Interactive Proving with Coq

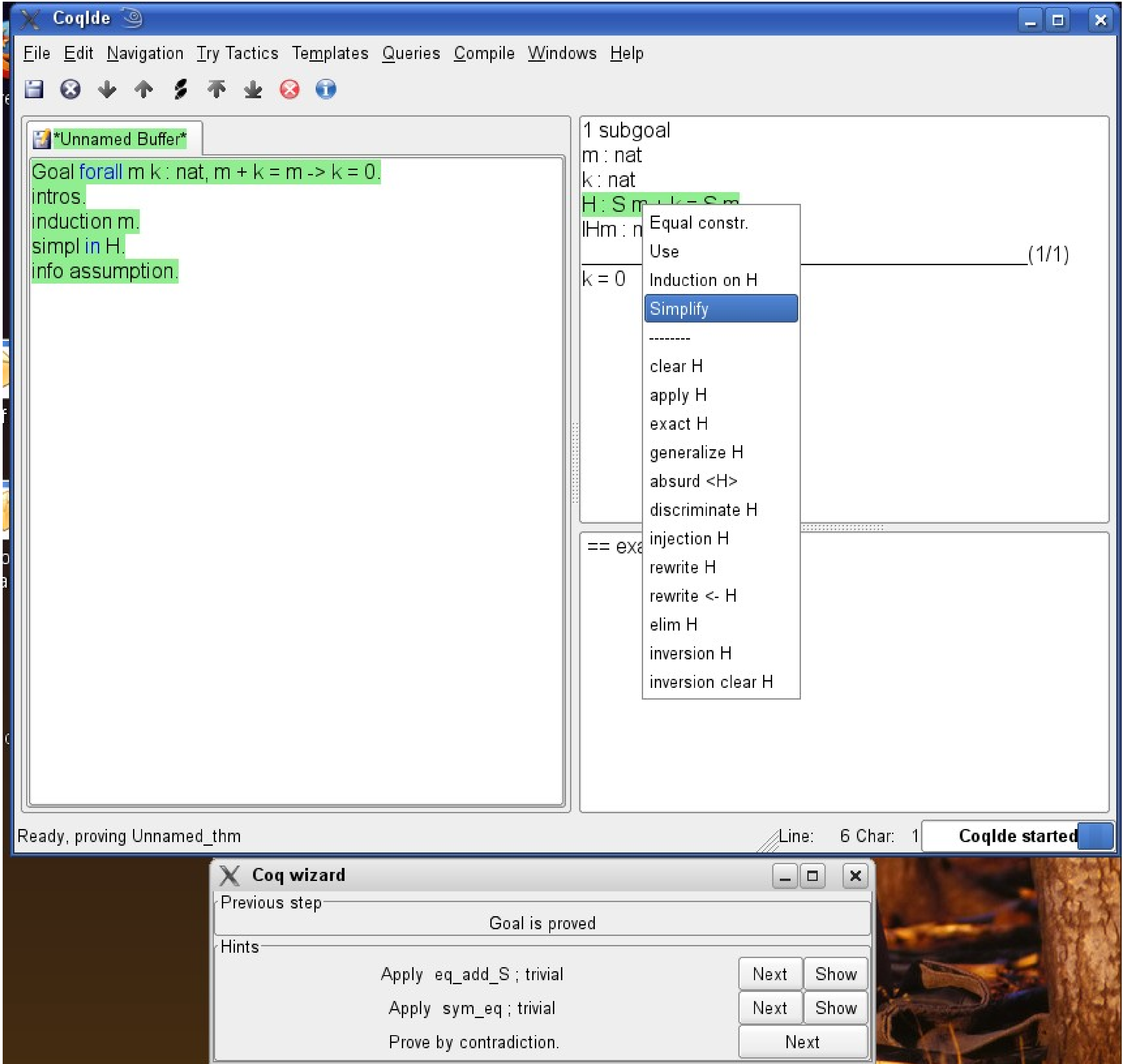
proving with mouse and some examples

CoqIde + Papuq

- CoqIde works with multiple files,
- marks and protects processed fragment of the script,
- separate windows for proof state and for inspection/search results,
- menus with inspection commands, tactics and script command templates.

