



Elpi: the language you already know without knowing it

Enrico Tassi – Xmas 2023 / Nantes

The workflow of a poor soul

- Code, play, validate...

```
| Lambda (name,src,tgt) ->  
  let sigma, src_ty = typecheck env sigma src in  
  let sigma, decl = (LocalAssum (name, src_ty)) in  
  let env1 = push_rel decl env in  
  let sigma, tgt_ty = typecheck env1 sigma tgt in  
  sigma, mkProd (name, src_ty, tgt_ty)
```

- ... and then communicate ...

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

The workflow of a poor soul

- ... cheating ...

All rules work on a signature Σ , containing previously defined constants, metavariables, and guarded constants. In other words, we can write all judgements on the form $\langle \Sigma \rangle J \Rightarrow \langle \Sigma' \rangle$. To make the rules easier to read we first define a set of operations reading and modifying the signature and when presenting the algorithm simply write J for the judgement above. In rules with multiple premisses the signature is threaded top-down, left-to-right. For instance,

$$\frac{P_1 \quad P_2 \quad P_3}{J} \quad \text{is short-hand for} \quad \frac{\langle \Sigma_1 \rangle P_1 \Rightarrow \langle \Sigma_2 \rangle \quad \langle \Sigma_2 \rangle P_2 \Rightarrow \langle \Sigma_3 \rangle \quad \langle \Sigma_3 \rangle P_3 \Rightarrow \langle \Sigma_4 \rangle}{\langle \Sigma_1 \rangle J \Rightarrow \langle \Sigma_4 \rangle}$$

The workflow of a poor soul

- ... scaring

$$(I_l : \Pi \overrightarrow{x_l} : \overrightarrow{F_l} . \Pi \overrightarrow{y_r} : \overrightarrow{G_r} . s) \in \text{Env}$$

$$(k_j : \Pi \overrightarrow{x_l} : \overrightarrow{F_l} . \Pi y_{n_j}^j : T_{n_j}^j . I_l \overrightarrow{x_l} \overrightarrow{M_r^j}) \in \text{Env} \quad j \in \{1 \dots n\}$$

$$\Sigma \rightsquigarrow \Sigma \cup \{\Gamma \vdash ?u_i : F_i[\overrightarrow{x_{i-1}} / ?u_{i-1}]\} \quad i \in \{1 \dots l\}$$

$$\Sigma \rightsquigarrow \Sigma \cup \{\Gamma \vdash ?v_i : G_i[\overrightarrow{x_l} / ?u_l; \overrightarrow{y_{i-1}} / ?v_{i-1}]\} \quad i \in \{1 \dots r\}$$

$$\Gamma \vdash t : I_l \overrightarrow{?u_l} \overrightarrow{?v_r} \overset{\mathcal{R}^\Downarrow}{\rightsquigarrow} t'$$

$$G'_i = G_i[\overrightarrow{x_l} / ?u_l] \quad i \in \{1 \dots r\}$$

$$T'^j_i = T^j_i[\overrightarrow{x_l} / ?u_l] \quad j \in \{1 \dots n\}, i \in \{1 \dots n_j\}$$

$$M'^j_i = M^j_i[\overrightarrow{x_l} / ?u_l] \quad j \in \{1 \dots n\}, i \in \{1 \dots r\}$$

$$\Sigma \rightsquigarrow \Sigma \cup \{\Gamma' \vdash ?_1 : \mathbf{Type}_\top\}$$

$$\Gamma \vdash T : \Pi \overrightarrow{y_r} : \overrightarrow{G_r} . \Pi x : I_l \overrightarrow{?u_l} \overrightarrow{y_r} . ?_1 \overset{\mathcal{R}^\Downarrow}{\rightsquigarrow} T'$$

