Formal Verification using COQ

Two lessons on Pro{gramm|v}ing with COQ

Simão Melo de Sousa
RELEASE
\textsc{RELiabLE And SEcure} Computation Group
Computer Science Department
Universidade da Beira Interior
Preliminary Considerations
- Formal Methods, Formal Verification and COQ
- Proof Assistants in the Jungle of Formal Methods

Foundations
- Once upon a time...
- Formal reasoning
- Inductive Definitions
- The essence of computation
- Programming is proving: The Curry-Howard Correspondence

The Basics

Getting Serious

Concluding Remarks
- Success Stories
- That’s all folks
Outline

1. Preliminary Considerations
   - Formal Methods, Formal Verification and COQ
   - Proof Assistants in the Jungle of Formal Methods

2. Foundations

3. The Basics

4. Getting Serious

5. Concluding Remarks
Software and cathedrals are much the same. First we build them, then we pray.

Anonymous

Formal Methods approach:

**Aide toi, et le ciel t’aidera!**

*Jean de La Fontaine - Le Chartier embourbé. Livre VI - Fable 18.*

...That what this course (and these two lessons) is about.
COQ is a Interactive Proof Assistant, thus:

1. COQ is a Heavyweight Formal Method: Powerful, yes... but not for all.
2. Roughly and in a picture, COQ can be seen as a “MATLAB for proofs”
3. Can be used for
   - The formalization of constructive mathematics (e.g. most of “The Hundred Greatest Theorems” have been proved in COQ)
   - The formal verification of programs (our focus in these two lessons)
4. Is not intended in its present form as a tool to be integrated in a “regular” software development process.
5. Nevertheless, COQ is successfully used in the industry. It belongs to the family of tools that are advocated by the *Common Criteria for Information Technology Security Evaluation* (the international standard ISO/IEC 15408 for computer security) for the highest evaluations.
Research institution and universities that use COQ: (France) Ecole Normale Supérieure (ULM, Lyon), Ecole Polytechnique, INRIA (Sophia Antipolis, LORIA, IRISA), Paris 7, LRI-Paris Sud and much more. (USA) CMU, Berkeley, Cornell, Stanford, Harvard, Yale, University of Pennsylvania, etc... (NL) Nijmegen, (Sweden) Chalmers etc...

Here in Portugal: FCUP, UM, IST, UBI, MAPi.

Some Companies that use COQ: Dassault Aviation, France Telecom, CEA, Trusted Logic, Gemauto, etc...
Some resources:

- website: http://coq.inria.fr,
- a very good, complete and exhaustive book about COQ, *The COQ’Art*:
  http://www.labri.u-bordeaux.fr/perso/casteran/CoqArt/index.html,
- a wiki http://logical.futurs.inria.fr/cocorico.
- *<pub>* the forthcoming Springer-Verlag book “*Rigorous Software Development*” by Jorge Sousa Pinto, José Carlos Bacelar and myself (scheduled for 2009) that will include a chapter on Formal Verification with COQ (but also Design by Contract, Model based Specification and Verification) *</pub>*
Resources

- Coq comes with a compiler coqc and a toplevel (coqtop)
- Interface for Editing Proofs:
  - CoqIDE, the graphical user interface distributed with Coq. http://coq.inria.fr/coqide/
  - ProofGeneral. ProofGeneral is an EMACS generic interface for proof assistants. http://proofgeneral.inf.ed.ac.uk/
- Presenting Proofs: coqdoc exports vernacular file to TeX or HTML. It is part of the Coq distribution and documented in the Reference Manual.
- Related Tools for Software Verification (tools belonging to the galaxy of “Design By Contract-JML” and that target COQ):
  - Caduceus - FramaC: http://why.lri.fr/caduceus/index.en.html
  - Krakatoa: http://krakatoa.lri.fr/
The Main Quest of Formal Methods

The Quest:

Provide *evidences* that a computer/software artifact has a given (expected) behavior
The Main Quest of Formal Methods

The Quest:

Provide *evidences* that a computer/software artifact has a given (expected) behavior

Central Notion of \( \{ \text{− model} \quad \text{− specification} \} \) that brings the object under study to mathematics.
The Main Quest of Formal Methods

The Quest:

Provide *evidences* that a computer/software artifact has a given (expected) behavior

This quest can be split into two sub-problems to solve:

**Guarantees of behavior:** How to ensure/verify, at the model level, a given behavior?

**Model versus Implementation:**

1. How to obtain, from the model, an implementation that follows its behavior?
2. How to ensure that a given code shares the same behavior with the model?

