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,body) ->
    let sigma, _ = typecheck env sigma src in
    let decl = LocalAssum(name,src) in
    let env = push_rel decl env in
    let sigma, tgt = typecheck env sigma body in
    sigma, Prod(name,src,tgt)
```

- … and then communicate ...

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

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 $\langle\Sigma\rangle\ J \Longrightarrow \langle\Sigma'\rangle$. 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,

$$\frac{\begin{array}{l} P_1 \\ P_2 \quad\quad P_3 \end{array}}{J} \quad \text{is short-hand for} \quad \frac{\begin{array}{l} \langle\Sigma_1\rangle\ P_1 \Longrightarrow \langle\Sigma_2\rangle \\ \langle\Sigma_2\rangle\ P_2 \Longrightarrow \langle\Sigma_3\rangle \quad\quad \langle\Sigma_3\rangle\ P_3 \Longrightarrow \langle\Sigma_4\rangle \end{array}}{\langle\Sigma_1\rangle\ J \Longrightarrow \langle\Sigma_4\rangle}$$

The workflow of a poor soul

- … scaring

$$
(\overset{\mathcal{R}\Uparrow}{\rightsquigarrow}-\text{match})\quad
\dfrac{\begin{array}{ll}
(I_l : \Pi\overrightarrow{x_l : F_l}.\Pi\overrightarrow{y_r : G_r}.s) \in \mathsf{Env} & \\
(k_j : \Pi\overrightarrow{x_l : F_l}.\Pi\overrightarrow{y^j_{n_j} : T^j_{n_j}}.I_l\ \overrightarrow{x_l}\ \overrightarrow{M^j_r}) \in \mathsf{Env} & j \in \{1\dots n\} \\
 & \\
\Sigma \rightsquigarrow \Sigma \cup \{\Gamma \vdash ?u_i :\ F_i[\overrightarrow{x_{i-1}/?u_{i-1}}]\} & 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}}]\} & 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}] & i \in \{1\dots r\} \\
T'^j_i = T^j_i[\overrightarrow{x_l/?u_l}] & j \in \{1\dots n\}, i \in \{1\dots n_j\} \\
M'^j_i = M^j_i[\overrightarrow{x_l/?u_l}] & 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 : G'_r}.\Pi x : I_l\ \overrightarrow{?u_l}\ \overrightarrow{y_r}.?_1[] \overset{\mathcal{R}\Downarrow}{\rightsquigarrow} T' & \\
(s, \Phi(?_1)) \in \mathsf{elim}(\mathsf{PTS}) & \\
 & \\
\Gamma; \overrightarrow{y^j_{n_j-1} : P^j_{n_j-1}} \vdash P^j_{n_j} \overset{?}{\equiv} T'^j_{n_j} \overset{\mathcal{U}}{\rightsquigarrow} & j \in \{1\dots n\} \\
\Gamma; \overrightarrow{y^j_{n_j} : P^j_{n_j}} \vdash t_j :\ T'\ \overrightarrow{M'^j_r}\ (k_j\ \overrightarrow{?u_l}\ \overrightarrow{y^j_{n_j}}) \overset{\mathcal{R}\Downarrow}{\rightsquigarrow} t'_j & j \in \{1\dots n\}
\end{array}}{\begin{array}{l}
\Gamma \vdash \left(\begin{array}{l}\mathbf{match}\ t\ \mathbf{in}\ I_l\ \mathbf{return}\ T \\ \quad [k_1\ (\overrightarrow{y^1_{n_1} : P^1_{n_1}}) \Rightarrow t_1 \mid \dots \mid k_n\ (\overrightarrow{y^n_{n_n} : P^n_{n_n}}) \Rightarrow t_n]\end{array}\right) \overset{\mathcal{R}\Uparrow}{\rightsquigarrow} \\
\quad \left(\begin{array}{l}\mathbf{match}\ t'\ \mathbf{in}\ I_l\ \mathbf{return}\ T' \\ \quad [k_1\ (\overrightarrow{y^1_{n_1} : P'^1_{n_1}}) \Rightarrow t'_1 \mid \dots \mid k_n\ (\overrightarrow{y^n_{n_n} : P'^n_{n_n}}) \Rightarrow t'_n]\end{array}\right) : T'\ \overrightarrow{?v_r}\ t'
\end{array}}
$$

Inference rule(s)!

- One key ingredient in the Odd Order Theorem is that we could program the elaborator by adding rules like:

$$\frac{\texttt{nat} \sim \texttt{EQ.obj nat\_EQty} \qquad ?x \sim \texttt{nat\_EQty}}{\texttt{nat} \sim \texttt{EQ.obj } ?x}$$

$$\frac{t_1 \sim \texttt{EQ.obj } ?y \qquad t_2 \sim \texttt{EQ.obj } ?z \qquad ?x \sim \texttt{pair\_EQty } ?y\ ?z}{t_1 * t_2 \sim \texttt{EQ.obj } ?x}$$

Can we turn \frac into a programming language?

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

Elpi = λProlog + CHR

- λProlog for …
  - Context, substitution, assignment
  - ✔ programming with binders, recursively
- CHR for …
  - State and metadata management
  - ✔ manipulate unification variables
  - ✔ non-local deductions

Outline

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

λProlog 101

$$e = x \mid e_1 e_2 \mid \lambda x.e$$

```
% HOAS of terms

type app term → term → term.

type lam (term → term) → term.
```

$$\tau = C \mid \tau \to \tau$$

```
% 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

Q_0 = arrow S_1 T_1

F_1 = (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 c_1 c_2) T_2.