$$(s, \Phi(?_1)) \in \text{elim}(\text{PTS})$$

$$\begin{array}{c} \Gamma; \overrightarrow{y_{n_j-1}^j} : \overrightarrow{P_{n_j-1}^j} \vdash P_{n_j}^j \overset{?}{=} T_{n_j}^j \overset{\mathcal{U}}{\rightsquigarrow} \quad j \in \{1 \dots n\} \\ \Gamma; \overrightarrow{y_{n_j}^j} : \overrightarrow{P_{n_j}^j} \vdash t_j : T' M'^j_r (k_j \overrightarrow{?u_l} \overrightarrow{y_{n_j}^j}) \overset{\mathcal{R}^\Downarrow}{\rightsquigarrow} t'_j \quad j \in \{1 \dots n\} \\ \hline (\overset{\mathcal{R}^\Uparrow}{\rightsquigarrow} \text{-match}) \quad \Gamma \vdash \left(\begin{array}{l} \mathbf{match } t \text{ in } I_l \text{ return } T \\ [k_1 (\overrightarrow{y_{n_1}^1} : \overrightarrow{P_{n_1}^1}) \Rightarrow t_1 \mid \dots \mid k_n (\overrightarrow{y_{n_n}^n} : \overrightarrow{P_{n_n}^n}) \Rightarrow t_n] \end{array} \right) \overset{\mathcal{R}^\Uparrow}{\rightsquigarrow} \\ \left(\begin{array}{l} \mathbf{match } t' \text{ in } I_l \text{ return } T' \\ [k_1 (\overrightarrow{y_{n_1}^1} : \overrightarrow{P_{n_1}^1}) \Rightarrow t'_1 \mid \dots \mid k_n (\overrightarrow{y_{n_n}^n} : \overrightarrow{P_{n_n}^n}) \Rightarrow t'_n] \end{array} \right) : T' \overrightarrow{?v_r} t' \end{array}$$

Inference rule(s)!

- One key in the Odd Order Theorem

$$\frac{\text{nat} \sim \text{EQ.obj nat_EQty} \quad ?x \sim \text{nat_EQty}}{\text{nat} \sim \text{EQ.obj } ?x}$$

$$\frac{t_1 \sim \text{EQ.obj } ?y \quad t_2 \sim \text{EQ.obj } ?z \quad ?x \sim \text{pair_EQty } ?y ?z}{t_1 * t_2 \sim \text{EQ.obj } ?x}$$

Can we turn λ into a programming language?

- ADTs with binders and holes
- A context (for the bound variables)
- A sigma (for the hole's metadata)

Elpi = λ Prolog + CHR

- λ Prolog for ...
 - Context and substitution/assignment
 - ✓ programming with binders recursively
- CHR for ...
 - State/metadata management
 - ✓ manipulate unification variables
 - ✓ non-local deductions

Outline

- Elpi 101
 - λ Prolog 101: type checker for λ_{\rightarrow}
 - λ Prolog + CHR 101: even & odd
- Demo: j.elpi / hm.elpi
 - HM type inference + equality types
- Coq-Elpi

λ Prolog 101

% HOAS of terms

$e = x$

| $e_1 e_2$

| $\lambda x. e$

type app term \rightarrow term \rightarrow term.

type lam (term \rightarrow term) \rightarrow term.

% HOAS of types

$\tau = C$

| $\tau \rightarrow \tau$

type arrow ty \rightarrow ty \rightarrow ty.

% Example: identity function

lam (x\ x)

% Example: fst

lam x\ lam y\ x

λ Prolog 101

pred of i:term, o:ty.

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

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

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

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.

% Convention

X % universally quantified around the rule

X_i % not quantified (existentially quantified, globally)

λ Prolog 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

$Q_0 = \dots$

λ Prolog 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)
```

λ Prolog 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)
```

λ Prolog 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)
```

λ Prolog 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)
```

λ Prolog 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

```
Q0 = arrow S1 (arrow S2 T2)  
F1 = (x\ lam y\ app x y)  
F2 = (y\ app c1 y)  
H3 = c1  
A3 = c2
```


λ Prolog 101

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

Goal

of $c_2 \ S_3$.

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  $c_1$  (arrow  $S_3 \ T_2$ ).  
of  $c_2 \ S_2$ .
```

Assignments