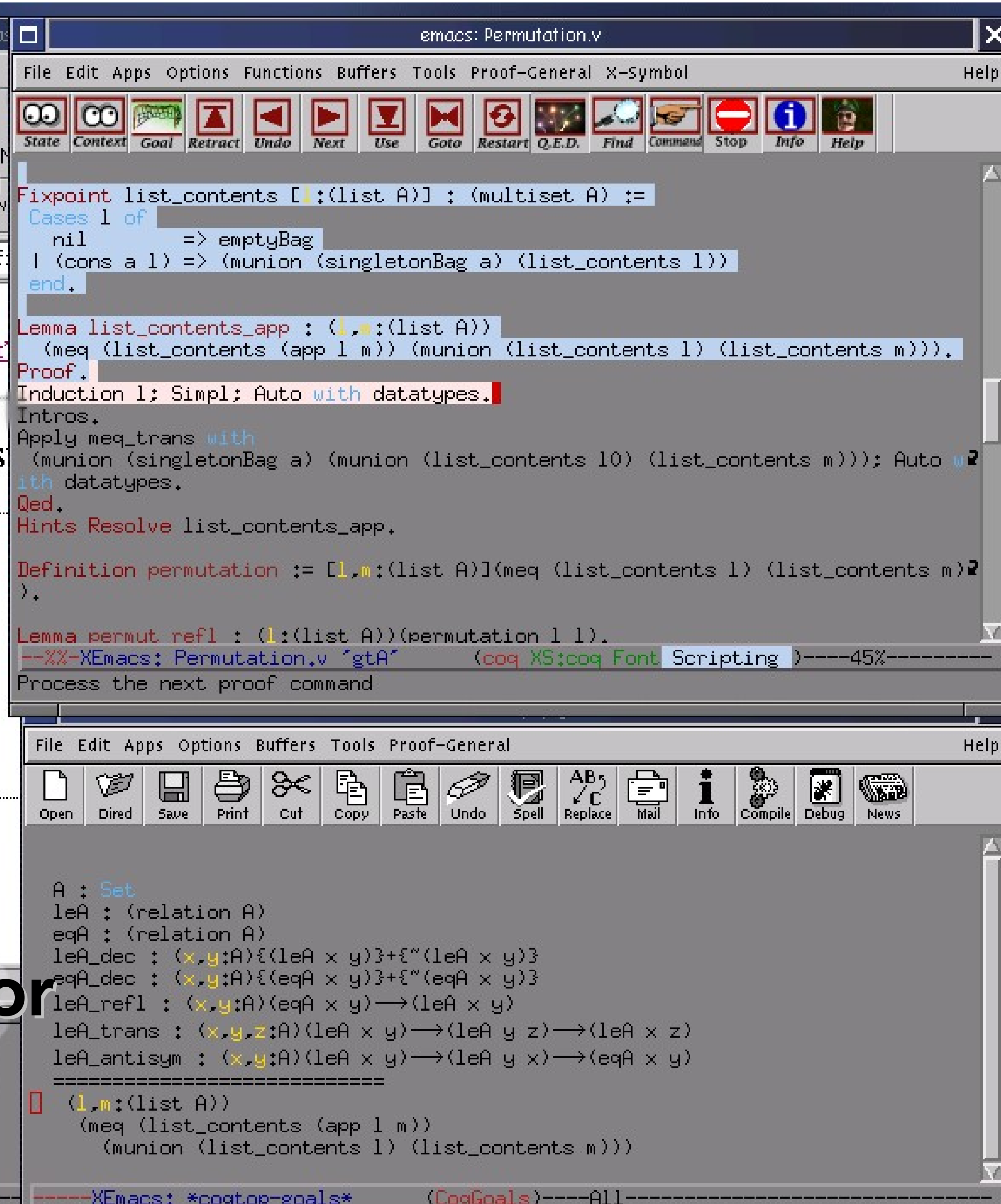
- Papuq adds a help window proposing simple actions,
- lets inspect the lemmas it proposed to apply,
- extends the context menu of CoqIde showing most worthwhile actions first,
- „wizard” automation tries out several tactics.

CoqIde + Papuq: example



Proof General

- a plugin for Emacs and Eclipse,
- works with many proof assistants,
- featureset overlapping with CoqIde (no context menu),
- proof-by-pointing for some provers (LEGO).



Proof-by-Pointing

- Not present (yet) in CoqIde, but present in earlier Coq interfaces (IDEs): CtCoq (mid-nineties) and Pcoq (2003).
- Similar to the idea of context menu present in CoqIde, performs multiple actions from a single mouse interaction.
- Already proved theorems should be accessible similarly to assumptions.
- Drag-and-drop: e.g. drag equality to the subterm to rewrite (match the terms and generate subgoals for assumptions of the equality).
- Point-and-shoot: while pointing, select the tactic to apply to the subterm after it is brought to the surface.