COQ provides support for these two sub-problems: Model, prove, and extract to programming languages
Specifying and Proving properties

3 kinds of support for the Formal verification (J. Rushby)

1. Tools that just provide a formal environment
2. Tools that provide a formal system for the precise formulation of the reasoning
3. Tools that provide a formal system and computational support for the precise formulation of the reasoning

Proofs “by hand”, expressed in a natural language. The proof is validated only if there is consensus by the research community.

Where are we?

⇒ COQ belong to the “level 3” family

S. Melo de Sousa

Coq in two lessons
Specifying and Proving properties

3 kinds of support for the Formal verification (J. Rushby)

1. Tools that just provide a formal environment
2. Tools that provide a formal system for the precise formulation of the reasoning
3. Tools that provide a formal system and computational support for the precise formulation of the reasoning

Proof “by hands”, but the proof is expressed in a rigorous language.

Where are we?

⇒ COQ belong to the “level 3” family

⇒ S. Melo de Sousa

⇒ Coq in two lessons
Specifying and Proving properties

3 kinds of support for the Formal verification (J. Rushby)

1. Tools that just provide a formal environment
2. Tools that provide a formal system for the precise formulation of the reasoning
3. Tools that provide a formal system and computational support for the precise formulation of the reasoning

Much more rigorous (e.g. Lamport, etc...) i.e. in the same framework: the model and its “conformance” proof are done in the same framework, systematic/mechanical verification of the proof of the model properties proofs.

Where are we?

⇒ COQ belong to the “level 3” family
Specifying and Proving properties

3 kinds of support for the Formal verification (J. Rushby)

1. Tools that just provide a formal environment
2. Tools that provide a formal system for the precise formulation of the reasoning
3. Tools that provide a formal system and computational support for the precise formulation of the reasoning

Much more rigorous (e.g. Lamport, etc...) i.e. in the same framework: the model and its “conformance” proof are done in the same framework, systematic/mechanical verification of the proof of the model properties proofs.

Where are we?

⇒ COQ belong to the “level 3” family
Concepts

Formal Systems = Deductive Systems

I.e.: provide in the same formalism means for the expression of models, properties and proofs.

The large variety of formalisms that fall in this family can be explained by the compromise between two conflicting factors that must be taken into account:

Logic expressiveness

versus

Automation of the reasoning
Interactive Proof Systems

...or Proof Assistants

Preference: Logic expressiveness (usually a High-Order Logic)

Advantages: Can express a very big class of concepts, properties and proofs, that are required for the complete formal verification of complex systems. The reasoning ability is close to the standard mathematical reasoning.

Drawbacks: Undecidability of the underlying logic \(\Rightarrow\) User intervention is often required

Systems: Coq, PVS, DECLARE, HOL, ISABELLE
Outline

1 Preliminary Considerations

2 Foundations
   • Once upon a time...
   • Formal reasoning
   • Inductive Definitions
   • The essence of computation
   • Programming is proving: The Curry-Howard Correspondence

3 The Basics

4 Getting Serious

S. Melo de Sousa
Coq in two lessons
A short bio

- Coq is the result of more than 20 years of research.
- It started in 1984 from an implementation of the Calculus of Constructions at INRIA-Rocquencourt (France) by Thierry Coquand and Gérard Huet.
- In 1991, Christine Paulin (LRI-Orsay) extended it to the Calculus of Inductive Constructions.
- COQ is an interactive proof assistant with a strongly typed (including inductive and dependent types), high order, polymorphic functional language *(the programming language)* equipped with a higher order logic *(the proof language and support)*. I.e. Calculus of Inductive Constructions in a few words...
- Fortunately, we will see in the next slides the meaning of all these weird concepts.
Consider a logic (propositional, first order or high order, etc...) and its set of formula $\mathcal{F}$.

**Definition (Judgment)**

A judgment is a pair $(\Gamma, F)$ where $\Gamma (\subset \mathcal{F})$ is a finite set of formulas and $F$ a formula.

**Notation:** $\Gamma \vdash F$.

Meaning: $\Gamma \vdash F$ is an affirmation. Admitting the validity of the formulas of $\Gamma$, $F$ can be deduced. ($\vdash = \text{deduction}$). Obviously such an affirmation is valid iff on can provide a proof of its validity. This is the goal of a deductive system like **Natural Deduction**.
Inference rule

\[
P_1 \quad P_2 \quad \cdots \quad P_n
\]

\[
\begin{array}{c}
\text{Goal} \\
\hline
P_1 \quad P_2 \quad \cdots \quad P_n
\end{array}
\]

- **forward meaning**: If the premises \( P_1 \ P_2 \ \cdots \ P_n \) are verified (i.e. have a proof) then one can built a proof of Goal.

