The Makam Metalanguage

Reducing the cost of PL experimentation

Antonis Stampoulis   Adam Chlipala

MIT Computer Science and Artifical Intelligence Laboratory

Softlab PL Seminar 2013

We have various crazy PL ideas

We have various crazy PL ideas

- Dependent types: capture program invariants in types
- VeriML: programs to prove the invariants
- Ur/Web: avoid SQL injections etc. in webapps statically

We have various crazy PL ideas

- Dependent types: capture program invariants in types
- VeriML: programs to prove the invariants
- Ur/Web: avoid SQL injections etc. in webapps statically

… but running experiments takes huge up-front cost

Experimentation requires implementation

Experimentation requires implementation

- Design space is large; many choices arbitrary at initial phases
- Implement a language from scratch vs. extend an existing language
- Practical aspects are important but tricky
  - efficiency? error-messages?
- Extensibility/malleability of implementation is key but runs counter to doing full-fledged implementation

Experimentation requires implementation

- Design space is large; many choices arbitrary at initial phases
- Implement a language from scratch vs. extend an existing language
- Practical aspects are important but tricky
  - efficiency? error-messages?
- Extensibility/malleability of implementation is key but runs counter to doing full-fledged implementation

$\rightarrow$ Many PL ideas stay at prototype level

Makam
a metalanguage for quick PL prototyping

Makam

a metalanguage for quick PL prototyping

- **declarative** and **executable** rules for specifying languages
- logic programming (Prolog) + PL-related magic
- can model type systems, transformations to existing languages, etc.

Makam

a metalanguage for quick PL prototyping

- **declarative** and **executable** rules for specifying languages
- logic programming (Prolog) + PL-related magic
- can model type systems, transformations to existing languages, etc.

- reduce time for prototype from months to days
- fast changes to key design decisions
- specifications are easy to extend
- metalanguage takes care of tricky parts

Let's use Makam to model the simply-typed lambda calculus.

Abstract syntax.

$$\tau ::= Int \mid Bool \mid \tau_1 \to \tau_2$$

Abstract syntax.

$$\tau ::= Int \mid Bool \mid \tau_1 \to \tau_2$$

```
typ : sort.
```

Abstract syntax.

$$\tau ::= Int \mid Bool \mid \tau_1 \to \tau_2$$

```
typ : sort.
tint : typ. tbool : typ.
tarrow : typ -> typ -> typ.
```

Abstract syntax.

$$\tau ::= Int \mid Bool \mid \tau_1 \to \tau_2$$

```
typ : sort.
tint : typ. tbool : typ.
tarrow : typ -> typ -> typ.
```

$$\begin{aligned} e ::= {} & e_1 + e_2 \mid e_1 < e_2 \mid n \mid \textsf{if}\ e\ \textsf{then}\ e_1\ \textsf{else}\ e_2 \\ & \mid e_1\ e_2 \mid \lambda x.e \end{aligned}$$

Abstract syntax.

$$\tau ::= Int \mid Bool \mid \tau_1 \to \tau_2$$

```
typ : sort.
tint : typ. tbool : typ.
tarrow : typ -> typ -> typ.
```

$$\begin{aligned} e ::= {} & e_1 + e_2 \mid e_1 < e_2 \mid n \mid \textsf{if } e \textsf{ then } e_1 \textsf{ else } e_2 \\ & \mid e_1\ e_2 \mid \lambda x.e \end{aligned}$$

```
expr : sort.
plus : expr -> expr -> expr.
lt : expr -> expr -> expr.
intconst : int -> expr.
```

Typing and evaluation relations.

Typing

$$\Gamma \vdash e : \tau$$

```
typeof : expr -> typ -> prop.
```

Big-step semantics

$$e \Downarrow e'$$

```
eval : expr -> expr -> prop.
```

Sample rule.

Let’s do an easy rule first.

Sample rule.

Let's do an easy rule first.

$$\boxed{\dfrac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 < e_2 : Bool}}$$

Sample rule.

Let's do an easy rule first.

$$\boxed{\frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 < e_2 : Bool}}$$

```
typeof (lt E1 E2) tbool <-
  typeof E1 tint,
  typeof E2 tint.
```

Sample rule.

Let's do an easy rule first.

$$\boxed{\frac{\Gamma \vdash e_1 : Int \qquad \Gamma \vdash e_2 : Int}{\Gamma \vdash e_1 < e_2 : Bool}}$$

```
typeof (lt E1 E2) tbool <-
  typeof E1 tint,
  typeof E2 tint.
```

Easy: Just as in Prolog.