```
 $Q_0 = \text{arrow} (\text{arrow } S_3 \ T_2) (\text{arrow } S_2 \ T_2)$   
 $F_1 = (x \backslash \text{lam } y \backslash \text{app } x \ y)$   
 $F_2 = (y \backslash \text{app } c_1 \ y)$   
 $H_3 = c_1 \quad S_1 = (\text{arrow } S_3 \ T_2)$   
 $A_3 = c_2$ 
```

λ Prolog 101

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

Goal

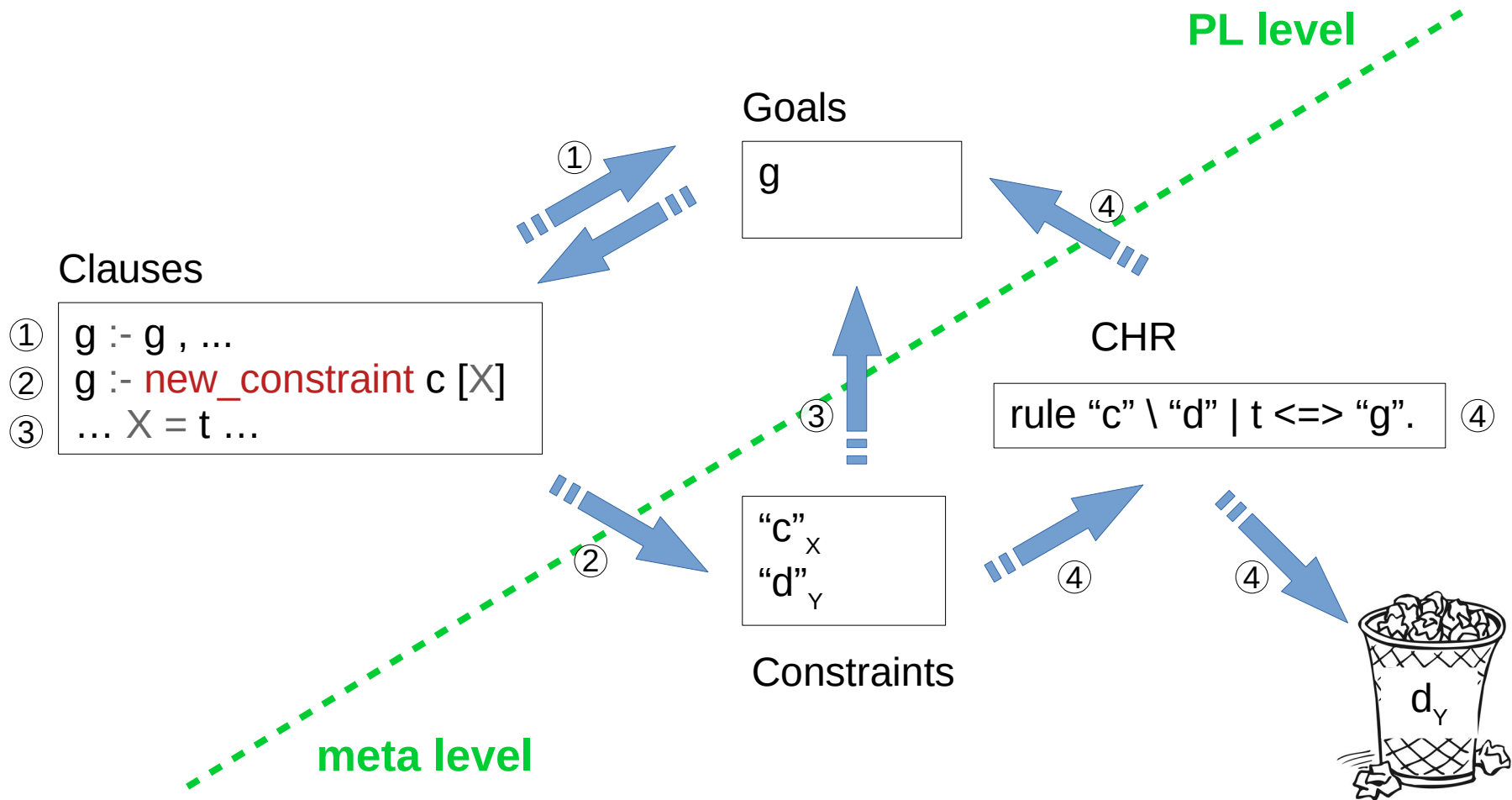
Program

```
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

```
Q0 = arrow (arrow S2 T2) (arrow S2 T2)  
F1 = (x\ lam y\ app x y)  
F2 = (y\ app c1 y)  
H3 = c1      S1 = (arrow S3 T2)  
A3 = c2      S3 = S2
```

λ Prolog + 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 FX
```

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 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 FY  
double FZ FY
```

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 FY  
double FZ FY
```

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 FY  
double FZ FY  
even FY
```

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 FY`
`double FZ FY`
`even FY`

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 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 = λ Prolog + CHR

- λ Prolog for ...
 - Context and substitution/assignment
 - ✓ programming with binders recursively
- CHR for ...
 - State/metadata management
 - ✓ manipulate unification variables
 - ✓ non-local deductions

Demo: HM in Elpi

Hindley–Milner type system

🌐 3 languages ▾

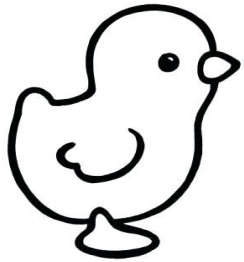
Article [Talk](#)