- **backward meaning**: In order to prove Goal, one can successfully prove \( P_1 \ P_2 \ \cdots \ P_n \).

Special case (the most basic, in fact)

\[
\begin{array}{c}
\text{Goal} \\
\hline
\text{Goal}
\end{array}
\]

A proof of Goal can be built without premises.
A proof tree is a combination of inference rules of the form

\[
\begin{align*}
\vdots \\
\vdots \\
\vdots \\
&P_1 \\
&P_2 \\
&\cdots \\
&P_n
\end{align*}
\]

such that the root is the statement to be proved, the leaves are inference rules without premises.

Proof tree = Proof
Natural Deduction

- Deduction system defined by G. Gentzen (1935) which rules try to reflect the usual mathematical reasoning and proofs.
- For each connective, there are two types of rules:
  - **introduction** rules
  - **elimination** rules.
axiom \[ A \in \Gamma \quad \frac{}{\Gamma \vdash A} \]

intro \[ \Gamma, A \vdash B \quad \frac{}{\Gamma \vdash A \rightarrow B} \]

intro \[ \Gamma \vdash A \quad \Gamma \vdash B \quad \frac{}{\Gamma \vdash A \land B} \]

elim \[ \Gamma \vdash A \land B \quad \frac{}{\Gamma \vdash B} \]

elim \[ \Gamma \vdash A \land B \quad \frac{}{\Gamma \vdash A} \]

intro \[ \Gamma \vdash A \quad \frac{}{\Gamma \vdash A \lor B} \]

elim \[ \Gamma \vdash A \lor B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C \quad \frac{}{\Gamma \vdash C} \]

elim \[ \Gamma \vdash A \quad \Gamma \vdash \neg A \quad \frac{}{\Gamma \vdash \bot} \]

intro \[ \Gamma, A \vdash \bot \quad \frac{}{\Gamma \vdash \neg A} \]

elim \[ \Gamma \vdash A \quad \Gamma \vdash \neg A \quad \frac{}{\Gamma \vdash \bot} \]

intro \[ \Gamma \vdash A \quad \Gamma \vdash B \quad \frac{}{\Gamma \vdash A \leftrightarrow B} \]

elim \[ \Gamma \vdash A \leftrightarrow B \quad \Gamma \vdash B \quad \frac{}{\Gamma \vdash A} \]

elim \[ \Gamma \vdash \exists x F \quad t \text{ term} \quad \Gamma, F[x := t] \vdash G \quad \frac{}{\Gamma \vdash G} \]

elim \[ \Gamma \vdash \forall x F \quad \forall t \text{ term} \quad \frac{}{\Gamma \vdash F[x := t]} \]

*Figura: Natural Deduction for first order logic*
An example

\[
\begin{array}{c}
\text{axiom} \quad (A \rightarrow B) \vdash (A \rightarrow B) \\
\text{axiom} \quad A \vdash A \\
\text{axiom} \quad (B \rightarrow C) \vdash (B \rightarrow C)
\end{array}
\]

\[
\begin{array}{c}
(A \rightarrow B), A \vdash B \\
(A \rightarrow B), (B \rightarrow C), A \vdash C \\
(A \rightarrow B), (B \rightarrow C) \vdash (A \rightarrow C) \\
(A \rightarrow B) \vdash (B \rightarrow C) \rightarrow (A \rightarrow C) \\
\vdash (A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))
\end{array}
\]

A proof of \( \vdash (A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C)) \)
Definition (Inductively Defined Set)

Consider a Set $E$, a non-empty sub-set $B$ of $E$ (called the base) and a set $K$ of operations $\phi : E^{\#(\phi)} \rightarrow E$ (the set of constructors). A sub-set $X$ of $E$ is said **inductively defined** if it is the smallest set that verify:

(B): $\forall x \in B \implies x \in X$,

(I): $\forall \phi \in K, x_1, \ldots, x^{\#(\phi)} \in X \implies \phi(x_1, \ldots, x^{\#(\phi)}) \in X$.

It can be convenient to see such term as trees (Leaf = elements of the base, Nodes = constructors). From this point of view it is easy do induce a notion of sub-term (= sub-tree).
Theorem (Structural Induction)

Consider an inductively defined set $X$ and a predicate $P(x)$ over $X$. If

(B') For all $x \in B$, $P(x)$.

(I') For all $\phi \in K$ ($P(x_1), \ldots, P(x_{\#(\phi)})$) $\implies P(\phi(x_1, \ldots, x_{\#(\phi)}))$

then $\forall x \in X, P(x)$. 

