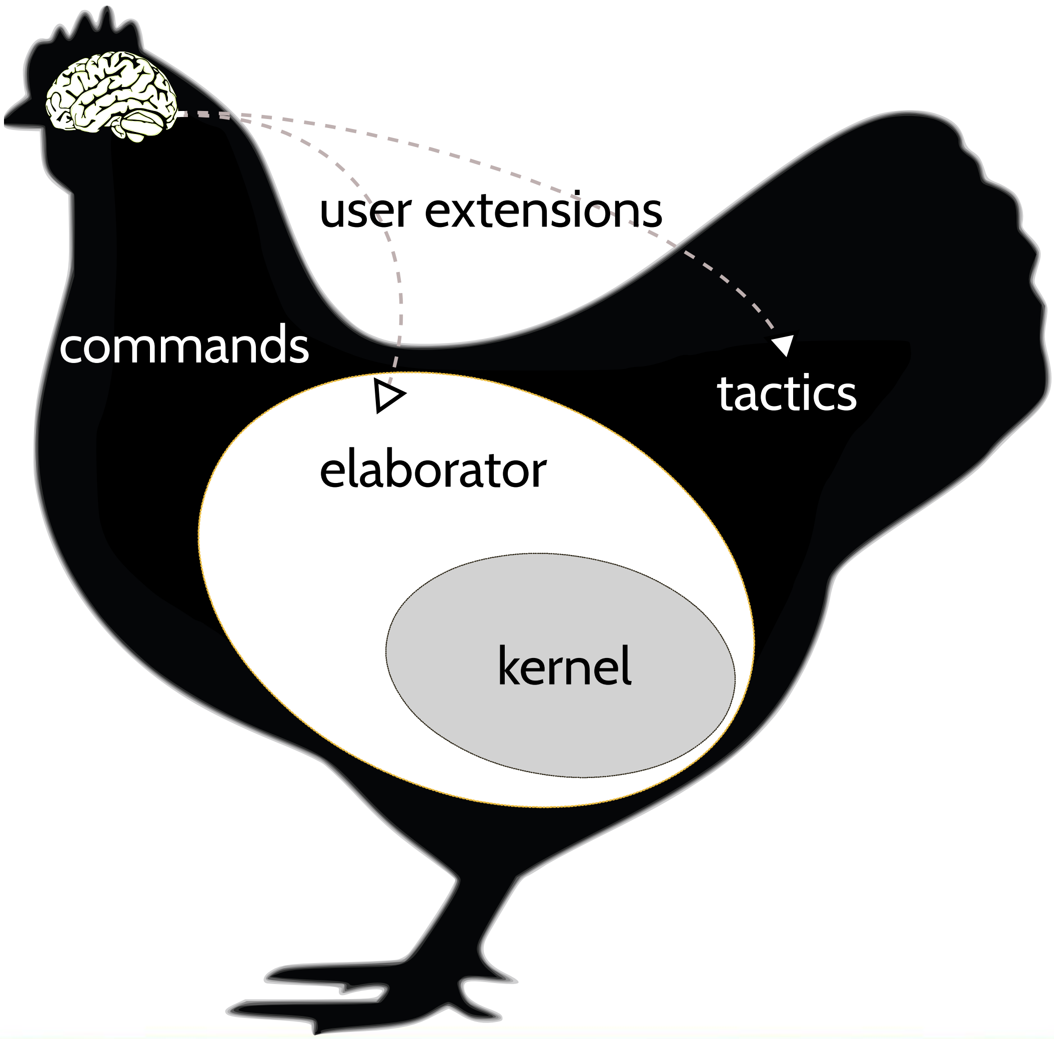
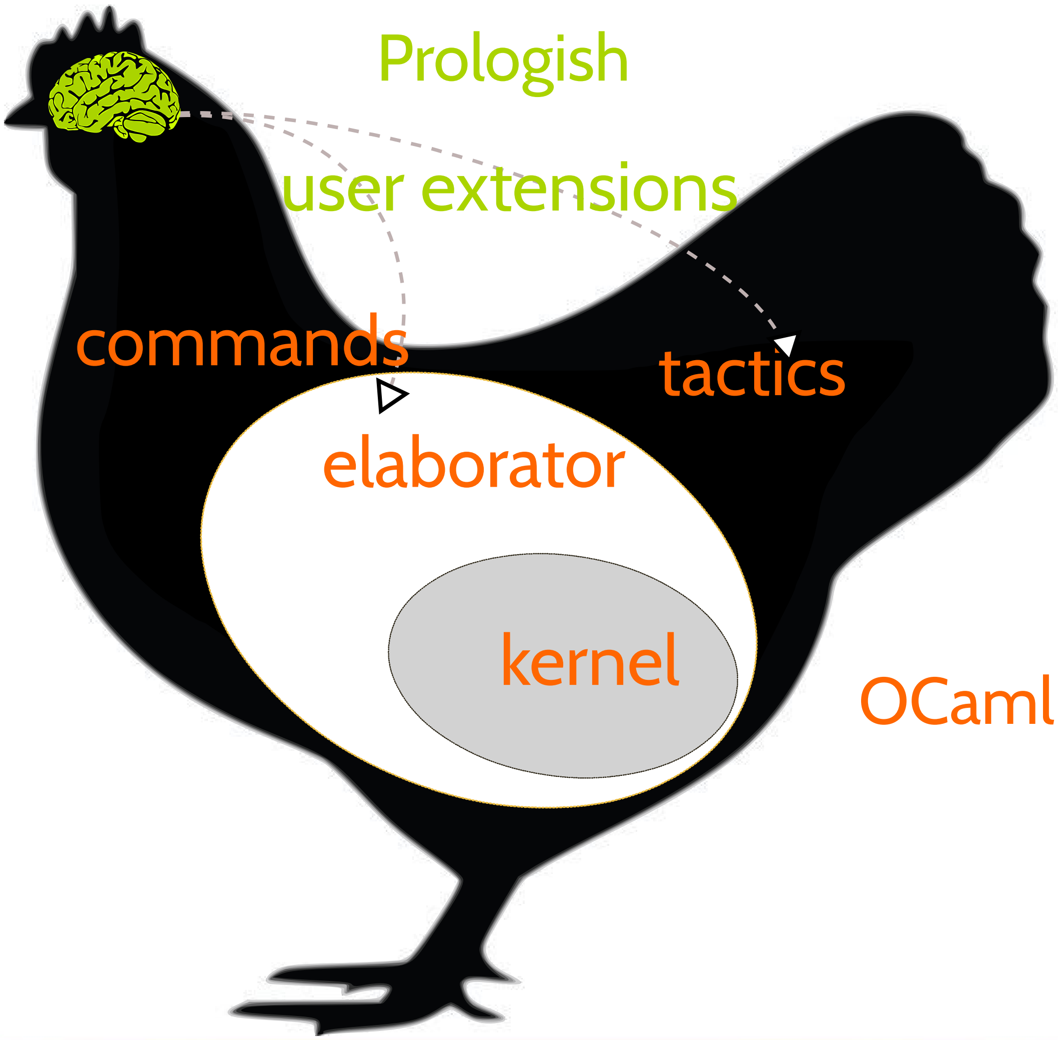
Elpi: an extension language with binders and unification variables

