Program Verification and Certification

What is it?

Using the computer as a proof assistant to develop a formal proof of correctness for code.

Economic Demand

Wealthy and important corporations, as well as government, are being routinely embarrassed by hacking incidents. Secure computing and communications are at a premium.

+ Limited Supply

Most programming is focused on rapid development. Proof-checked programming is still primarily done by researchers, though this is changing.

= Fame and Fortune

Great demand and limited supply work to the advantage of the programmer.
In our present world of cyber-warfare, no other method can provide the level of defense against code errors as can machine-checked proofs.
Besides, the subject is a fascinating one.

Why Coq?

- The combination of dependent types and intuitionistic logic best suit its use for program checking.
- Coq has extensive and current documentation, making it suitable for independent study.
- It is possible to extract a program from a correct proof.
- See Certified Programming with Dependent Types for more reasons.

Some Other Proof Checkers

- ACL2
- Epigram
- F*
- Idris/Agda
- Isabelle
- PVS
- Twelf

Sub-Languages within Coq

- Gallina: programming
- CIC: Logic and types
- Vernacular: Tactic and declaration commands
- Ltac: Coding new tactics

While You are Learning

Becoming familiar with a functional language like OCaml, F# or Haskell should make it much easier to understand Coq (Gallina).

Using assertion checks is a step toward using types that are propositions.

It is most helpful to have an informal proof of correctness before attempting an automated proof.

Basic Ideas 1

- Statically typed functional language
- Dependent and decidable types
- Propositions as types
- Arrow notation for types;
  `->` `(` `)` `/\` `\/`
- Higher order logic

Basic Ideas 2

- Intuitionistic logic
- BHK interpretation
- Curry-Howard correspondence
- Natural Deduction: Gentzen
- Mathematical Induction; Recursion

Statically Typed Functional Language

- First-class and higher-order functions
- Pure functions (no side-effects)
- Recursion
- Type systems
- Referential transparency (no variables)
- Data structure: Mutable/Immutable
- Pattern Matching; Guards

Dependent Types

The type of a quantity may depend on its value. Types may be complex formulas, or terms.

With *decidable types*, the equality of two types is a decidable (recursive) question.

Arrow Notation; Currying

In OCaml:

```
# let f x y z = x + y + z;;
val f : int -> int -> int -> int = <fun>
# f 2;;
- : int -> int -> int = <fun>
# f 2 3;;
- : int -> int = <fun>
# f 2 3 4;;
- : int = 9
#
```

Higher Order Logic

1. Functions without functional arguments are of first order.
2. Functions with functional arguments of order at most n are of order n+1.

Requiring functions to be of finite type prevents paradoxes, loops, and failure.

Intuitionistic Logic

Some formulas not valid in intuitionistic logic:

1. $\phi \vee \neg\phi$
2. $\neg\neg\phi \rightarrow \phi$
3. $((\phi \rightarrow \chi) \rightarrow \phi) \rightarrow \phi$
4. $(\neg\phi \rightarrow \neg\chi) \rightarrow (\chi \rightarrow \phi)$

BHK Interpretation of IL

(Brouwer, Heyting, Kolmogorov)
Rather than values in {True, False}, let the sentential variables have values in {Provable, Non-provable}. Then the rules of intuitionistic logic make sense.