Proof-by-Pointing

Rules descend recursively to the position pointed by mouse.

$$\wedge\, left_1 : \frac{\boxed{A}, B, A \wedge B, \Gamma \vdash C}{\boxed{A} \wedge B, \Gamma \vdash C} \qquad \wedge\, right_1 : \frac{\Gamma \vdash \boxed{A} \qquad \Gamma \vdash B}{\Gamma \vdash \boxed{A} \wedge B}$$

$$\wedge\, left_2 : \frac{A, \boxed{B}, A \wedge B, \Gamma \vdash C}{A \wedge \boxed{B}, \Gamma \vdash C} \qquad \wedge\, right_2 : \frac{\Gamma \vdash A \qquad \Gamma \vdash \boxed{B}}{\Gamma \vdash A \wedge \boxed{B}}$$

$$\vee\, left_1 : \frac{\boxed{A}, A \vee B, \Gamma \vdash C \qquad B, A \vee B, \Gamma \vdash C}{\boxed{A} \vee B, \Gamma \vdash C} \qquad \vee\, right_1 : \frac{\Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{A} \vee B}$$

$$\vee\, left_2 : \frac{A, A \vee B, \Gamma \vdash C \qquad \boxed{B}, A \vee B, \Gamma \vdash C}{A \vee \boxed{B}, \Gamma \vdash C} \qquad \vee\, right_2 : \frac{\Gamma \vdash \boxed{B}}{\Gamma \vdash A \vee \boxed{B}}$$

$$\supset left_1 : \frac{A \supset B, \Gamma \vdash \boxed{A} \qquad B, A \supset B, \Gamma \vdash C}{\boxed{A} \supset B, \Gamma \vdash C} \qquad \supset right_1 : \frac{\boxed{A}, \Gamma \vdash B}{\Gamma \vdash \boxed{A} \supset B}$$

$$\supset left_2 : \frac{A \supset B, \Gamma \vdash A \qquad \boxed{B}, A \supset B, \Gamma \vdash C}{A \supset \boxed{B}, \Gamma \vdash C} \qquad \supset right_2 : \frac{A, \Gamma \vdash \boxed{B}}{\Gamma \vdash A \supset \boxed{B}}$$

$$\forall\, left : \frac{\boxed{A[x \backslash e]}, \forall x\, A, \Gamma \vdash C}{\forall x\, \boxed{A}, \Gamma \vdash C} \qquad \forall\, right : \frac{\Gamma \vdash \boxed{A[x \backslash c]}}{\Gamma \vdash \forall x\, \boxed{A}}$$

type-specific
induction scheme:

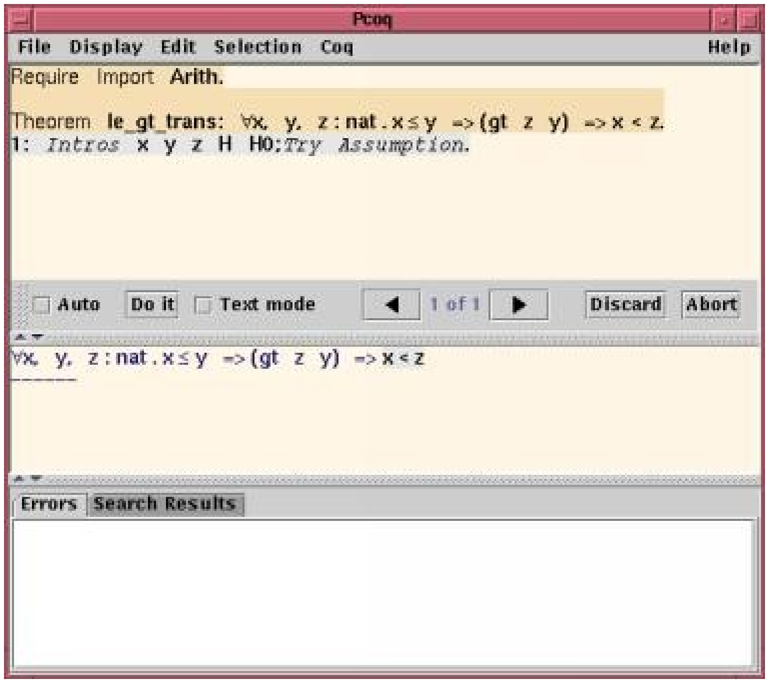
$$\frac{\Gamma \vdash P(0) \qquad P(n), \Gamma \vdash P(n+1)}{\Gamma \vdash \forall \boxed{n : int}\ P(n)}$$