S. Melo de Sousa

Coq in two lessons
**Definition**

Let be $X \subseteq E$ an (non-ambiguous) inductively defined set. Let be $F$ a set. A structurally recursive function is a recursive function in which each recursive call is done with a sub-term of the initial argument.

**Theorem (Termination for free)**

There is no infinite recursive calls in the evaluation of a structurally recursive function.
Example: Binary Trees

- The set $AB$ of binary tree over $A$ is inductively defined over $(A ∪ \{\emptyset, (\"\",")\",")\",")\})^*$ by
  
  (B) $\emptyset \in AB$ (empty tree)
  
  (I) $e, d \in AB, \forall a \in A \implies (a, g, d) \in AB$ (the tree with the root $a$, the left subtree $e$ and the right subtree $d$).

- The induction principle is:
  $$P(\emptyset) \land (\forall x \in A, e, d \in ABP(e) \land P(d) \implies P((x, e, d))) \implies \forall a \in AB, P(a)$$

- A function:
  $$inf(x) = \begin{cases} 
  \epsilon & \text{se } x = \emptyset \\
  inf(g).a.inf(d) & \text{se } x = (a, g, d)
  \end{cases}$$
λ-calculus

- conceived (ca. 1930) as part of a general (later shown inconsistent) theory of functions and logic, intended as a foundation for mathematics;

- all recursive functions can be represented in the (pure) λ-calculus (i.e. Turing Complete);

- theory modeling functions and their applicative behavior;

- concept of function seen as a rule, i.e. process of passing an argument to a value (contrary to the notion of seeing a function as a graph);

- this is important for the study of computability and for theory of computation in general, since it emphasizes the computational aspect associated to the notion of function.

- Give rise to important applications (functional programming languages, constructive mathematics, computational linguistics, reasoning by computer, programming languages semantics, and much more..)
Consider $\mathcal{V} = \{x, y, z, t, \ldots\}$, a possibly infinite, countable set of variables.

**Definition (Inductive definition of $\lambda$-terms)**

- Each variable $x$ is a $\lambda$-term;
- If $M$ and $N$ are $\lambda$-terms, then $(MN)$ is a $\lambda$-term, (application);
- If $M$ is a $\lambda$-term and $x$ a variable, then $(\lambda x. M)$ is a $\lambda$-term, (abstraction).

Examples: $(\lambda x.x)$, $(x(\lambda y.(xy)))$, $\lambda x y z t. x y z t$ that stands for $((((\lambda x.(\lambda y.(\lambda z.(\lambda t.(((xy)z)t))))))))$.
\(\alpha\)-conversion

all occurrences of a variable \(x\) that occur in an expression of the form \(\lambda x.M\) are bound;

- an occurrence of a variable that is not bound is called free;
- \(FV(M)\) is the set of variables with free occurrences in \(M\);
- if \(FV(M) = \emptyset\) we say that \(M\) is closed;
- we will consider \(\lambda\)-terms equivalent up to bound variable renaming, (\(\alpha\)-conversion).

Examples: \(\lambda xy.xyz \equiv_\alpha \lambda yu.yuz\), but \((\lambda x.x)z \not\equiv_\alpha (\lambda x.y)z\)
Substitutions

The expression $M[N/x]$ denotes the result of substituting in $M$ each free occurrence of $x$ by $N$ and making any changes of bound variables needed to prevent variables free in $N$ from becoming bound in $M[N/x]$.

Example:

$$(\lambda xy. xyz)[(\lambda u. y)/z] \not\equiv \lambda xy. xy(\lambda u. y)$$

but

$$(\lambda xy. xyz)[(\lambda u. y)/z] \equiv \lambda xv. xv(\lambda u. y)$$
The execution model: \( \beta \) reduction

- a term of the form \((\lambda x. M) N\) is called a \(\beta\)-redex;
- its contractum is the term \( M[N/x] \);
- we write \( M \rightarrow^{1\beta} N \), and say that \( M \) reduces in one step of \( \beta \)-reduction to \( N \), iff \( N \) can be obtained from \( M \) by replacing one \( \beta \)-redex in \( M \) by its contractum;
- \( \rightarrow_\beta \) is the reflexive and transitive closure of \( \rightarrow^{1\beta} \);
- \( \equiv_\beta \) is the reflexive, symmetric and transitive closure of \( \rightarrow^{1\beta} \).
The result of a computation: a normal form

- A term M is said to be in $\beta$-normal form (or $\beta$-nf) if it contains no $\beta$-redex;
- we say that M has a $\beta$-nf if there is some $\beta$-nf N such that $M \rightarrow_\beta N$.