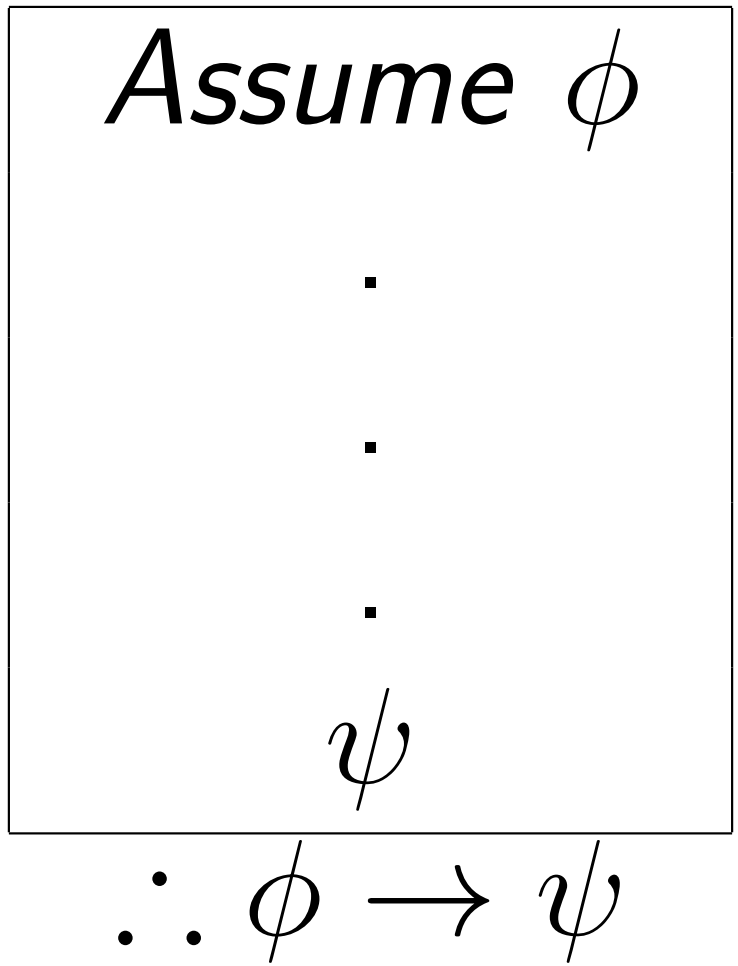
Curry-Howard Correspondence

This is an incredibly fruitful idea linking, for example, intuitionistic sentential formulas with function types displayed with arrows. Here, valid sentential formulas using arrow and parens correspond with those function types where the function is definable from its arguments using only function application. This provides a link between proofs and programs, and between propositions and types.

Natural Deduction

In these systems, there are no axioms.
Rather, each connective is provided with introduction and elimination rules.

Arrow Introduction



Arrow Elimination

Modus Ponens

$$\frac{\phi,\ \phi \rightarrow \psi}{\therefore \psi}$$

OR Introduction

$$\frac{\phi}{\therefore \phi \vee \psi} \qquad \frac{\psi}{\therefore \phi \vee \psi}$$

OR Elimination

$$\frac{\phi \vee \psi,\ \phi \rightarrow \chi,\ \psi \rightarrow \chi}{\therefore \chi}$$

AND Introduction

$$\frac{\phi,\ \psi}{\therefore \phi \wedge \psi}$$

AND Elimination

$$\frac{\phi \wedge \psi}{\therefore \phi} \qquad \frac{\phi \wedge \psi}{\therefore \psi}$$

NOT Introduction

$$\frac{\phi \to \psi,\ \phi \to \neg\psi}{\therefore \neg\phi}$$

NOT Elimination

$$\frac{\phi,\ \neg\phi}{\therefore \psi}$$

FOR ALL Introduction

$$\frac{\phi(t)}{\therefore \forall x \phi(x)}$$

where x is not free in t or any premise or undischarged assumption.

FOR ALL Elimination

$$\frac{\forall x \phi(x)}{\therefore \phi(t)}$$

where no occurrence of any variable in t becomes bound in $\phi(t)$

THERE EXISTS Introduction

$$\frac{\phi(t)}{\therefore \exists x \phi(x)}$$

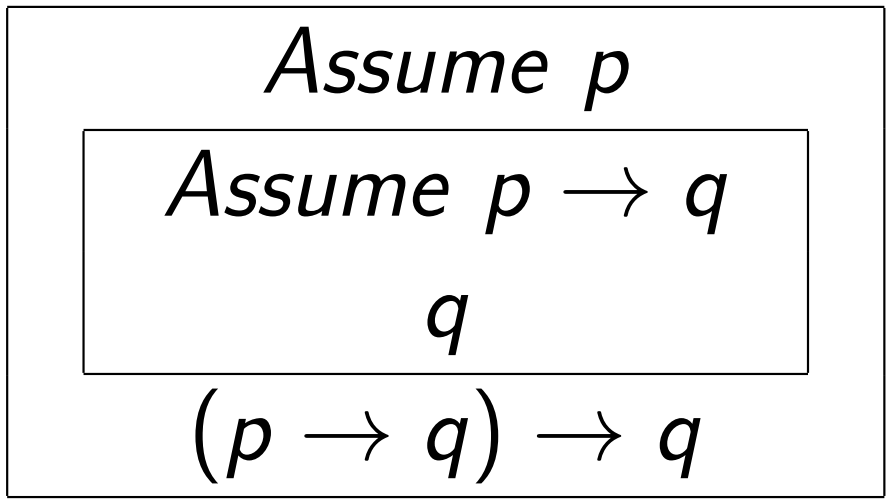
where no occurrence of any variable in t becomes bound in $\phi(t)$