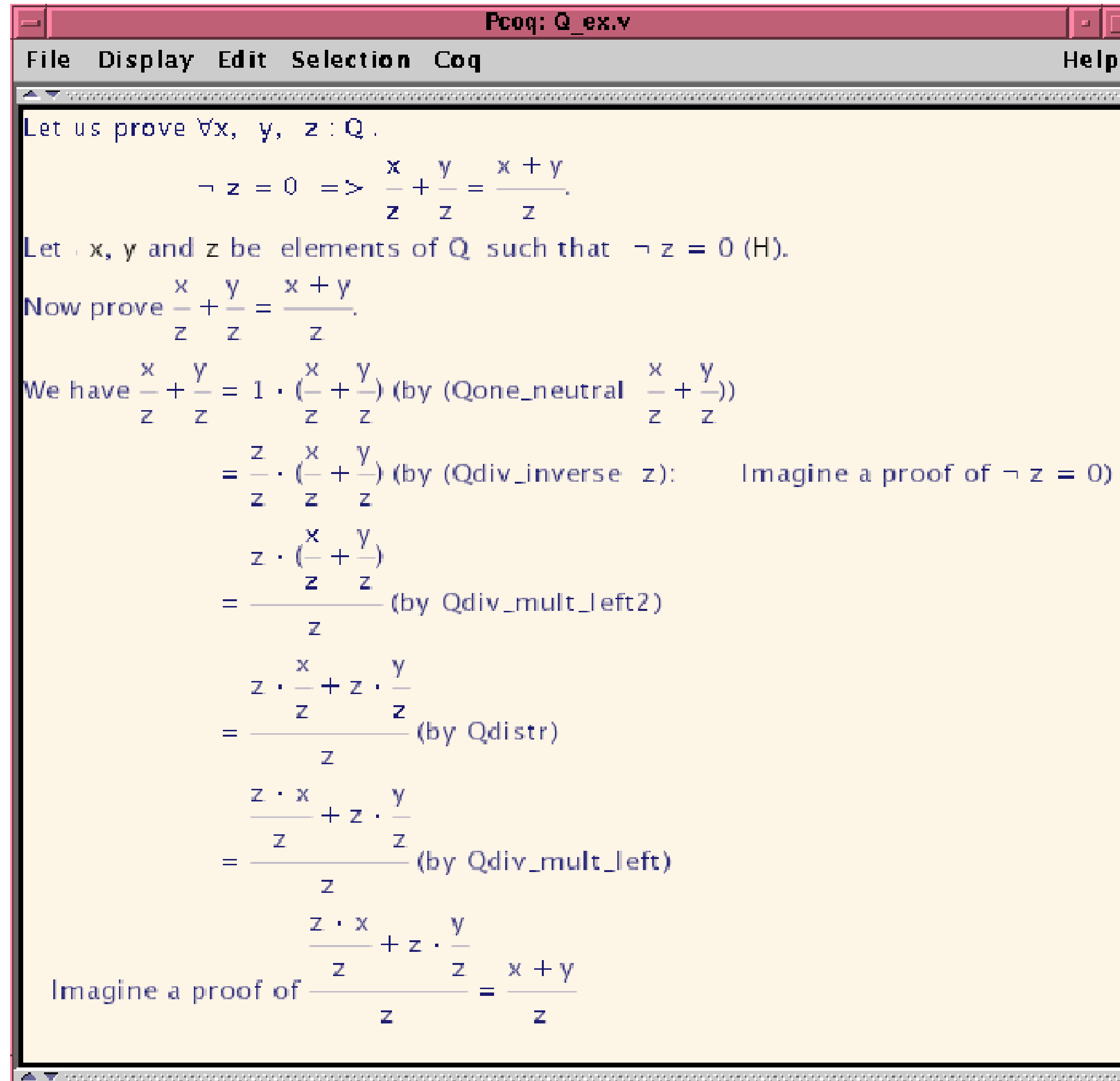
$$\exists\, left : \frac{\boxed{A[x \backslash c]}, \exists x\, A, \Gamma \vdash}{\exists x\, \boxed{A}, \Gamma \vdash C} \qquad \exists\, right : \frac{\Gamma \vdash \boxed{A[x \backslash e]}}{\Gamma \vdash \exists x\, \boxed{A}}$$

Proof-by-Pointing: 5-click example

- *Goal (p a \/ q b) /\ (forall x, p x -> q x) -> (exists x, q x).*
- Point to *p a* in conclusion.
  - Two subgoals with *p a* resp. *q b*, and *forall x, p x -> q x* in assumptions, the one with *p a* selected.
- Point to *p x* from *forall x, p x -> q x* in assumptions.
  - *p x* is automatically proved, and thus *q a* is added to assumptions.
- Twice: point to *q x* in conclusion.