Not all $\lambda$-terms have $\beta$-nf or not all $\beta$-reductions lead to $\beta$-nf:
- The term $(\lambda x.x)(\lambda x.x)$ has no $\beta$-nf since
  $(\lambda x.x)(\lambda x.x) \rightarrow_{1\beta} (\lambda x.x)(\lambda x.x) \rightarrow_{1\beta} (\lambda x.x)(\lambda x.x) \rightarrow_{1\beta} \cdots$
- the term $(\lambda xy.x)(\lambda x.x)((\lambda x.xx)(\lambda x.xx))$ has normal form $\lambda x.x$, but not every reduction sequence leads to this normal form.
Programming in the pure $\lambda$-calculus

- **notation:** $F^nX = F(F(\ldots(FX))\ldots)$
- **Natural numbers, via Church numerals:** $c_n = \lambda fx.f^n x$, for $n \geq 0$;
- **$A_+$ = add =** $\lambda mnfx.mf(nfx)$, $(A_+ c_n c_m \equiv c_{n+m})$;
- **$A_\times = mult =** \lambda mnfx.m(nf)x$, $(A_\times c_n c_m \equiv c_{n+m})$;
- **$A_{\exp} = exp = \lambda mnfx.nmfix$,** $(A_{exp} c_n c_m \equiv c_{n+m})$;
- **succ = $\lambda nfx.f(nfx);**
- **Booleans,** $true = \lambda xy.x$, $false = \lambda xy.y$;
- **if = $\lambda bxy.bxy,**
  $(if \ true \ M \ N \equiv M \ and \ if \ false \ M \ N \equiv N)$;
- **iszero = $\lambda n.x(\lambda x.false)true$;**
- **Ordered pairs,**
  $(pair = (.,.) = \lambda xy.fxy)$;
- **fst = in$_1 = \lambda p.\true,**
  $(fst (pair M N) \equiv M)$;
- **snd = in$_2 = \lambda p.\false,**
  $(snd (pair M N) \equiv N)$;
- **prefn =** $\lambda fp.pair(f(fstp))(fstp)$;
- **pre =** $\lambda nfx.snd(n(prefn f)(pair \times x))$;
- **sub = $\lambda mn.n prem;**
- **Lists nil = $\lambda z.z;$**
- **cons = $\lambda xy.pair \false (pair \times y);$**
- **null = fst;**
- **hd = $\lambda z.fst(snd \ z);$**
- **tl = $\lambda z.snd(snd \ z).**
- **fixed point combinators Y: term such that $\forall F, YF \equiv F(YF)$**
- **Curry Fixed Point combinator:** $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- **Recursive Functions via fixed point combinator**
- **fact =** $\lambda n.((Y(\lambda fx.(\iszero x)1(multx(f\pre x))))\times n)$.\n
S. Melo de Sousa  Coq in two lessons
And what happens if one wants to type λ-calculus? Simply typed λ-calculus.

- allow to conveniently refuse λ terms like \((\lambda x.x \; x)\)

- This has a price: but we lose the ability to “loop” (i.e. not Turing complete).

- Intuitively, we enrich the λ-calculus with a notion of type. Let \(\{\alpha, \beta, \ldots\}\) a countable set of type variables. The variables \(\alpha, \beta, \ldots\) are the atomic types. Thus, if \(x\) has type \(\alpha\) and \(M\) the type \(\beta\) then \((\lambda x. M)\) has type \(\alpha \rightarrow \beta\)

- Notion of type judgment: “under the declarations \(x_1 : \alpha_1 \ldots x_n : \alpha_n\) the term \(t\) has type \(\alpha\)”.\n
Types and \( \lambda \) calculi

- A variable has type \( \alpha \) if it was declared of type \( \alpha \)

\[
\frac{x : \alpha}{\Gamma \vdash x : \alpha}
\]

- If \( M \) is a function of type \( \alpha \rightarrow \beta \) and \( N \) of type \( \alpha \), then \( (M \, N) \) is of type \( \beta \).

\[
\frac{\Gamma \vdash M : \alpha \rightarrow \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash (MN) : \beta}
\]