Read [Edit](#) [View history](#) [Tools](#) ▾

From Wikipedia, the free encyclopedia

A **Hindley–Milner (HM) type system** is a classical [type system](#) for the [lambda calculus](#) with [parametric polymorphism](#). It is also known as **Damas–Milner** or **Damas–Hindley–Milner**. It was first described by J. Roger Hindley^[1] and later rediscovered by Robin Milner.^[2] Luis Damas contributed a close formal analysis and proof of the method in his PhD thesis.^{[3][4]}

Among HM's more notable properties are its [completeness](#) and its ability to infer the [most general type](#) of a given program without programmer-supplied [type annotations](#) or other hints. [Algorithm W](#) is an efficient [type inference](#) method in practice and has been successfully applied on large code bases, although it has a high theoretical [complexity](#).^[note 1] HM is preferably used for [functional languages](#). It was first implemented as part of the type system of the programming language [ML](#). Since then, HM has been extended in various ways, most notably with [type class](#) constraints like those in [Haskell](#).



HM: syntax

$e = x$

| $e_1 e_2$

| $\lambda x. e$

| **let** $x = e_1$ **in** e_2

| $e_1 = e_2$

mono $\tau = \alpha$

| $\tau \rightarrow \tau$

| boolean

| integer

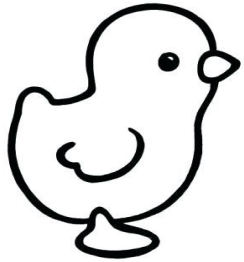
| pair $\tau \tau$

| list τ

poly $\rho = \tau$

| $\forall \alpha. \rho$

| $\forall \bar{\alpha}. \rho$



Typing rules

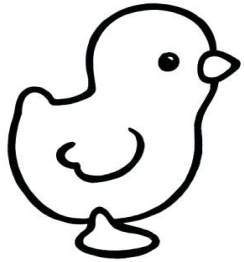
$$\frac{x : \rho \in \Gamma \quad \rho \sqsubseteq_{\Theta} \tau}{\Gamma \vdash_{\Theta} x : \tau}$$

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

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

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

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



Type schemas: introduction

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

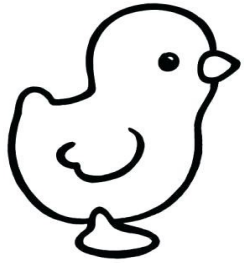
$$\hat{\alpha} = \bar{\alpha} \text{ if } \alpha \in \Theta$$

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

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

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

...



Type schemas: elimination

$$\overline{\tau \sqsubseteq_{\Theta} \tau}$$

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

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

$\overline{eq}_{\Theta}(\text{boolean})$

$\overline{eq}_{\Theta}(\text{integer})$

$\overline{eq}_{\Theta}(\text{list } \tau) \text{ if } \overline{eq}_{\Theta}(\tau)$

$\overline{eq}_{\Theta}(\text{pair } \tau_1 \ \tau_2) \text{ if } \overline{eq}_{\Theta}(\tau_1) \text{ and } \overline{eq}_{\Theta}(\tau_2)$

$\overline{eq}_{\Theta}(\alpha) \text{ if } \alpha \in \Theta$

Elpi \rightarrow Coq-Elpi

- Elpi is ***not*** a general purpose PL
 - It is a DSL
- Elpi finds his place inside a larger software, e.g. Coq
 - Only used for tasks that fit well
 - Needs (a lot of) glue code

Coq-Elpi

- Elpi is 12KLOC, Coq-Elpi is 12KLOC

```
MLCode(Pred("coq.typecheck",
  CIn(term, "T",
    CInOut(B.ioargC term, "Ty",
      InOut(B.ioarg B.diagnostic, "Diagnostic",
        Full (proof_context, {|...doc...|}))),
    (fun t ety diag ~depth proof_context _ state ->
      try
        let sigma = get_sigma state in
        let sigma, ty = Typing.type_of proof_context.env sigma t in
        (*...*)
        let state, assignments = set_current_sigma ~depth state sigma in
        state, !: ty +! B.mkOK, assignments
      with Pretype_errors.PretypeError (env, sigma, err) ->
        let error = string_of_ppcmds proof_context.options @@ Himgsg.explain_
          state, ? : None +! B.mkERROR error, [])),
    DocAbove);
```

Good, Bad and Ugly

- Users!
- Complete apps written in Coq-Elpi
- Debugger
- Constraints are hard to program with
- Tutorials but no refman
- No proper language-server

Future

- Type Class solver / Elaborator
 - HB hides the inference engine of MC2
- Automation (eg Diaframe)
 - Indexing techniques from ATP
 - Memoization techniques from LP

Elpi ! logic programming

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

