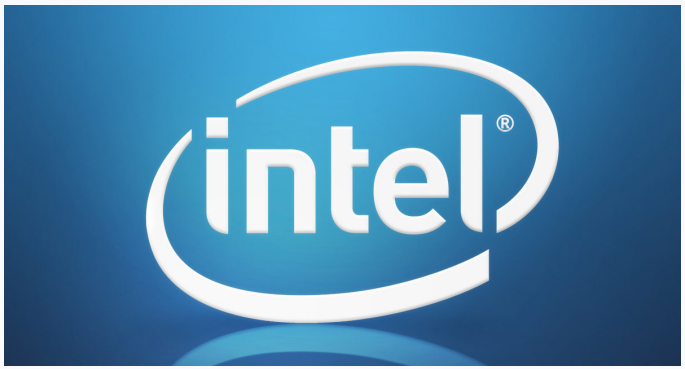
Constructing Trusted Code Base IV

Aleksy Schubert & Jacek Chrząszcz



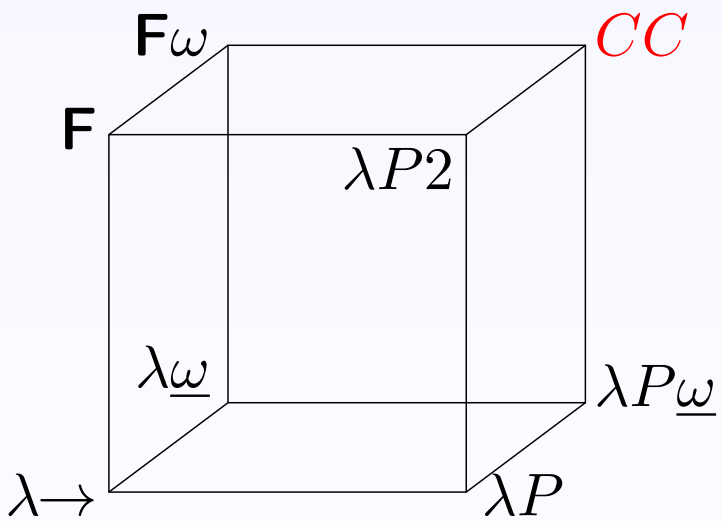
Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Coq — formalism

Coq — higher order logic (calculus of constructions (CC) + inductive definitions)



↑ polimorphism

↗ type constructors

→ dependent types

Coq — formalism: fun for all

simple types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \Pi x{:}A.B(x)}$$

Shorthand: $A \to B$ is $\forall x{:}A.B$, where $x \notin FV(B)$

concrete Coq syntax:

`fun n:nat =>` M `: forall n:nat, vector n`

application rule:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash F\,G : B} \qquad \frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash N : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — formalism: sorts

• Is `nat` $\rightarrow$ `nat` well-formed? Or `vector -> nat`? What is its type?
It is `Set`.

In general, the type of a (well-formed) product is a *sort*.

• What is the type of `Set` ? (or, in general, of a sort ?)

Another sort :)

• Sorts in Coq:

$$\begin{matrix} Prop \\ Set \end{matrix} : Type_1 : Type_2 : \ldots$$

• Cummulativity (or sub-sorting):

$$Prop \leq Set \leq Type_1 \leq Type_2 \leq \ldots$$

Coq — products and sorts

product rule

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ if s_1 and s_2 are ...

- $s_1 \leq s_2$, or
- $s_2 = \mathit{Prop}$

cummulativity rule

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2}$$ if $s_1 \leq s_2$

Coq — products

- function types ($\lambda n.n+1$)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{nat} \ : \ \texttt{Set}}{\Gamma \vdash \texttt{nat} \to \texttt{nat} \ : \ \texttt{Set}}$$

- polymorphism ($\lambda x.x$)

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \to \alpha : \texttt{Set}}{\Gamma \vdash \forall \alpha : \texttt{Set}.\alpha \to \alpha \ : \ \texttt{Type}}$$

- types of predicates (`Even`)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Prop} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Prop} \ : \ \texttt{Type}}$$

Coq — products (cont.)

- type constructors (`List`)

$$\frac{\Gamma \vdash \texttt{Set} \ : \ \texttt{Type} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{Set} \to \texttt{Set} \ : \ \texttt{Type}}$$

- dependent types (`vector`)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Set} \ : \ \texttt{Type}}$$

- impredicativity (Church numerals)

$$\frac{\Gamma \vdash \texttt{Prop} \ : \ \texttt{Type} \qquad \Gamma, \alpha : \texttt{Prop} \vdash \alpha \to (\alpha \to \alpha) \to \alpha \ : \ \texttt{Prop}}{\Gamma \vdash \forall \alpha : \texttt{Prop}. \alpha \to (\alpha \to \alpha) \to \alpha \ : \ \texttt{Prop}}$$

Coq — sorts

Where are the sorts?

- If $\Gamma_1, x : A, \Gamma_2 \vdash M : B$ then
  $\Gamma_1 \vdash A : s$ and
  $\Gamma_1, x : A, \Gamma_2 \vdash B : s'$

- If $\Gamma \vdash$ `fun` x`:`A `=>` $M : B$, then $\Gamma \vdash A : s$

- If $\Gamma \vdash M :$ `forall` x`:`A`.`B, then $\Gamma \vdash A : s$

Is there M such that $\Gamma \vdash M :$ `vector` ?

No, because $\Gamma \vdash$ `vector : nat` $\rightarrow$ `Set`

Is there M such that $\Gamma \vdash M :$ `vector 7` ?

Possibly, because $\Gamma \vdash$ `vector 7 : Set`

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (inductive types reductions — soon :)

Coq — conversion

conversion rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

Coq — inductive definitions

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Inductive type:

```
bool : Set
```

- Constructors:

```
true : bool
false : bool
```

- Destruction:

```
match n with
| true => ...
| false => ...
end
```

Introduction tactics

- apply c_i
- constructor i
- constructor
- left, right = constructor 1, constructor 2
  (if there are only 2 constructors)
- split = constructor 1 = constructor
  (if there are only 1 constructor)
- exists t = split with t
  (if there are only 1 constructor)

Destruction tactics

- `destruct`
- `simple destruct`
- `case` (basic)

- `intros`

Induction tactics

- `induction`
- `simple induction`
- `elim` (basic)
- `elimtype`

For mutually inductive types automatic lemmas are good for nothing.
To generate good ones use:

`Scheme` $\mathtt{id}_1$ `:= Induction for` I_i `Sort` s_1
`with ...`
$\mathtt{id}_n$ `:= Induction for` I_n `Sort` s_n

or non-dependent version:

`Scheme` $\mathtt{id}_1$ `:= Minimality for` I_i `Sort` s_1
`with ...`
$\mathtt{id}_n$ `:= Minimality for` I_n `Sort` s_n

Inductive types and equality

- `0=1 -> ?` `discriminate`
- `S x = S y -> x = y ?` `injection`
- either one: `simplify_eq`

Advanced destruction

How this could happen ?

- `inversion`
  (conclusions from (Even n) and discriminate and injection)
- `inversion_clear`
  (as above, but does some cleaning)
- `dependent inversion`
  (if the destructed name occurs in the “goal”)

Equality inductive type

- Leibniz equality `eq`
- direct proving: `reflexivity` or `split` or `constructor` or `apply eq_refl`
- proving: `symmetry`, `transitivity`
- use: `rewrite`, `rewrite <-`, `subst`, `replace`