Enrico Tassi (w. help C.Sacerdoti Coen & D.Miller)
ML Workshop 2018

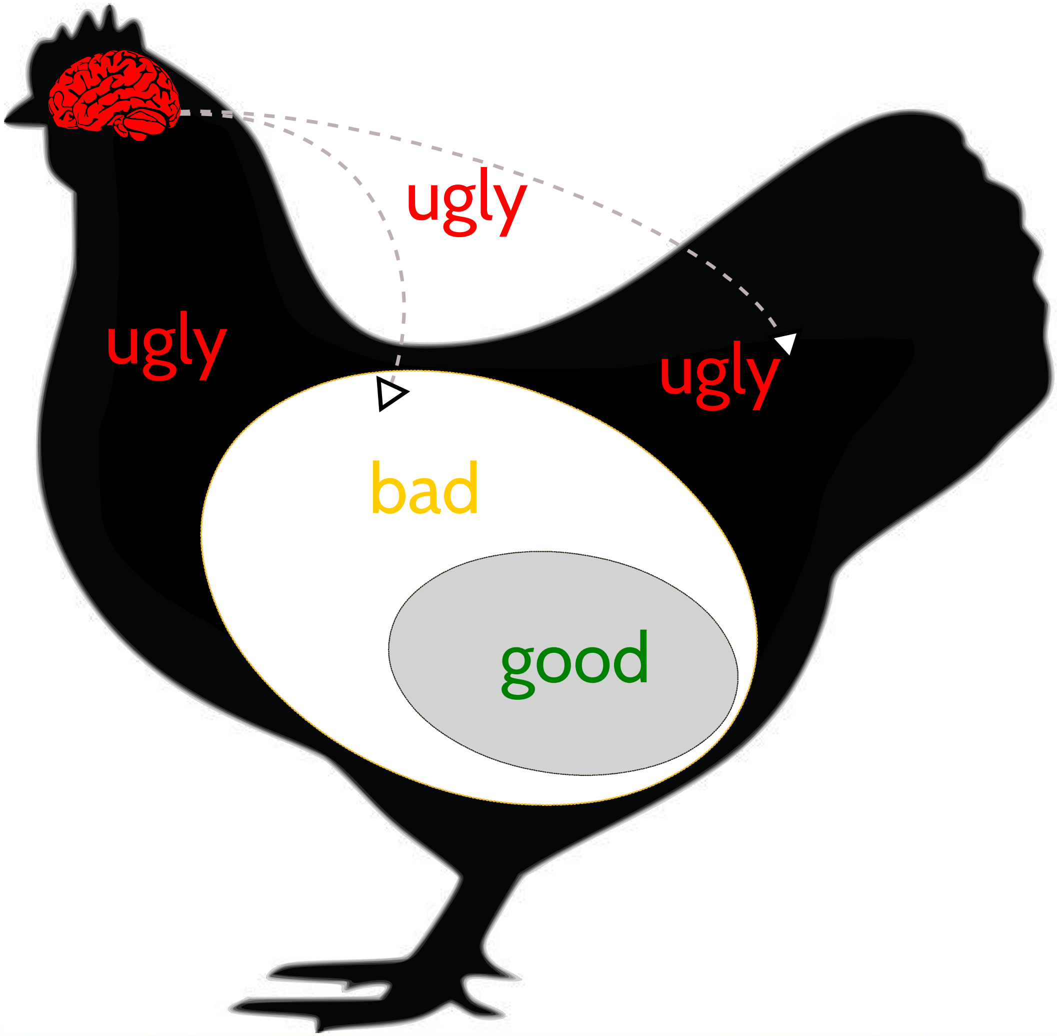
Architecture of Coq



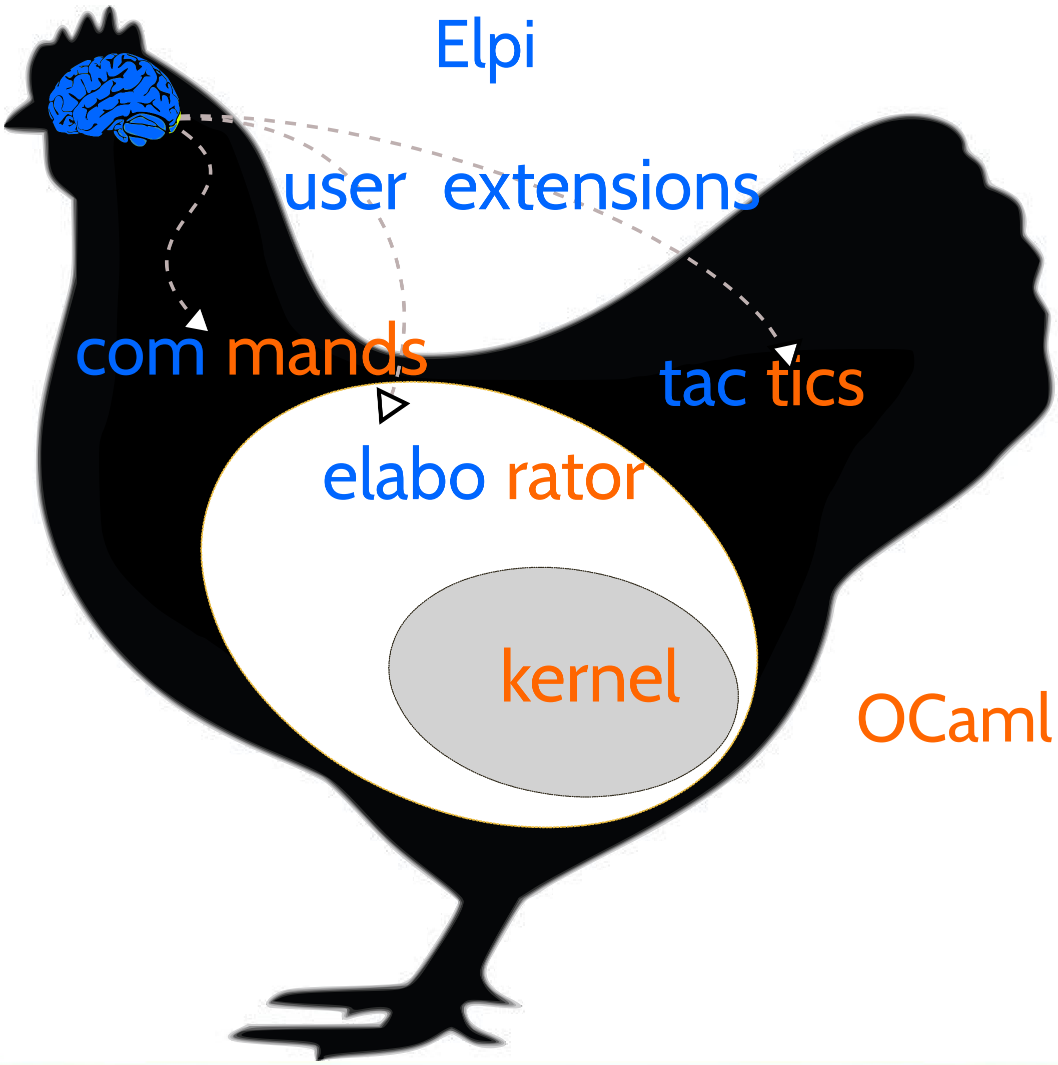
Language wise



Quality wise



Plan


Elpi
user extensions
com mands
tac tics
elabo rator
kernel
OCaml

This talk is about Elpi, that is...

- An extension language
  - its interpreter comes as a library
  - with an API/FFI to write glue code
- A very high level language
  - binders
  - unification variables
  - constraints
- LGPL, by C.Sacerdoti Coen and myself

Elpi = λProlog + CHR

Outline

- Elpi 101
  - λProlog 101: type checker for $\lambda_{\rightarrow}$
  - λProlog + CHR 101: even & odd
- Code: toyml.ml + w.elpi
  - HM type inference + equality types
- Demo: coq-elpi / derive.eq
- Implementation of Elpi in OCaml

λProlog 101

$$
\begin{aligned}
e &= x \\
&\mid e_1 e_2 \\
&\mid \lambda x.e
\end{aligned}
$$

$$
\begin{aligned}
\tau &= C \\
&\mid \tau \to \tau
\end{aligned}
$$

```
% HOAS of terms

type app term → term → term.

type lam (term → term) → term.

% HOAS of types

type arrow ty → ty → ty.
```

```
% Example: identity function
lam (x\ x)
% Example: fst
lam x\ lam y\ x
```

λProlog 101

```
pred of i:term, o:ty.
```

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

```
of (app H A) T :-
  of H (arrow S T), of A S.
```

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

```
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
```

% Convention
X % universally quantified around the rule
X_i % not quantified (existentially quantified, globally)

λProlog 101

$$\vdash \lambda x.\lambda y.x\ y : Q$$

Goal

```
of (lam x\ lam y\ app x y) Q0.
```

Program

```
of (app H A) T :- of H (arrow S T), of A S.
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
```

Assignments

Q_0 = ...

