



# Dependently typed programming in Coq

Herman Geuvers

Institute for Computing and Information Sciences – Foundations  
Radboud University Nijmegen

The Future of Programming  
TU Delft  
January 16, 2014



# We are moving from e to i



From eGovernment  
Also from eVisser



**iBestuur**  
o n l i n e  
to iGovernment  
to iVisser?





## Vision

### Integration of programming and proving

- Find the computational content of (abstract) mathematical theorems.
- Mathematical proofs become too hard to check by hand (Flyspeck project)
- Precise mathematical specifications of programs
- Prove the (partial) correctness of programs

Method: Powerful type system that can express

- programs
- specifications
- propositions
- proofs



## Outline

- Types in functional languages
- Dependent types and the Propositions-as-Types Isomorphism
- The Coq system and inductive types
- Rich types for programming and proving





# Types in functional languages

```
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y < x]
                ++ [x]
                ++ quicksort [y | y <- xs, y >= x]
```

`quicksort : list nat → list nat`

But we can get more out of types

`quicksort : list a → list a??`

`quicksort` now has a polymorphic type ...?

But that is not correct, because the type  $a$  must have an ordering defined on it.



# Types in functional languages

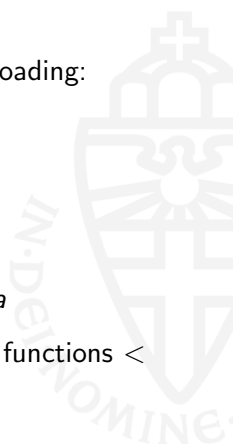
In Haskell, this can be solved by using type class overloading:

```
class Ord a where
  (<), (>=) : a -> a -> Bool
  x >= y   = not (y < x)
```

Then

```
quicksort : (Ord a) => list a -> list a
```

Note: this requires type  $a$  to have two binary boolean functions  $<$  and  $\geq$  defined on it; these need not be orderings.





# Proving properties of programs

quicksort should give a sorted list:

$$\text{Sorted}(l) := \forall i < |l| (l_i \leq l_{i+1})$$

Also the output list should be a permutation of the input list. We define

$$\text{Perm}(l, k) := |l| = |k| \wedge \forall i \leq |l| (\text{occ}(l_i, k) = \text{occ}(l_i, l))$$

where  $\text{occ}(n, l)$  is the number of occurrences of  $n$  in  $l$ .

$$\forall l : \text{list nat}, (\text{Sorted}(\text{quicksort}(l)) \wedge \text{Perm}(l, \text{quicksort}(l)))$$

Gives a complete specification of “sorting”.



# Curry Howard Isomorphism

## Propositions-as-Types

- A constructive proof of a formula is itself a program
- Propositions are **T**ypes
- Proofs are **T**erms
- **PAT**, or in a modern setting **iPAT** (interpretation of **P-as-T**)

$M : A$

Has two readings:

- $A$  is a type, and  $M$  is a program (data) of type  $A$ .
- $A$  is a proposition, and  $M$  is a proof of  $A$ .





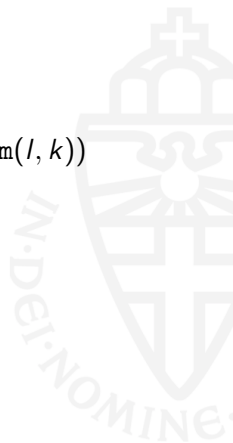
## Curry Howard Isomorphism: another look at sorting

A proof of

$$\forall l : \text{list nat}, \exists k : \text{list nat}, (\text{Sorted}(k) \wedge \text{Perm}(l, k))$$

consists of

- a construction of a list  $k$  out of a list  $l$
- a proof of  $\text{Sorted}(k)$
- a proof of  $\text{Perm}(l, k)$





## Program extraction: A sorting algorithm out of a proof

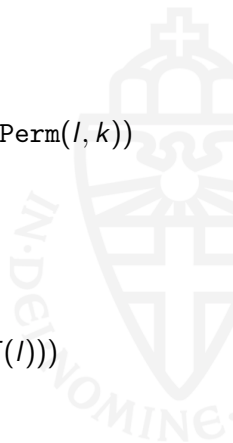
Given a proof

$$P \quad : \quad \forall l : \text{list nat}, \exists k : \text{list nat}, (\text{Sorted}(k) \wedge \text{Perm}(l, k))$$

One can extract from  $P$

- $F : \text{list nat} \rightarrow \text{list nat}$ ;
- a proof of

$$\forall l : \text{list nat}, (\text{Sorted}(F(l)) \wedge \text{Perm}(l, F(l)))$$





## Program extraction: general picture

From a proof

$$P : \forall x : A, \exists y : B, R(x, y)$$

one can extract

- $F : A \rightarrow B$
- a proof of

$$\forall x : A, R(x, F(x))$$

The dependent type system implemented in Coq supports this:  
Coq is an **integrated system for proving and programming**.