- If, assuming that \( x \) is of type \( \alpha \), one can infer that \( M \) is of type \( \beta \) then there is a function \( \lambda x. M \) of type \( \alpha \rightarrow \beta \)

\[
\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash (\lambda x. M) : \alpha \rightarrow \beta}
\]
Natural Deduction Intuitionistic Propositional Logic

\[
\begin{align*}
[A] & \quad \therefore B \\
\therefore B & \quad (\exists I)
\end{align*}
\]

\[
\begin{align*}
A \supset B & \quad B \\
A \supset B & \quad (\supset \varepsilon)
\end{align*}
\]

\[
\begin{align*}
\therefore A, B & \quad \therefore \neg \neg A \\
A & \quad \neg \neg A
\end{align*}
\]

\[
\begin{align*}
A \wedge B & \quad \therefore A \\
A \wedge B & \quad (\wedge \varepsilon)
\end{align*}
\]

\[
\begin{align*}
\therefore A, B & \quad \therefore \neg \neg (A \wedge B) \\
A \wedge B & \quad \neg \neg (A \wedge B)
\end{align*}
\]

\[
\begin{align*}
A & \quad \neg A \\
A & \quad (\neg \varepsilon)
\end{align*}
\]

\[
\begin{align*}
\therefore A, B & \quad \therefore \neg \neg (A \vee B) \\
A \vee B & \quad \neg \neg (A \vee B)
\end{align*}
\]

\[
\begin{align*}
\therefore A, B & \quad \therefore \neg \neg C \\
A \vee B, C & \quad \neg \neg C
\end{align*}
\]
‘Sequent Style’ Natural Deduction

\[ \Gamma, A \vdash A \ (ax) \]

\[ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B} \ (\supset \ I) \]

\[ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \land B} \ (\land \ I) \]

\[ \frac{\Gamma \vdash A}{\Gamma \vdash A \lor B} \ (\lor \ I) \]

\[ \frac{\Gamma \vdash B}{\Gamma \vdash A \lor B} \ (\lor \ I) \]

\[ \frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B} \ (\supset \ E) \]

\[ \frac{\Gamma \vdash A \land B}{\Gamma \vdash A} \ (\land \ E) \]

\[ \frac{\Gamma \vdash A \land B}{\Gamma \vdash B} \ (\land \ E) \]

\[ \frac{\Gamma \vdash A \land B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \ (\lor \ E) \]
Recap with suggestive spaces . . .

\[ \Gamma, \quad A \vdash A \]

\[ \begin{array}{c}
\Gamma, \quad A \vdash B \\
\hline
\Gamma \vdash A \cup B
\end{array} \quad \begin{array}{c}
\Gamma \vdash A \cup B \\
\Gamma \vdash A
\end{array} \]

\[ \begin{array}{c}
\Gamma \vdash B \\
\hline
\Gamma \vdash B
\end{array} \quad \begin{array}{c}
\Gamma \vdash A \land B \\
\Gamma \vdash A
\end{array} \quad \begin{array}{c}
\Gamma \vdash A \land B \\
\Gamma \vdash B
\end{array} \]

\[ \begin{array}{c}
\Gamma \vdash A \\
\hline
\Gamma \vdash A \lor B
\end{array} \quad \begin{array}{c}
\Gamma \vdash B \\
\hline
\Gamma \vdash A \lor B
\end{array} \]

\[ \begin{array}{c}
\Gamma, \quad A \vdash C \\
\hline
\Gamma, \quad B \vdash C
\end{array} \quad \begin{array}{c}
\Gamma \vdash C
\end{array} \]
...

\[ \Gamma, \quad A \vdash A \]

\[ \frac{\Gamma, \quad A \vdash B}{\Gamma \vdash A \rightarrow B} \]

\[ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \]

\[ \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \]

\[ \frac{\Gamma \vdash A \times B}{\Gamma \vdash B} \]

\[ \frac{\Gamma \vdash A}{\Gamma \vdash A + B} \]

\[ \frac{\Gamma \vdash A + B \quad \Gamma, \quad A \vdash C \quad \Gamma, \quad B \vdash C}{\Gamma \vdash C} \]
... just add terms to get Simply Typed $\lambda$-Calculus!

\[
\begin{align*}
\Gamma, x : A & \vdash x : A \\
\Gamma, x : A & \vdash M : B \\
\Gamma & \vdash \lambda x : A.M : A \to B \\
\Gamma & \vdash M : A \to B \\
\Gamma & \vdash N : A \\
\Gamma & \vdash MN : B \\
\Gamma & \vdash M : A \times B \\
\Gamma & \vdash \pi_1 M : A \\
\Gamma & \vdash \pi_2 M : B \\
\Gamma & \vdash M : A \\
\Gamma & \vdash \text{in}_1 M : A + B \\
\Gamma & \vdash \text{in}_2 M : A + B \\
\Gamma & \vdash M : A + B \\
\Gamma, x_1 : A & \vdash N_1 : C \\
\Gamma, x_2 : B & \vdash N_2 : C \\
\Gamma & \vdash \text{case } M \text{ of } \text{in}_1x_1 \Rightarrow N_1 | \text{in}_2x_2 \Rightarrow N_2 : C
\end{align*}
\]
BHK - interpretation (wikipedia)

In mathematical logic, the Brouwer-Heyting-Kolmogorov interpretation, or BHK interpretation, of intuitionistic logic was proposed by L. E. J. Brouwer, Arend Heyting and independently by Andrey Kolmogorov.