Pcoq and Proof Presentation





Extracting Text from Proof

Rules for abstraction

$(\lambda l{:}A_{Type}.\,M)_\tau$	$\triangleright$	Let l: A M We have proved τ
$(\lambda h{:}A_{Prop}.\,M)_\tau$	$\triangleright$	Assume A (h) M We have proved τ
$(\lambda x{:}A_{Set}.\,M)_\tau$	$\triangleright$	Consider an arbitrary x in A M We have τ, since x is arbitrary

Rules for application

$(M_{\forall x:P.\,Q}\,N)_\tau$	$\triangleright$	M In particular τ
$(M_{P \supset Q}\,N)_\tau$	$\triangleright$	- N - M We deduce τ

Rules for identifiers

h_τ	$\triangleright$	By h we have τ
T_τ	$\triangleright$	Using T we get τ

Analogous (compact) rules are built for repeated abstractions and applications.

Extracting Text from Proof

Rules for introduction theorems

$(\mathtt{C}intro\ M^1 \cdots M^n\ N^1 \cdots N^i)_\tau$	$\triangleright$	$-N^1$ $\vdots$ $-N^i$ So by definition of $\mathtt{C}$ we have τ
$i = 0$ $(\mathtt{C}intro\ M^1 \cdots M^n)_\tau$	$\triangleright$	By definition of $\mathtt{C}$ we have τ
$i = 1$ $(\mathtt{C}intro\ M^1 \cdots M^n N)_\tau$	$\triangleright$	N By definition of $\mathtt{C}$ we have τ

Rules for elimination theorems

$(\mathtt{C}elim\ M^1 \cdots M^n\ N^1 \cdots N^i\ P)_\tau$	$\triangleright$	P Therefore by definition of $\mathtt{C}$, to prove τ we have i cases: Case$_1$: N^1 $\vdots$ Case$_i$: N^i So we have τ
$i = 0$ $(\mathtt{C}elim\ M^1 \cdots M^n P)_\tau$	$\triangleright$	P, by definition of $\mathtt{C}$ there is a contradiction So we can assert τ
$i = 1$ $(\mathtt{C}elim\ M^1 \cdots M^n\ N\ P)_\tau$	$\triangleright$	P Therefore by definition of $\mathtt{C}$ to prove τ N So we have τ

Extracting Text from Proof: examples

Let U: *Type*
Let $P, Q: U \to Prop$
Let a: U
Assume $(P\,a)$ (h) and $\forall x{:}U.\,(P\,x) \supset (Q\,x)$ (h_0)
  Applying h_0 with h we get $(Q\,a)$
We have proved $(P a) \supset (\forall x{:}U.\,(P\,x) \supset (Q\,x)) \supset (Q\,a)$
We have proved $\forall U{:}Type.\,\forall P, Q{:}U \to Prop.\,\forall a{:}U.\,(P\,a) \supset (\forall x{:}U.\,(P\,x) \supset (Q\,x)) \supset (Q\,a)$

By definition of $\mathbb{N}$ to prove $\forall n{:}\mathbb{N}.\,0 \leq n$, we have two cases:
Case$_1$:
  By definition of $\leq$ we have $0 \leq 0$
Case$_2$:
  Let $m{:}\mathbb{N}$
  Assume $0 \leq m$ (h)
    From h and the definition of $\leq$, we have $0 \leq (\mathrm{Suc}\,m)$
  We have proved $0 \leq m \supset 0 \leq (\mathrm{Suc}\,m)$
  We have proved $\forall m{:}\mathbb{N}.\,0 \leq m \supset 0 \leq (\mathrm{Suc}\,m)$
So we have $\forall n{:}\mathbb{N}.\,0 \leq n$

Let A, B : *Prop*
Assume $A \vee B$ (h)
  Assume A (i)
    From i and the definition of $\vee$, we have $B \vee A$
  -We have proved $A \supset B \vee A$
  Assume B (j)
    From j and the definition of $\vee$, we have $B \vee A$
  -We have proved $B \supset B \vee A$
  -We have h
  Applying $\vee elim$ we get $B \vee A$
We have proved $A \vee B \supset B \vee A$
We have proved $\forall A, B{:}Prop.\,A \vee B \supset B \vee A$