λProlog 101

$$\vdash \lambda x.\lambda y.x\ y : Q$$

Goal

```
of ((x\ lam y\ app x y) c1) T1.
```

Program

```
of (app H A) T :- of H (arrow S T), of A S.
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
of c1 S1.
```

Assignments

```
Q0 = arrow S1 T1
F1 = (x\ lam y\ app x y)
```

λProlog 101

$$\vdash \lambda x.\lambda y.x\ y : Q$$

Goal

```
of (lam y\ app c1 y) T1.
```

Program

```
of (app H A) T :- of H (arrow S T), of A S.
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
of c1 S1.
```

Assignments

```
Q0 = arrow S1 T1
F1 = (x\ lam y\ app x y)
```

λProlog 101

$$\vdash \lambda x.\lambda y.x\ y : Q$$

Goal

```
of ((y\ app c1 y) c2) T2.
```

Program

```
of (app H A) T :- of H (arrow S T), of A S.
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
of c1 S1.
of c2 S2.
```

Assignments

```
Q0 = arrow S1 (arrow S2 T2)
F1 = (x\ lam y\ app x y)
F2 = (y\ app c1 y)
```

λProlog 101

$$\vdash \lambda x.\lambda y.x\ y : Q$$

Goal

```
of (app c₁ c₂) T₂.
```

Program

```
of (app H A) T :- of H (arrow S T), of A S.
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
of c₁ S₁.
of c₂ S₂.
```

Assignments

```
Q₀ = arrow S₁ (arrow S₂ T₂)
F₁ = (x\ lam y\ app x y)
F₂ = (y\ app c₁ y)
```

λProlog 101

$$\vdash \lambda x.\lambda y.x\ y : Q$$

Goal

```
of c1 (arrow S3 T2).
of c2 S3.
```

Program

```
of (app H A) T :- of H (arrow S T), of A S.
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
of c1 S1.
of c2 S2.
```

Assignments

Q_0 = arrow S_1 (arrow S_2 T_2)
F_1 = (x\ lam y\ app x y)
F_2 = (y\ app c_1 y)
H_3 = c_1
A_3 = c_2

λProlog 101

$$\vdash \lambda x.\lambda y.x\ y : Q$$

Goal

```
of c2 S3.
```

Program

```
of (app H A) T :- of H (arrow S T), of A S.
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
of c1 (arrow S3 T2).
of c2 S2.
```

Assignments

Q_0 = arrow (arrow S_3 T_2) (arrow S_2 T_2)
F_1 = (x\ lam y\ app x y)
F_2 = (y\ app c_1 y)
H_3 = c_1 S_1 = (arrow S_3 T_2)
A_3 = c_2

λProlog 101

$$\vdash \lambda x.\lambda y.x\ y : (S \to T) \to S \to T$$

Goal

Program

```
of (app H A) T :- of H (arrow S T), of A S.
of (lam F) (arrow S T) :-
  pi x\ of x S => of (F x) T.
of c1 (arrow S2 T2).
of c2 S2.
```

Assignments

Q_0 = arrow (arrow S_2 T_2) (arrow S_2 T_2)