If-then-else.

$$\boxed{\dfrac{e \Downarrow \mathit{True} \qquad e_1 \Downarrow v}{\textsf{if}\ e\ \textsf{then}\ e_1\ \textsf{else}\ e_2 \Downarrow v}}$$

If-then-else.

$$\boxed{\dfrac{e \Downarrow \mathit{True} \qquad e_1 \Downarrow v}{\textsf{if } e \textsf{ then } e_1 \textsf{ else } e_2 \Downarrow v}}$$

```
eval (ifthenelse E E1 E2) V <-
  eval E btrue, eval E1 V.
```

Application is easy too.

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

```
app : expr -> expr -> expr.
tarrow : typ -> typ -> typ.

typeof (app E1 E2) T' <-
  typeof E1 (tarrow T T'),
  typeof E2 T.
```

Lambda-case.

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

Lambda-case.

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

```
var : string -> expr. lam : string -> expr -> expr.
typeof (lam X E) (tarrow T T’) <-
  (typeof (var X) T ->
  typeof E T’).
```

Lambda-case.

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

```
var : string -> expr. lam : string -> expr -> expr.
typeof (lam X E) (tarrow T T') <-
  (typeof (var X) T ->
  typeof E T').
```

Now let's do the evaluation rules.

$$\boxed{\dfrac{e_1 \Downarrow \lambda x.e' \qquad e_2 \Downarrow v \qquad e'[v/x] \Downarrow v'}{e_1\ e_2 \Downarrow v'}}$$

Now let's do the evaluation rules.

$$\boxed{\dfrac{e_1 \Downarrow \lambda x.e' \qquad e_2 \Downarrow v \qquad e'[v/x] \Downarrow v'}{e_1\ e_2 \Downarrow v'}}$$

$$\begin{array}{l}
x[e/x] = e \\
y[e/x] = y \\
(\lambda x.e)[e'/x] = \lambda x.e \\
(\lambda y.e)[e'/x] = \lambda y.(e[e'/x])\ \ if\ \ y \notin fv(e') \\
(e_1\ e_2)[e/x] = e_1[e/x]\ e_2[e/x] \\
\ldots
\end{array}$$

Now let's do the evaluation rules.

$$\boxed{\dfrac{e_1 \Downarrow \lambda x.e' \qquad e_2 \Downarrow v \qquad e'[v/x] \Downarrow v'}{e_1\ e_2 \Downarrow v'}}$$

$$\begin{aligned}
&x[e/x] = e\\
&y[e/x] = y\\
&(\lambda x.e)[e'/x] = \lambda x.e\\
&(\lambda y.e)[e'/x] = \lambda y.(e[e'/x])\ \mathit{if}\ \ y \notin \mathit{fv}(e')\\
&(e_1\ e_2)[e/x] = e_1[e/x]\ e_2[e/x]\\
&\ldots
\end{aligned}$$

This seems like a lot of work. Let's backtrack...

Lambda-case.

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

Lambda-case.

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

```
lam : ?
```

Lambda-case.

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

```
lam : ?
```

Idea: use meta-level function type to represent binding. Meta-level application is object-level substitution.
(Higher-order abstract syntax.)

Lambda-case.

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

```
lam : (expr -> expr) -> expr.
```

Lambda-case.

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

```
lam : (expr -> expr) -> expr.
```

```
typeof (lam EF) (tarrow T T’) <-
  (x:expr ->
  typeof x T -> typeof (EF x) T’).
```

Lambda-case.

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

```
lam : (expr -> expr) -> expr.
```

```
typeof (lam EF) (tarrow T T’) <-
  (x:expr ->
  typeof x T -> typeof (EF x) T’).

eval (app E1 E2) V’ <-
  eval E1 (lam EF), eval E2 V,
  eval (EF V) V’.
```

Querying.

$$\boxed{\Gamma \vdash e : ?}$$

```
typeof (lam (fun x => lam (fun y =>
         ifthenelse x y (plus y y))))
       T ?
```

Querying.

$$\boxed{\Gamma \vdash e : ?}$$

```
typeof (lam (fun x => lam (fun y =>
       ifthenelse x y (plus y y))))
      T ?

» T := tarrow tbool (tarrow tint tint)
```

What about more complicated typing features?