# Dependent type theory and propositions-as-types

Data types		Propositions	
non-dependent	dependent	non-dependent	dependent
$A \rightarrow B$		$A \rightarrow B$	
	$\prod x:A. B(x)$		$\forall x:A. B(x)$
$A \times B$		$A \wedge B$	
	$\sum x:A. B(x)$		$\exists x:A. B(x)$

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B}$$

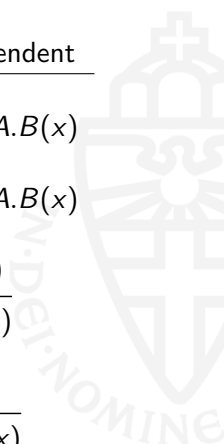
$$\langle a, b \rangle : A \times B$$

$$\frac{x : A \vdash b : B}{\lambda x:A. b : A \rightarrow B}$$

$$\frac{a : A \quad b : B(a)}{\langle a, b \rangle : \sum x:A. B(x)}$$

$$\langle a, b \rangle : \sum x:A. B(x)$$

$$\frac{x : A \vdash b : B(x)}{\lambda x:A. b : \prod x:A. B(x)}$$





## The Coq system: Prop versus Set/Type

Coq treats data types and propositions in exactly **the same way**, but they are **not identified**. (E.g. in Agda they are.)

Data types and Logical propositions live in different **type universes**

- Data types:  $A : \text{Set}$  or  $A : \text{Type}$
- Logical propositions:  $A : \text{Prop}$

Advantage: the system can extract a (correct) program from a proof by “removing everything related to Prop”.

$$P \quad : \quad \prod l : \underbrace{\text{list nat}}_{:\text{Set}}, \Sigma k : \underbrace{\text{list nat}}_{:\text{Set}}, \underbrace{(\text{Sorted}(k) \wedge \text{Perm}(l, k))}_{:\text{Prop}}$$



## The Coq system: program extraction

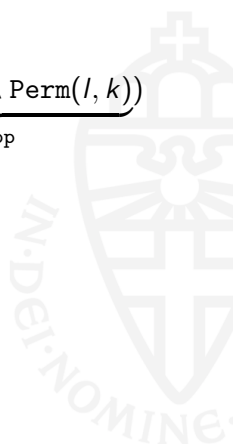
$$P : \underbrace{\prod l : \text{list nat}}_{:\text{Set}}, \underbrace{\sum k : \text{list nat}}_{:\text{Set}}, \underbrace{(\text{Sorted}(k) \wedge \text{Perm}(l, k))}_{:\text{Prop}}$$

The extraction  $E$  gives

$$E(P) : \text{list nat} \rightarrow \text{list nat}$$

Extraction can be done to

- Coq itself
- Haskell
- OCaml





## The inverse of extraction: from programs to proofs

What if I have a program that I want to prove correct?

Given

- $F : A \rightarrow B$
- $R : A \rightarrow B \rightarrow Prop$

I want to prove

$$\forall x:A, R(x, F(x))$$

This can be done (Program tactic by M. Sozeau):

- “Claim”  $F : \Pi x:A, \Sigma y:B, R(x, F(x))$ .
- Coq will interpret  $F$  as a proof-term **with holes**
- These holes are returned as **proof obligations**, that have to be dealt with by the user.





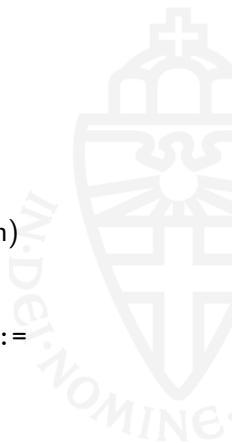
## Inductive types in Coq

```
Inductive nat : Set :=  
  | 0 : nat  
  | S : nat -> nat
```

This yields

- a type `nat` and terms `0 : nat` and `S : nat → nat`
- a function definition principle (structural recursion)
- a proof principle (induction)

```
Fixpoint plus (n m : nat) {struct n} : nat :=  
  match n with  
  | 0 => m  
  | S p => S (plus p m)  
end.
```







## Inductive types in Coq

```
Inductive nat : Set :=  
  | 0 : nat  
  | S : nat -> nat
```

This yields

- a type `nat` and terms `0 : nat` and `S : nat → nat`
- a function definition principle (structural recursion)
- a proof principle (induction)

`nat_ind`

```
: forall P : nat -> Prop,  
  P 0 -> (forall n : nat, P n -> P (S n))  
  -> forall n : nat, P n
```





## Other Inductive types in Coq

```
Inductive list (A : Type) : Type :=  
  | nil   : list A  
  | cons  : A -> list A -> list A.
```

Also relations are defined inductively:

```
Inductive le (n : nat) : nat -> Prop :=  
  | le_n   : n <= n  
  | le_S  : forall m : nat, n <= m -> n <= S m
```





## Structures are also Inductive types in Coq

```
Structure OrderedType := {  
  car :>      Type;  
  ord :       car -> car -> Prop;  
  ord_refl   : forall x, ord x x;  
  ord_symm   : forall x y, ord x y -> ord y x;  
  ord_trans  : forall x y z, ord x y -> ord y z -> ord x z}.
```

A term of type `OrderedType` is a tuple  $\langle A, R, p_1, p_2, p_3 \rangle$  with

- $A : \text{Type}$
- $R : A \rightarrow A \rightarrow \text{Prop}$
- $p_1$  proves that  $R$  is reflexive
- $p_2$  proves that  $R$  is symmetric
- $p_3$  proves that  $R$  is transitive

The labels allow to project to the appropriate field.