The interpretation states exactly what is intended to be a proof of a given formula. This is specified by induction on the structure of that formula:

- A proof of $P \land Q$ is a pair $< a, b >$ where $a$ is a proof of $P$ and $b$ is a proof of $Q$.
- A proof of $P \lor Q$ is a pair $< a, b >$ where $a$ is 0 and $b$ is a proof of $P$, or $a$ is 1 and $b$ is a proof of $Q$.
- A proof of $P \rightarrow Q$ is a function $f$ which converts a proof of $P$ into a proof of $Q$.
- A proof of $\exists x \in S : \phi(x)$ is a pair $< a, b >$ where $a$ is an element of $S$, and $b$ is a proof of $\phi(a)$.
- A proof of $\forall x \in S : \phi(x)$ is a function $f$ which converts an element $a$ of $S$ into a proof of $\phi(a)$.
- The formula $\neg P$ is defined as $P \rightarrow \bot$, so a proof of it is a function $f$ which converts a proof of $P$ into a proof of $\bot$.
- $\bot$ is absurdity. There ought not be a proof of it.

The interpretation of a primitive proposition is supposed to be known from context.
The Curry-Howard correspondence extends the BHK-interpretation and is the direct relationship between computer programs and mathematical proofs. It refers to the generalization of a syntactic analogy between systems of formal logic and computational calculi that was first discovered by the American mathematician Haskell Curry and logician William Alvin Howard.

<table>
<thead>
<tr>
<th>logic</th>
<th>$\lambda$-calculus</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>formula</td>
<td>type</td>
<td>specification</td>
</tr>
<tr>
<td>proof</td>
<td>term</td>
<td>program</td>
</tr>
<tr>
<td>cut</td>
<td>$\beta$-reduction</td>
<td>execution</td>
</tr>
</tbody>
</table>

The relation with programming is particularly interesting when the logic/type language is rich enough. This is the case for the Calculus of Inductive Construction.
Consider

\[ \forall (x, y) \in \mathbb{N}^2. \exists (q, r) \in \mathbb{N}^2. (y = (q \times x + r) \land r < x) \]

The proof of such theorem can be seen as a function that computes from \((x, y)\) the pair \((q, r)\) and the proof that 
\(y = (q \times x + r) \land r < x\).

COQ can even produce a complete program (Haskell, OCaml, Scheme) from such proof! Indeed, such a program is proved correct by construction.
Even Nicer....

- Theorem: Every Java program has an equivalent x86 machine language program.
- By choosing a suitable constructive logic, we guarantee that any proof of this theorem can be converted into a genuine Java compiler!
- By using a generic program extraction mechanism, we get the free theorem that our compiler preserves the semantics of programs.

... which saves us a huge amount of testing.
Outline

1 Preliminary Considerations
2 Foundations
3 The Basics
4 Getting Serious
5 Concluding Remarks
Slides set 1.
Slides set 2.
Outline

1 Preliminary Considerations
2 Foundations
3 The Basics
4 Getting Serious
5 Concluding Remarks
   • Success Stories
   • That’s all folks
Some successful Applications of COQ

A contribution

Formal verification of the Java-Card Platform in COQ (joint work with G. Barthe, G. Dufay): Design of an innovative methodology for the automatic generation of

1. a specification and prototype of JavaCard Execution Platform and
2. the proof that (Milner citation)

Well-typed (JavaCard) Programs cannot go wrong

3. A (proved) correct implementation of the ByteCode Verifier (BCV), a crucial security module based on static program analysis.
Completion of the formalization of the four colors theorem in Coq. Full formalization of the 4 colors theorem in Coq has been completed in December 2004. Started in 1999 at INRIA-Rocquencourt by Georges Gonthier et Benjamin Werner, the project has been completed by Georges Gonthier who found support at Cambridge Microsoft Research. The formalization, based on Robertson, Sanders, Seymour and Thomas (RSST) proof from 1995 has been done in Coq V7.3. Port to Coq 8.0 is ongoing. The formalization has involved the following issues:

- Use of computational reflexion to prove the part of the RSST proof that was left to a C program.
- Use of an original notion of hypermaps to concisely represent planar graphs.
- Use of specific proof techniques: design of a very small language of expressive tactics with a concise syntax; computation on decidable propositions by embedding within the booleans.
- Formalization of a topological "hat" on top of the combinatoric proof on graphs and extension to the infinite case thanks to a construction of real numbers using Dedekind cuts.