Polymorphism.

$$\boxed{\dfrac{\Delta,\ \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \Pi\alpha.\tau}}$$

Polymorphism.

$$\boxed{\frac{\Delta,\ \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \Pi\alpha.\tau}}$$

```
pi : (typ -> typ) -> typ.
lamt : (typ -> expr) -> expr.

typeof (lamt EF) (pi TF) <-
  (a:typ -> typeof (EF a) (TF a)).
```

Polymorphism.

$$\boxed{\dfrac{\Delta; \Gamma \vdash e : \Pi\alpha.\tau}{\Delta; \Gamma \vdash e\, \tau' : \tau[\tau'/\alpha]}}$$

```
appt : expr -> typ -> expr.
typeof (appt E T) (TF T) <-
  typeof E (pi TF).
```

HM-style generalization.

```
typeof (lam (fun x => x)) ?
```

HM-style generalization.

```
typeof (lam (fun x => x)) ?
» T := tarrow T1 T1
```

HM-style generalization.

```
typeof (lam (fun x => x)) ?
» T := tarrow T1 T1

gen (tarrow T1 T1) T ?
```

HM-style generalization.

```
typeof (lam (fun x => x)) ?
» T := tarrow T1 T1

gen (tarrow T1 T1) T ?
» T := tpi (fun a => tarrow a a)
```

HM-style generalization.

```
typeof (lam (fun x => x)) ?
» T := tarrow T1 T1

gen (tarrow T1 T1) T ?
» T := tpi (fun a => tarrow a a)

gen : typ -> typ -> prop.
gen T T <- not(getunif T (X : typ) _).
gen T (pi TF) <-
  getunif T (X : typ) T',
  (a:typ -> gen (T' a) (TF a)).
```

HM-style generalization.

```
typeof (lam (fun x => x)) ?
» T := tarrow T1 T1

gen (tarrow T1 T1) T ?
» T := tpi (fun a => tarrow a a)

gen : typ -> typ -> prop.
gen T T <- not(getunif T (X : typ) _).
gen T (pi TF) <-
  getunif T (X : typ) T',
  (a:typ -> gen (T' a) (TF a)).
```

Given T, get the first unification variable of type typ and abstract over it.

Mutually recursive definitions.

$$\boxed{\dfrac{\Gamma, \overrightarrow{xs} : \overrightarrow{\tau} \vdash es_i : \tau_i \qquad \Gamma, \overrightarrow{xs} : \overrightarrow{\tau} \vdash e : \tau'}{\Gamma \vdash \textsf{letrec}\ \overrightarrow{xs} = \overrightarrow{es}\ \textsf{in}\ e : \tau'}}$$

Mutually recursive definitions.

$$\boxed{\frac{\Gamma, \overrightarrow{xs} : \overrightarrow{\tau} \vdash es_i : \tau_i \qquad \Gamma, \overrightarrow{xs} : \overrightarrow{\tau} \vdash e : \tau'}{\Gamma \vdash \textsf{letrec}\ \overrightarrow{xs} = \overrightarrow{es}\ \textsf{in}\ e : \tau'}}$$

```
letrec : bindmany expr (list expr * expr)
         -> expr.

typeof (letrec F) T’ <-
  open F as (Defs, Body) binding xs in
  assumemany typeof xs TS in
  (map typeof Defs TS,
  typeof Body T’).
```

Scalable in terms of expressivity.

- Big part of the OCaml type system in ~500 lines of code.
- Mutually recursive definitions of types and expressions.
- Algebraic datatypes.
- Pattern matching.
- Modules and module signatures.
- HM-style generalization.
- Type synonyms with expansion.
- Extensions are possible: e.g. type classes.
- No code modified, new code: ~100 lines of code.

Existential types.

```
typeof (pack T’ E) (sigma TF) <-
  typeof E (TF T’).

typeof (unpack E EF) T’ <-
  typeof E (sigma TF),
  (a:typ -> x:term ->
    typeof x (TF a) -> typeof (EF a x) T’).
```

Existential types.

```
typeof (pack T’ E) (sigma TF) <-
  typeof E (TF T’).

typeof (unpack E EF) T’ <-
  typeof E (sigma TF),
  (a:typ -> x:term ->
    typeof x (TF a) -> typeof (EF a x) T’).
```

These two rules are comparably effective to serious type inferencing for existential types. How is this possible?

Behind the hood.

Behind the hood.

Makam is essentially a new implementation and refinement of λProlog.

Behind the hood.

Makam is essentially a new implementation and refinement of λProlog.

- Prolog uses *s-expressions* as atomic terms.

Behind the hood.

Makam is essentially a new implementation and refinement of λProlog.

- Prolog uses *s-expressions* as atomic terms.
- Typed abstract syntax $\rightarrow$ *typed s-expressions*.

Behind the hood.

Makam is essentially a new implementation and refinement of λProlog.

