHING due to the Curry-Howard isomorphism!

Theory 'n Stuff

- Curry-Howard isomorphism relates programs to proofs.
  - Specifically it relates terms of the simply-typed lambda calculus to intuitionistic logic.
  - Coq actually uses the "Calculus of Constructions". It's another lambda calculus, but has some special sauce which enable quantifiers and has some other nice properties.
  - Simply-typed lambda calculus still provides some good intuition, however.

Theory 'n Stuff

- Curry-Howard isomorphism relates programs to proofs.
  - Specifically it relates terms of the simply-typed lambda calculus to intuitionistic logic.
  - Coq actually uses the "Calculus of Constructions". It's another lambda calculus, but has some special sauce which enable quantifiers and has some other nice properties.
  - Simply-typed lambda calculus still provides some good intuition, however.
- Types are propositions. For instance, the type:

```
    a -> b
```

corresponds to the proposition $a \to b$. "a implies b".

Theory 'n Stuff

- Curry-Howard isomorphism relates programs to proofs.
  - Specifically it relates terms of the simply-typed lambda calculus to intuitionistic logic.
  - Coq actually uses the "Calculus of Constructions". It's another lambda calculus, but has some special sauce which enable quantifiers and has some other nice properties.
  - Simply-typed lambda calculus still provides some good intuition, however.
- Types are propositions. For instance, the type:

```
a -> b
```

corresponds to the proposition $a \to b$. "a implies b".

- A program inhabiting that type is an existence proof of the proposition.
  - Roughly speaking the program implements the proposition, so it demonstrates that the proposition is true.

Curry-Howard: A Brief Introduction

- Any type which is inhabited (has a value) represents a provable proposition

Curry-Howard: A Brief Introduction

- Any type which is inhabited (has a value) represents a provable proposition
- Implication is represented with "->" in types
  - If you have a value of the first type, then you can produce a value of the second type!

Curry-Howard: A Brief Introduction

- Any type which is inhabited (has a value) represents a provable proposition
- Implication is represented with "`->`" in types
  - If you have a value of the first type, then you can produce a value of the second type!
- Conjunction "$A \wedge B$" corresponds to a tuple "`(a, b)`"
  - Both `a` and `b` have to be inhabited in order for `(a, b)` to be inhabited.

Curry-Howard: A Brief Introduction

- Any type which is inhabited (has a value) represents a provable proposition
- Implication is represented with "`->`" in types
  - If you have a value of the first type, then you can produce a value of the second type!
- Conjunction "$A \wedge B$" corresponds to a tuple "`(a, b)`"
  - Both `a` and `b` have to be inhabited in order for `(a, b)` to be inhabited.
- Disjunction "$A \vee B$" corresponds to `Either a b`

```
data Either a b = Left a | Right b
```

If either `a` or `b` has a value then `Either a b` can have a value.

Curry-Howard: Not Quite Brief Enough for one Slide

- False is an uninhabited type. We call this type `Void`
  - If the type can't have a value, then it can not be "true".
  - Any false proposition is equivalent to `Void`, e.g., $a \rightarrow b$.

Curry-Howard: Not Quite Brief Enough for one Slide

- False is an uninhabited type. We call this type `Void`
  - If the type can't have a value, then it can not be "true".
  - Any false proposition is equivalent to `Void`, e.g., $a \rightarrow b$.
- Negation is given by `a -> Void`
  - If `a` is `Void` then it is inhabited by `id :: Void -> Void`
  - Otherwise `a -> Void` must be uninhabited, since a function must return a value when given a value.

The Problem of Non-termination

- If programs don't have to terminate every type is inhabited by an infinite loop!
  - Every proposition is true, and that's not useful at all!

Curry-Howard Example

Say we want to prove $A \to B \to A$.

- C-H claims we just need to provide a function with this type

Curry-Howard Example

Say we want to prove $A \rightarrow B \rightarrow A$.

- C-H claims we just need to provide a function with this type
- Note that A and B can be ANY type.

Curry-Howard Example

Say we want to prove $A \to B \to A$.

- C-H claims we just need to provide a function with this type
- Note that A and B can be ANY type.
- What could our function be?

```
??? :: a -> b -> a
```

Curry-Howard Example Continued...

How about the constant function?

```
const :: a -> b -> a
const a b = a
```

Curry-Howard Example Continued...

How about the constant function?

```
const :: a -> b -> a
const a b = a
```

- This actually makes sense!
  - Given a proof (value) of `a`, and a proof of `b`, we can provide a proof for a...
  - Just return the proof that was given to us!

The Return of the Lambda

- We can rewrite `const` as a lambda term in the simply typed lambda calculus:

$$(\lambda x : a, y : b \,.\, x)$$

The Return of the Lambda

- We can rewrite `const` as a lambda term in the simply typed lambda calculus:

$$(\lambda x : a, y : b \,.\, x)$$

- This is what Coq is doing behind the scenes...
  - The lambda calculi are really simple.
  - Verifying that the terms are valid is “easy”.
  - No matter how complicated the tactics are, which generate the lambda terms they must generate a lambda term which proves the proposition!
  - Tactic code can be complex and buggy. It doesn't affect the validity of the proofs.

Wrapping up!

- Proof assistants are useful for writing correct code!
  - Coq provides a means of extracting the code for use in other programming languages.
  - You can use it for small, but important parts of your code base.
  - Reasoning about code is so much easier when you have a system to help you!
- Useful for mathematics as well. Show the grader who's boss.
- Formal verification is a lot of effort, but Coq can provide tactics through Ltac which ease the burden.

References / Cool Stuff

- https://coq.inria.fr/
- http://proofgeneral.inf.ed.ac.uk/
- http://www.cis.upenn.edu/~bcpierce/sf/current/index.html
- http://adam.chlipala.net/cpdt/
- http://en.wikibooks.org/wiki/Haskell/The_Curry%E2%80%93Howard_isomorphism
- http://www.lix.polytechnique.fr/~barras/publi/coqincoq.pdf
- http://homotopytypetheory.org/book/