Coq specification language: Gallina

Some of the syntax:

```
term        ::=  forall binderlist , term
             |   fun binderlist => term
             |   fix fix_bodies
             |   cofix cofix_bodies
             |   let ident_with_params := term in term
             |   let fix fix_body in term
             |   let cofix cofix_body in term
             |   let ( [name , … , name] ) [dep_ret_type] := term in term
             |   if term [dep_ret_type] then term else term
             |   term : term
             |   term -> term
             |   term arg … arg
             |   match match_item , … , match_item [return_type] with
                   [[|] equation | … | equation] end
fix_bodies  ::=  fix_body
             |   fix_body with fix_body with … with fix_body for ident
fix_body    ::=  ident binderlet … binderlet [{struct ident}] [: term] := term
dep_ret_type ::= [as name] return term
match_item  ::=  term [as name] [in term]
equation    ::=  mult_pattern | … | mult_pattern => term
```

Coq specification language: Gallina

- Type hierarchy: proofs in formulas, formulas in ***Prop***, other types (specifications) of non-types in ***Set***, ***Prop*** and ***Set*** in ***Type***$_{(0)}$, ***Type***$_{(i)}$ in ***Type***$_{(i+1)}$
- Products ***forall*** *x* **:** *A***,** *B* are written *A* **->** *B* when *x* doesn't occur in *B*.
- *(x* **:** *A* **:=** *B)*... is a shortcut for ***let*** *x* **:** *A* **:=** *B* ***in*** ...
- Subterms replaced by **_** or declared as implicit are (tried to be) inferred by type inference.
- return_type is the type of a pattern matching term, it can depend on the matched value, or its type:

```
Definition sym_equal (A:Type) (x y:A) (H:eq A x y) : eq A y x :=
  match H in eq _ _ z return eq A z x with
  | refl_equal => refl_equal A x   end.
```

Gallina's command language: The Vernacular

sentence ::=

| (**Axiom** | **Conjecture** | **Parameter**[**s**] | **Variable**[**s**] | **Hypothes**[**is**|**es**])
(ident … ident : term | binder … binder)**.**

| (**Definition** | **Let**) ident_with_params **:=** term**.**

| [**Co**]**Inductive** ind_body **with** … **with** ind_body**.**

| [**Co**]**Fixpoint** fix_body **with** … **with** fix_body**.**

| (**Theorem** | **Lemma** | **Definition**) ident [binderlet … binderlet] : term**.**
[**Proof**. proof_script (**Qed**.|**Defined**.|**Admitted**.)]

| **Record** ident [binderlet … binderlet] **:** sort **:=** [ident] **{**[name [**:** term] [**:=** term] **;** … **;** name [**:** term] [**:=** term]] **}.**

| **Function** ident binder…binder **{**(**struct** ident | **measure** term ident | **wf** term ident)**}** **:** term **:=** term**.**

| **Section** ident**.** | **End** ident**.**

| **Module** [**Import** | **Export**] ident [module_bindings] (**:** | **<:**) module_type**.**

| **Coercion** qualid **:** class1 **>->** class2**.**

The Vernacular

- Declaration introduces a name with a given type.
- Definition gives a name for a term.
- ind_body ::= ident [binderlet … binderlet] **:** term **:=** [[**|**] ident [binderlet … binderlet] [**:** term] **|** … **|** ident [binderlet … binderlet] [**:** term]]
- ***Fixpoint*** introduces recursive definition or inductive proof decreasing w.r.t. the argument in **{*struct*** *ident***}**
- Definitions can also be built interactively by tactics.
- ***Record***s are syntax sugar for one-constructor inductive definitions, known from programming langs.
- ***Let*** definitions are local to sections.

The Vernacular

- ***Function*** is a generalization of ***Fixpoint*** that besides the function, generates
  - an induction principle that reflects the recursive structure of the function
  - its fixpoint equality (if recursive)
  - graph (relation) of the function (silently).
- Non-recursive arguments should go first.
- Limited pattern-matching (currently dependent cases not supported).
- ***measure*** and ***wf*** allow to easily define a function decreasing on given ordering relation, generate proof obligations for monotonicity (and well-foundedness).