## Back to sorting

We can now program

$$\text{sort} : \forall A : \text{OrderedType}, \text{list } A \rightarrow \text{list } A$$

Or it can be extracted from a proof of

$$\forall A : \text{OrderedType}, \forall l : \text{list } A, \exists k : \text{list } A, (\text{Sorted}(k) \wedge \text{Perm}(l, k))$$

So: we can build very precise abstract interfaces for data structures and program with them.



## Using the rich types to guide your program

A type of vectors (list of a given length):

```
Inductive vec (A:Type): nat ->Type :=  
  vnil : vec A 0  
  | vcons : forall n : nat, A -> vec A n -> vec A (S n).
```

So  $\text{vec } A \ n$  denotes the lists over  $A$  of length  $n$ .

Defining the head of a list is annoying, because nil has no head ...

For the vector type we want

```
hd : forall (A : Type) (n : nat), vec A (S n) -> A
```

```
Definition hd (A:Type)(n:nat)(v:vec A (S n)) : A :=  
  match v with  
  | vcons n a v => a  
end.
```

**Dependently typed pattern matching:** there is no “nil case”!



## More interesting way of using the rich type system

A type of (untyped)  $\lambda$ -terms

```
Inductive term : Type :=  
| Var : nat -> term  
| Lam : nat -> term -> term  
| App : term -> term -> term.
```

Simple typed terms: term of a type in a context ( $\Gamma \vdash M : A$ )

```
Inductive type :=  
iota : type  
| arr : type -> type -> type.
```

Definition context := list type.

In order to define  $\text{Term } \Gamma A$  as the type of **terms of type  $A$  in context  $\Gamma$** .



## More interesting way of using the rich type system

```
Inductive Term : context -> type -> Type :=  
  | var : forall c t i,  
    lookup c i = Some t -> Term c t  
  | app : forall c t s,  
    Term c (arr t s) -> Term c t -> Term c s  
  | abs : forall c t s,  
    Term (t :: c) s -> Term c (arr t s).
```

Now we can prove, e.g.

```
Lemma weaken : forall (c: context)(t s:type),  
  Term c t -> Term (s :: c) t.
```



# Combining programming and proving: CoRN

In CoRN (Coq Repository at Nijmegen) we have developed a lot of results for real numbers. Goal:

- Develop abstract mathematical results
- Program with concrete mathematical data in a reliable way
- Especially: Exact Real Arithmetic

Example: Fundamental Theorem of Algebra

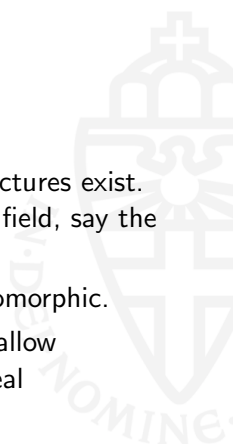
- Every polynomial over the complex numbers has a root.
- Result in (abstract) mathematics that has computational content.
- For given coefficients, a root should be computed at arbitrary precision.





## Real Numbers in Coq

- Axiomatic: a ‘Real Number Structure’ is a **Cauchy-complete Archimedean ordered field**.
- Prove FTA ‘for all real numbers structures’.
- Construct a model to show that real number structures exist. (Cauchy sequences over an Archimedean ordered field, say the rational numbers)
- Prove that any two real number structures are isomorphic.
- Construct computationally “better” models that allow infinitary approximation of real numbers (exact real arithmetic).





## Axioms for Real Numbers

The reciprocal operation is essentially **partial**

$$\frac{1}{-} : \prod x:F. x \neq 0 \rightarrow F$$

So, for  $x : F$ ,  $\frac{1}{x}$  is actually  $\frac{1}{x, H}$  with  $H : x \neq 0$ .

The term  $\frac{1}{x, H}$  depends on  $H : x \neq 0$  and we have to show that this is not a real dependency:

$$\frac{1}{x, H} = \frac{1}{x, H'}$$

for all  $H, H' : x \neq 0$ .





## Further Reading on (dependent typed programming in) Coq

- Coq in a hurry (Yves Bertot)
- Coq'Art book (Yves Bertot & Pierre Castéran)
- Certified programming with dependent types, book on-line (Adam Chlipala).
- Software Foundations course (Benjamin Pierce et al.)
- For the Agda angle: ask Wouter Swierstra (UU)
- For formalization of real programming language (C) features in Coq: ask Robbert Krebbers or Freek Wiedijk (RU)