The project has motivated new extensions of Coq. Especially, a new optimized reduction machine dedicated to computation of reflexion tactics will be available in the next released version of Coq.
COQ and Common Criteria (excerpt from the official announcements)

- September 2007: a big step in program certification in the real world: The Technology and Innovation group at Gemalto has successfully completed a Common Criteria (CC) evaluation on a Java Card based commercial product. This evaluation is the world’s first CC certificate of a Java product involving EAL7 components.

- Trusted Logic announces (press release of November 18th, 2003) that the DCSSI has successfully evaluated its security methodology applied to the Java Card System at the Common Criteria EAL7 level, in a report published earlier this year. Coq is the proof engine used by Trusted Logics, and was chosen for its expressiveness. As a part of the certification process, it is being acknowledged as trustworthy by the DCSSI.
Certified Compilation (excerpt from the compcert website)

Compcert is a compiler that generates PowerPC assembly code from Clight, a large subset of the C programming language. The particularity of this compiler is that it is written mostly within the specification language of the Coq proof assistant, and its correctness — the fact that the generated assembly code is semantically equivalent to its source program — was entirely proved within the Coq proof assistant.

A high-level overview of the Compcert compiler and its proof of correctness can be found in the following papers:


Main Goal

Ensuring Reliability and Safety of Code Execution in Embedded Systems

This FCT project aims at providing innovative, efficient and expressive mechanisms for the secure implementation and execution of code, with an emphasis on problems posed by embedded systems.

Proposed approach for safety mechanisms

emerging (source level) PCC as a back-end (and COQ as the proof system) for the formal compliance to embedded system safety policies
Rescue in a picture

**Producer Side**

1. Source Code
2. \( VCGEN_{Source} \)
3. Proof System
4. Certificate
5. Compiler
6. Compiled Code + Certificate

**Consumer Side**

1. Security Policies
2. Execution Platform
3. Proof Checker
4. Code
5. Code Loading Stage
6. Certificate
7. Proof Obligations
8. \( VCGEN_{machine} \)
Certified Cryptography

- Correctness of RSA Algorithm, by Jose C. Almeida (DIUM), Laurent Théry
- Certifying Prime Number with the Coq prover. CoqPrime is a library built on top of the Coq proof system to certify primality using Pocklington certificate and Elliptic Curve Certificate.
But also,

- Garbage Collection,
- Operating System modules,
- Communication and Cryptographic Protocols,
- Circuits and Hardware,
- Programming languages methodologies and semantics (see for instance POPLMark)
Final remarks and Perspectives

Current trends

Rise of an attractive market (i.e. $$!) that still has no clear leader (for how long?):

Intel, Microsoft, IBM, NASA, Esterel Technology, Prover Technology, Clearsy, Trusted Logic, Escher Technologies, Siemens, Alstom, Keesda, Systerel, Lerios Technologies, Critical Software (pt), EdiSoft (pt), Efacec (pt), etc...
The relevance of the Common Criteria is growing. But older computer system and software development standards are integrating formal methods in their quality insurance layer.

Perspectives: 2 directions.
- Conceptual and Technical Development of new solutions (i.e. formalisms, tools and techniques)
- Current trends: A growing use of FM (I don’t say in everywhere... but in much more industries than one usually think of)

FM =
- A valuable and highly sought skills
- The software/computer engineer, aside be able to produce well designed software, should also know how to validate and provides evidences.
Current trends

- The relevance of the Common Criteria is growing. But older computer system and software development standards are integrating formal methods in their quality insurance layer.

- Perspectives: 2 directions.
  - Conceptual and Technical Development of new solutions (i.e. formalisms, tools and techniques)
  - Current trends: A growing use of FM (I don’t say in everywhere.... but in much more industries than one usually think of)

FM =

- A valuable and highly sought skills
- The software/computer engineer, aside be able to produce well designed software, should also know how to validate and provides evidences

S. Melo de Sousa

Coq in two lessons
Current trends

- The relevance of the Common Criteria is growing. But older computer system and software development standards are integrating formal methods in their quality insurance layer.

- Perspectives: 2 directions.
  - Conceptual and Technical Development of new solutions (i.e. formalisms, tools and techniques)
  - Current trends: A growing use of FM (I don’t say in everywhere.... but in much more industries than one usually think of)

- FM =
  - A valuable and highly sought skills
  - The software/computer engineer, aside be able to produce well designed software, should also know how to validate and provides evidences
Un peu de programmation éloigne de la logique mathématique; beaucoup de programmation y ramène.

Xavier Leroy.