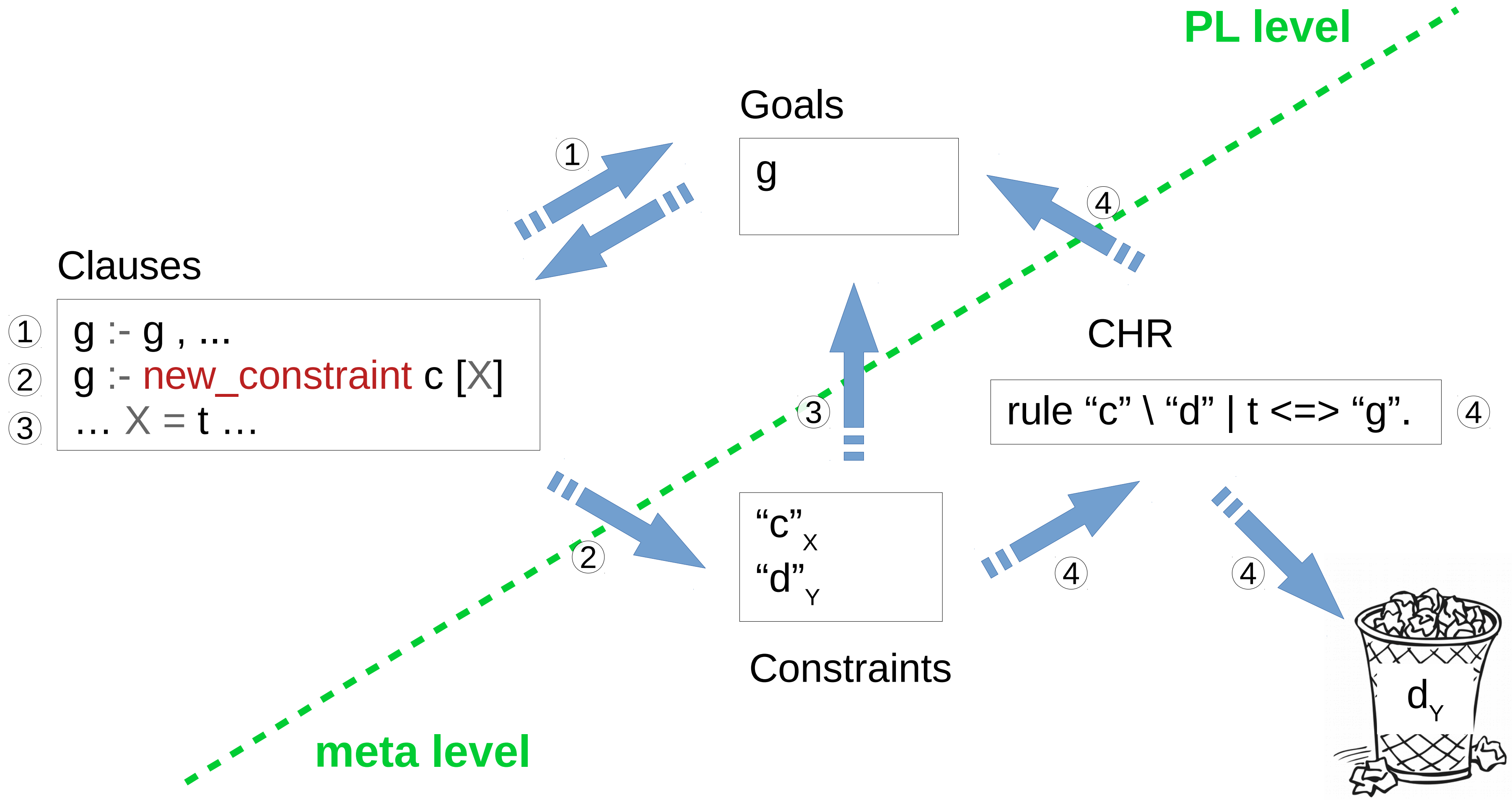
F_1 = (x\ lam y\ app x y)

F_2 = (y\ app c_1 y)

H_3 = c_1 S_1 = (arrow S_3 T_2)

A_3 = c_2 S_3 = S_2

λProlog + CHR 101



λProlog + CHR 101

```
type zero nat.  type succ nat -> nat.

pred odd i:nat.  pred even i:nat.  pred double i:nat, o:nat.

even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.

even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].

double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.

double X Y :- var X, new_constraint (double X Y) [X].

constraint even odd double {
  rule (even X) (odd X) <=> fail.
  rule (double _ X) <=> (even X).
}
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

```
even X
X = succ Y
not (double Z Y)
```

Constraint store

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

```
X = succ Y
not (double Z Y)
```

Constraint store

even F_X

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

```
even (succ Y)
not (double Z Y)
```

Constraint store

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

```
odd Y
not (double Z Y)
```

Constraint store

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

```
not (double Z Y)
```

Constraint store

odd $\mathbf{F}_{Y}$

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

not ( )

Constraint store

odd $\mathbf{F}_Y$

double $\mathbf{F}_Z$ $\mathbf{F}_Y$

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

```
not (even Y)
```

Constraint store

odd $\mathbf{F}_Y$

double $\mathbf{F}_Z$ $\mathbf{F}_Y$

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

not ( )

Constraint store

odd $\mathbf{F}_Y$

double $\mathbf{F}_Z$ $\mathbf{F}_Y$

even $\mathbf{F}_Y$

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

not (fail)

Constraint store

odd $\mathbf{F}_Y$

double $\mathbf{F}_Z$ $\mathbf{F}_Y$

even $\mathbf{F}_Y$

Program

```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

λProlog + CHR 101

even X, X = succ Y, not (double Z Y)