The Vernacular

- A module with parameters is a functor.
- Libraries (directories) and modules (files and modules in files) form a common hierarchy.
- Implicit coercions allow to write:
  - *f a* where *f:forall x:A, B* and *a:A'* when *A'* can be seen in some sense as a subtype of A.
  - *x:A* when *A* is not a type, but can be seen in a certain sense as a type: set, group, category etc.
  - *f a* when *f* is not a function, but can be seen in a certain sense as a function: bijection, functor, any structure morphism etc.
- For example, *forall (x1 : A1)..(xn : An)(y: C x1..xn), D u1..um* can coerce an object *t:C t1..tn* to *f t1..tn : D u1..um*: we declare *Coercion f : C >-> D.*

Coq Libraries

- Coq is easily extensible with user-provided notations.
- Initial library contains logical operators, basic datatypes: product *prod*, *sum*, specification *sig* (object with a proof of its property), *sumbool* (non-dependent sum of *Prop*s: a choice between two formulas), *nat*.
- Standard library contains useful basic logical and arithmetic (Peano, integers, reals) facts, and datatypes: lists, sets, maps.
- Everything else is in the contributions library.

Coq Standard Library

- **Logic** Classical logic and dependent equality
- **Arith** Basic Peano arithmetic
- **NArith** Basic positive integer arithmetic
- **ZArith** Basic relative integer arithmetic
- **Bool** Booleans (basic functions and results)
- **Lists** Monomorphic and polymorphic lists (basic functions and results), Streams (infinite sequences defined with co-inductive types)
- **Sets** Sets (classical, constructive, finite, infinite, power set, etc.)
- **FSets** Specification and implementations of finite sets and finite maps (by lists and by AVL trees)
- **IntMap** Representation of finite sets by an efficient structure of map (trees indexed by binary integers).
- **Reals** Axiomatization of real numbers (classical, basic functions, integer part, fractional part, limit, derivative, Cauchy series, power series and results,...)
- **Relations** Relations (definitions and basic results).
- **Sorting** Sorted list (basic definitions and heapsort correctness).
- **Strings** 8-bits characters and strings
- **Wellfounded**Well-founded relations (basic results).

The Vernacular: search

- ***Print*** *qualid*. displays name's associated term and its type.
- ***Check*** *term*. displays term's type (in current context=i.c.c.).
- ***Search*** *qualid*. displays all theorems i.c.c. (=a.t.i.c.c.) whose conclusion head is qualid.
- ***SearchAbout*** *qualid*. displays a.t.i.c.c. containing qualid.
- ***SearchPattern*** *term*. displays a.t.i.c.c. with conclusion matching the given term.
  - Coq < SearchPattern (_ + _ = _ + _).
  - plus_comm: forall n m : nat, n + m = m + n
  - plus_Snm_nSm: forall n m : nat, S n + m = n + S m ...
- ***SearchRewrite*** *term*. displays a.t.i.c.c. with conclusion being equality, its one side matching the given term.
  - Coq < SearchRewrite (_ + _ + _).
  - plus_assoc: forall n m p : nat, n + (m + p) = n + m + p ...

The Vernacular: more commands

- ***Load*** *ident*. loads a source file ident.v.
- ***Require [Import]*** *ident*. loads and opens a compiled module ident.vo. (Not visible outside.)
- ***Print Modules***. shows the currently loaded/opened modules.
- ***Qed***.|***Save.*** finishes proof defining an opaque constant (it cannot be unfolded or proven different to another opaque constant).
- ***Save*** *ident*. as above for goals started with ***Goal*** *term*.
- ***Defined***. finishes proof defining a transparent constant.
- ***Admitted***. gives up proving and declares the goal as an axiom.
- ***Abort***. aborts proving and discards the goal.

Coq Selected Tactics

- ***refine*** *term* allows to give an exact proof but still with some holes noted _.
- ***eapply*** *term* tries to unify current goal with the conclusion of given term, turns uninstantiated variables in premises into existential meta-variables.
- ***compute*** performs beta delta iota zeta reductions.
- ***functional induction*** *(qualid term … term)* performs case analysis and induction following the definition of a function.
- ***inversion*** *ident* „destructs” ident generating subgoals for each constructor of inductive predicate which is the type of ident, and discards the subgoals where „unpacked” assumptions are contradictory.

Coq Automation

- *[**e**]**auto** [**with** ident … ident | **with** *] [**using** lemma … lemma]* Prolog-like (depth-first) resolution procedure: reduces goal to an atomic one (intros), tries tactics associated with goal head in turn (lower cost tactics first; theorems used with ***apply***); recurses to subgoals. Either solves the goal completely or leaves intact. idents name hint databases, * means uses all hints, lemmas are additional hints. ***eauto*** uses ***eapply*** (unification rather than pattern-matching).