THERE EXISTS Elimination

$$\frac{\exists x \phi(x),\ \phi(y) \rightarrow \psi}{\therefore \psi}$$

where y is not free in ψ or any premise or undischarged assumption.

Example 1



$$p \to ((p \to q) \to q)$$

$$p$$

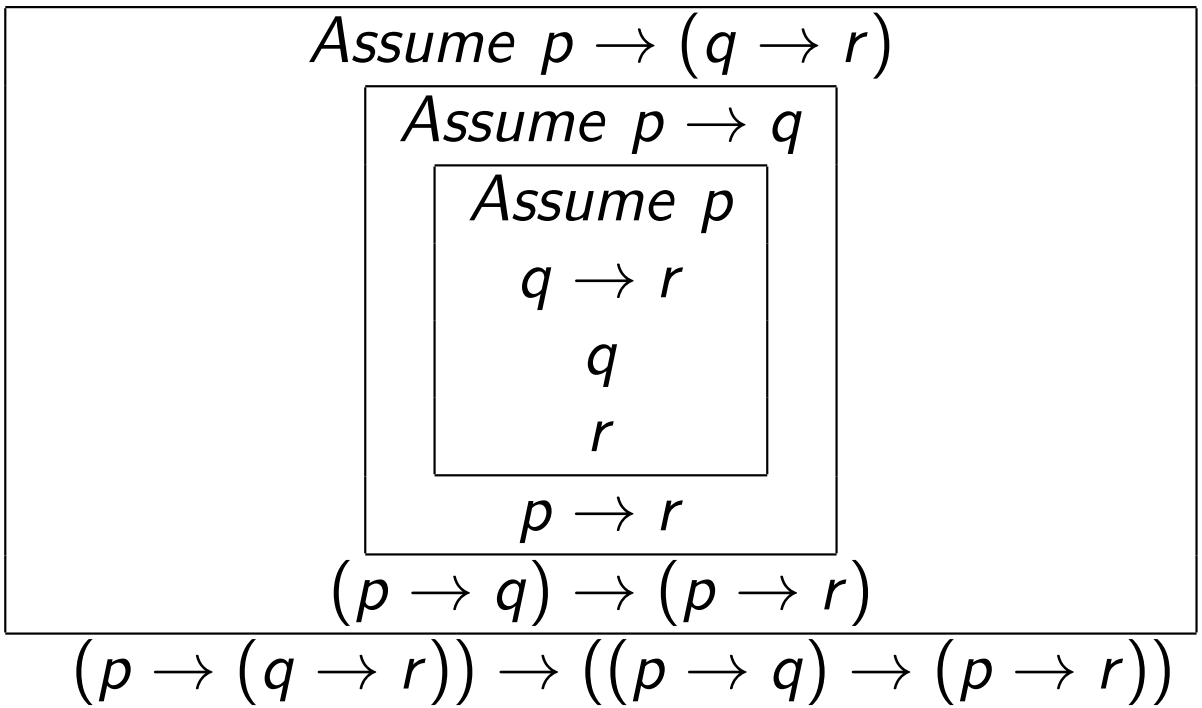
$$g$$

$$q = g\ p$$

$$q = h\ g$$

$$q = f\ p\ g$$

Example 2



$$
\begin{aligned}
f \\
i \\
p \\
g = fp \\
q = ip \\
r = gq = (fp)(ip) \\
r = kp = (fp)(ip) \\
r = jip = (fp)(ip) \\
r = hfip = (fp)(ip)
\end{aligned}
$$

coq.inria.fr

- Documentation list
- Download of Coq and CoqIDE or Proof General (Emacs)
- Coq Community

Search for: Coq Cheat Sheets

Self/Group Study:

- Use Software Foundations document
- Install CoqIDE
- Work through examples, problems

Links to Other Source Material

Princeton.pdf

CoqIDE Screenshot