Goals

Constraint store

odd $\mathbf{F}_Y$

Program

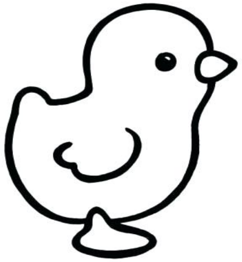
```
even zero.
odd (succ X) :- even X.
even (succ X) :- odd X.
even X :- var X, new_constraint (even X) [X].
odd X :- var X, new_constraint (odd X) [X].
double zero zero.
double (succ X) (succ (succ Y)) :- double X Y.
double X Y :- var X, new_constraint (double X Y) [X].
```

Rules

```
(even X) (odd X) <=> fail.
(double _ X) <=> (even X).
```

Elpi = λProlog + CHR

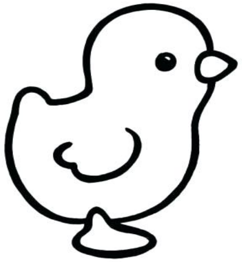
- λProlog for …
  - backward reasoning, search
  - ✔ programming with binders recursively
- CHR for …
  - forward reasoning
  - ✔ manipulate (frozen) unification variables
  - ✔ handle metadata on unification variables



Toyml: syntax

$$
\begin{aligned}
e = {} & x \\
& \mid e_1 e_2 \\
& \mid \lambda x.e \\
& \mid \textbf{let } x = e_1 \textbf{ in } e_2 \\
& \mid e_1 = e_2
\end{aligned}
$$

$$
\begin{aligned}
\text{mono } \tau = {} & \alpha \\
& \mid \tau \to \tau \\
& \mid \text{boolean} \\
& \mid \text{integer} \\
& \mid \text{pair } \tau\ \tau \\
& \mid \text{list } \tau \\
\text{poly } \rho = {} & \tau \\
& \mid \forall \alpha.\rho \\
& \mid \forall \overline{\alpha}.\rho
\end{aligned}
$$



Typing rules

$$\frac{x : \rho \in \Gamma \quad \rho \sqsubseteq_\Theta \tau}{\Gamma \vdash_\Theta x : \tau}$$

$$\frac{\Gamma \vdash_\Theta e_1 : \tau \to \tau' \quad \Gamma \vdash_\Theta e_2 : \tau}{\Gamma \vdash_\Theta e_1\ e_2 : \tau'}$$

$$\frac{\Gamma, x : \tau \vdash_\Theta e : \tau'}{\Gamma \vdash_\Theta \lambda x.e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash_\Theta e_1 : \tau \quad \Gamma, x : \overline{\Gamma}_\Theta(\tau) \vdash_\Theta e_2 : \tau'}{\Gamma \vdash_\Theta \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau'}$$

$$\frac{\Gamma \vdash_\Theta e_1 : \tau \quad \Gamma \vdash_\Theta e_2 : \tau \quad \overline{eq}_\Theta(\tau)}{\Gamma \vdash_\Theta e_1 = e_2 : \mathrm{boolean}}$$

Type schemas: introduction

$$\overline{\Gamma}_{\Theta}(\tau) = \forall \overrightarrow{\hat{\alpha}}.\tau$$

$$\hat{\alpha} = \overline{\alpha} \text{ if } \alpha \in \Theta$$

$$\hat{\alpha} = \alpha \text{ otherwisie}$$

$$\text{where } \alpha \in \text{free}(\tau) - \text{free}(\Gamma)$$

$$\text{free}(\Gamma) = \bigcup_{x:\rho\in\Gamma} \text{free}(\rho)$$

$$\ldots$$

Type schemas: elimination

$$\frac{}{\tau \sqsubseteq_\Theta \tau}$$

$$\frac{\rho[\alpha := \tau'] \sqsubseteq_\Theta \tau}{\forall \alpha . \rho \sqsubseteq_\Theta \tau}$$

$$\frac{\rho[\alpha := \tau'] \sqsubseteq_\Theta \tau \quad \overline{eq}_\Theta(\tau')}{\forall \overline{\alpha} . \rho \sqsubseteq_\Theta \tau}$$

$$\begin{aligned}
\overline{eq}_\Theta(\text{boolean}) & \\
\overline{eq}_\Theta(\text{integer}) & \\
\overline{eq}_\Theta(\text{list } \tau) & \text{ if } \overline{eq}_\Theta(\tau) \\
\overline{eq}_\Theta(\text{pair } \tau_1 \ \tau_2) & \text{ if } \overline{eq}_\Theta(\tau_1) \text{ and } \overline{eq}_\Theta(\tau_2) \\
\overline{eq}_\Theta(\alpha) & \text{ if } \alpha \in \Theta
\end{aligned}$$