- ***firstorder*** *[tactic] [**with** ident … ident] [**using** ident … ident]* performs first-order reasoning, applies tactic to subgoals where logical reasoning fails, extends the proof search environment with **with**-ident lemmas and lemmas from **using**-ident hint databases.

Coq Automation

- ***congruence*** for equational reasoning.
- ***autorewrite with*** *ident* … *ident [**using** tactic] [**in** qualid]* applies a rewrite system joining the *ident*s rewriting rule bases; applies *tactic* (if given) after each rewrite step; performs rewritings in assumption *qualid* (if given).
- omega solves Presburger arithmetic for nat and Z (binary integers).
- ***ring*** does associative-commutative rewriting in ring and semi-ring structures. It is implemented directly in Coq (reflection). It works by registering ring properties for given type, rules for evaluating coefficients, and a morphism from coefficients to the ring carrier type.

Example: mergesort

- Sorting
  - First, specify sorting lists of natural numbers through a predicate:
    - sort : list nat -> list nat -> Prop.
- Merging
  - Define a function *merge: list nat -> list nat -> list nat* such that the following lemma holds:
    - Lemma merge_and_sort : forall l l', sorted l -> sorted l' -> sort (l++l') (merge l l').
  - Prove this property.

Example: mergesort

- Balanced binary trees
  - Consider the type of binary trees whose nodes are labeled in type *N* and leaves in type *L*:
    - Inductive tree(N L:Type):Type := Leaf : L -> tree N L | Node : N -> tree N L -> tree N L -> tree N L.
  - We now consider trees whose nodes contain boolean values and leaves an optional value of type *L*, i.e trees of type *tree bool (option L)*. Complete the following definition:
    - Inductive balanced(L:Type): tree bool (option L) -> nat -> Prop :=
- Insertion in a balanced tree
  - Define a function:
    - insert (L:Type): L -> tree bool (option L)) -> tree bool (option L)
  - such that the insertion of *l:L* into a balanced tree ressults in a balanced tree

Example: mergesort

- Building a balanced tree from a list
  - Define a function such that *share _ ls* returns a balanced tree containing all the elements of ls
    - share (L:Type) : L -> tree bool (option L)
- We now have all material for building the function
    - mergesort : list nat -> list nat
  - let *l* be a list of natural numbers
  - build a balanced tree whose leaves are labled with the elements of *l*
  - flatten this tree, using *merge* to combine the leaves of the left and right subtrees
- Prove the theorem:
    - Theorem mergesort_ok : forall l, sort l (mergesort l).

Extraction of programs

- Output languages: OCaml, Haskell and Scheme
- ***Extraction*** *qualid*. extracts one constant or module.
- ***Recursive Extraction*** *qualid* ... *qualid*. extracts together with all dependencies.
- ***Extraction*** *„file” qualid* ... *qualid*. as above into one monolithic file.
- ***Extraction Library*** *ident*. extracts the whole library into ML module ident.ml.
- ***Recursive Extraction Library*** *ident*. extracts the library into ident.ml and all libraries/modules it depends on into their files.

Extraction of programs

- ***Extraction Language*** *(**Ocaml** | **Haskell** | **Scheme** | **Toplevel**).* *Toplevel* is pseudo-OCaml, doesn't change names so fails OCaml syntax, works only for toplevel.
- ***Extract Constant*** *qualid* **=>** *string*. extracts *qualid* as *string*, which can be an identifier or a quoted (arbitrary) string. (Defines *qualid* as *string*.)
- ***Extract Inlined Constant*** *qualid* **=>** *string*. as above, but inlines the string for each occurrence of *qualid*.
- ***Extract Constant*** *qualid string … string* **=>** *string*. extracts type schemes (e.g. Y „`a" „`b" => „`a * `b")
- ***Extract Inductive*** *qualid* **=>** *string [string …string].* extracts inductive definitions.
    - Extract Inductive sumbool => "bool" ["true" "false"].

Sources

- „*Proof by Pointing*” Yves Bretot, Gilles Kahn, Laurent Thery, 1994
- „*Mathematics and Proof Presentation in Pcoq*” Ahmed Amerkad, Yves Bretot, Loic Pottier, Laurence Rideau
- „*Extracting Text from Proof*” Yann Coscoy, Gilles Kahn, Laurent Thery
- „*The Coq Proof Assistant Reference Manual Version 8.1*” The Coq Development Team: LogiCal Project
- An exercise from the „*Coq'Art*” book webpage, Pierre Castéran, Julien Forest, based on an exercise from Epigram tutorial