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 S_1.

of c_2 S_2.

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)

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

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

λProlog 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 = 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

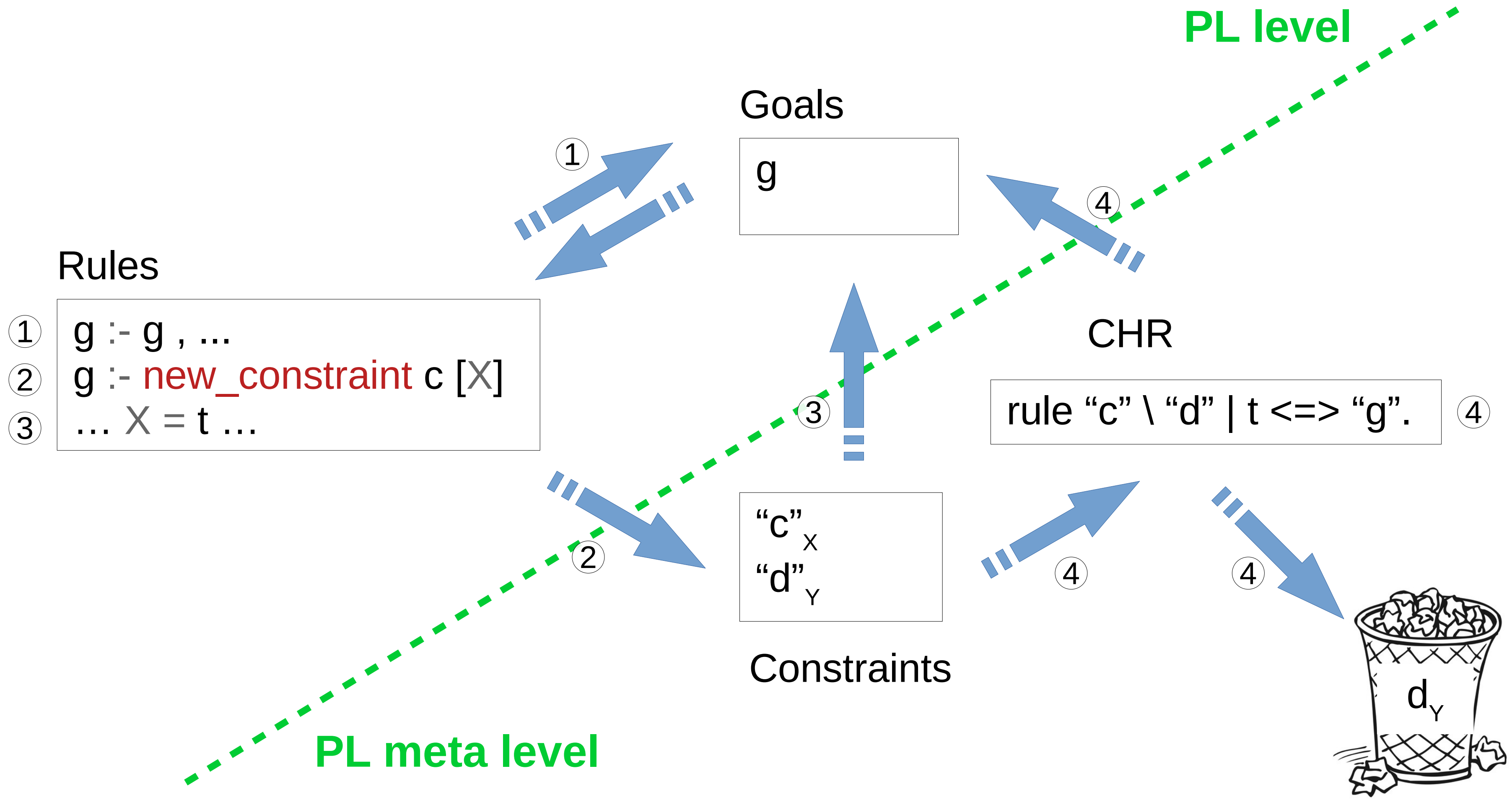
Program

```
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 $\mathbf{F}_X$

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 F_Y
double F_Z F_Y
even 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 …
  - Context, substitution, assignment
  - ✔ programming with binders, recursively
- CHR for …
  - State and metadata management
  - ✔ manipulate unification variables
  - ✔ non-local deductions

Demo: HM in Elpi

Hindley–Milner type system

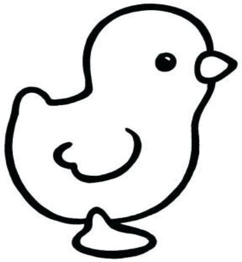
3 languages

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]

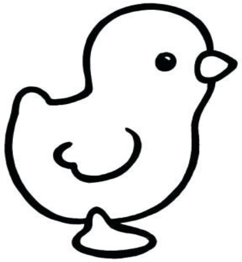
Among HM's more notable properties are its completeness and its ability to infer the most general type of a given program without programmer-supplied type annotations or other hints. Algorithm W is an efficient type inference method in practice and has been successfully applied on large code bases, although 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

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

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



Typing rules

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

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

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

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

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

Type schema: elimination

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

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

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

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

Type schema: introduction

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

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

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

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

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

$$\ldots$$

Elpi → 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 @@ Himsg.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

Thanks for listening!

Elpi ! logic programming

- high level with an operational meaning
  - yummy!
- Fact of life: 90% compute, 10% search
  - wrong default (who needs backtracking?)
- extensibility of programs (rule based)
  - a miracle