Code

- toyml.ml
- w.elpi

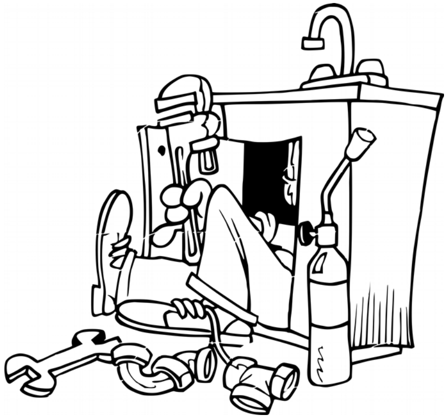
Was it W ?

- W is in usually presented in terms of *unify*, *newvar*; threading a substitution...
- w.elpi is closer to a declarative presentation but its operational meaning is W

```
let rec append l1 l2 =
  match l1 with
  | x :: xs → x :: append xs l2
  | [] → l2
```

Demo: coq-elpi

- Integration:
  - {{ quotataions }} and `pphints`
- CHR:
  - model uniqueness of typing
- Example:
  - Elpi derive.eq tree



Elpi: implementation

- The first prototype of Elpi was pure, and slow
- Elpi uses ML’s references (mutable)
  - closer to standard Prolog technology (stack, heap, trail)
  - easy to align with the GC of the host application
  - surprisingly, not a source of bugs

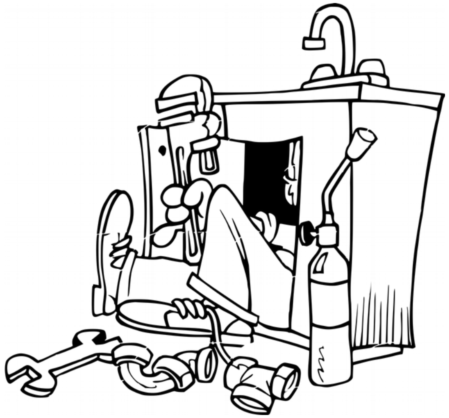
Conclusion

- ML is great!
- ML + Elpi is even better ;-)

https://github.com/LPCIC/elpi

Elpi ! logic programming

- high level with an operational meaning
  - yummy!
- fact: 90% compute, 10% search
  - wrong default
- extensibility of programs: clauses
  - a miracle



Elpi: implementation

- Binder mobility
- GADT

Binder mobility

- λProlog is presented as “locally nameless”
  - De Buijn indexes for bound variables
  - De Bruijn “levels” for nominal constants

```
w (lam F) (arrow S T) :- pi c\ w c S => w (F c) T.

?- w (lam x\ app f x) T  % w (lam _\ app f 1) T
```

- F c involves a β_0 reduction
  - (_\ app f 1) c → app f c

Binder mobility

- In Elpi
  - De Bruijn “levels” for everything

```
w (lam F) (arrow S T) :- pi c\ w c S => w (F c) T. % w (F 1) T

?- w (lam x\ app f x) T  % w (lam _\ app f 1) T
```

- F 1 involves no substitution
  - (_\ app f 1) 1 → app f 1

FFI

- OCaml's GADTs to describe the type of the ML code
  - no type conversion/checking boilerplate
  - mixes FFI call and projection of the result

```
MLCode(Pred("coq.env.const",
    In(gref,  "GR",
    Out(term, "Bo", Out(term, "Ty",
    Easy ("reads the type Ty and the body Bo of GR. ")))),
  (fun gr bo ty ~depth:_ ->
      let t = if ty = Discard then None else Some (embed ... gr) in
      let b = ... in
      ?: b +? t)),
  DocAbove);
```

```
main :- coq.locate “plus” GR, coq.env.const GR TY _, ...
```