- Prolog uses *s-expressions* as atomic terms.
- Typed abstract syntax $\rightarrow$ *typed s-expressions*.
- Meta-level functions for HOAS $\rightarrow$ *terms of the simply-typed lambda calculus*.

Behind the hood.

Makam is essentially a new implementation and refinement of λProlog.

- Prolog uses *s-expressions* as atomic terms.
- Typed abstract syntax $\rightarrow$ *typed s-expressions*.
- Meta-level functions for HOAS $\rightarrow$ *terms of the simply-typed lambda calculus*.
- Polymorphic types (e.g. lists) $\rightarrow$ *terms of the polymorphic lambda calculus*.

Behind the hood.

Makam is essentially a new implementation and refinement of λProlog.

- Prolog uses *s-expressions* as atomic terms.
- Typed abstract syntax $\rightarrow$ *typed s-expressions*.
- Meta-level functions for HOAS $\rightarrow$ *terms of the simply-typed lambda calculus*.
- Polymorphic types (e.g. lists) $\rightarrow$ *terms of the polymorphic lambda calculus*.
- These are the atomic terms of λProlog.

Behind the hood: λProlog.

Behind the hood: λProlog.

- Main operation in Prolog: *first-order unification*.

Behind the hood: λProlog.

- Main operation in Prolog: *first-order unification*.
- Switch to *polymorphic lambda calculus* $\rightarrow$ *typed higher-order unification*.

Behind the hood: λProlog.

- Main operation in Prolog: *first-order unification*.
- Switch to *polymorphic lambda calculus* $\to$ *typed higher-order unification*.
- Unify up to $\beta\eta$-equivalence.

Behind the hood: λProlog.

- Main operation in Prolog: *first-order unification*.
- Switch to *polymorphic lambda calculus* $\rightarrow$ *typed higher-order unification*.
- Unify up to $\beta\eta$-equivalence.
- Undecidable in the general case; restrict to decidable subset.

Behind the hood: λProlog.

- Main operation in Prolog: *first-order unification*.
- Switch to *polymorphic lambda calculus* $\rightarrow$ *typed higher-order unification*.
- Unify up to $\beta\eta$-equivalence.
- Undecidable in the general case; restrict to decidable subset.
- Even the restriction subsumes most common type inferencing problems.

Behind the hood: λProlog.

- Main operation in Prolog: *first-order unification*.
- Switch to *polymorphic lambda calculus* $\rightarrow$ *typed higher-order unification*.
- Unify up to $\beta\eta$-equivalence.
- Undecidable in the general case; restrict to decidable subset.
- Even the restriction subsumes most common type inferencing problems.
- Also useful elsewhere: e.g. semantics of pattern matching.

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- No distinction between propositions determined statically and dynamically.

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- No distinction between propositions determined statically and dynamically.
- Made possible by the use of interpretation vs. compilation.

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- No distinction between propositions determined statically and dynamically.
- Made possible by the use of interpretation vs. compilation.
- A technicality with far-reaching consequences.

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- No distinction between propositions determined statically and dynamically.
- Made possible by the use of interpretation vs. compilation.
- A technicality with far-reaching consequences.
- … e.g. generic binding structures

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- No distinction between propositions determined statically and dynamically.
- Made possible by the use of interpretation vs. compilation.
- A technicality with far-reaching consequences.
- … e.g. generic binding structures

```
bnil : B -> bindmany A B.
bcons : (A -> bindmany A B) -> bindmany A B.
```

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- … but more importantly:
  Makam does staging for free.

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- … but more importantly:
  Makam does staging for free.
- Makam programs can compute Makam programs.

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- ... but more importantly:
  Makam does staging for free.
- Makam programs can compute Makam programs.
- Useful for all sorts of things:

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- ... but more importantly:
  Makam does staging for free.
- Makam programs can compute Makam programs.
- Useful for all sorts of things:
- implementing DSLs (e.g. LM: invertible ML-like language)

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- ... but more importantly:
  Makam does staging for free.
- Makam programs can compute Makam programs.
- Useful for all sorts of things:
- implementing DSLs (e.g. LM: invertible ML-like language)
- doing parsing (PEG combinators + LM semantic actions)

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- ... but more importantly:
  Makam does staging for free.
- Makam programs can compute Makam programs.
- Useful for all sorts of things:
- implementing DSLs (e.g. LM: invertible ML-like language)
- doing parsing (PEG combinators + LM semantic actions) with pretty-printing for free

Behind the hood: Makam.

Main difference with λProlog:
weak hereditary Harrop formulas.

- ... but more importantly:
  Makam does staging for free.
- Makam programs can compute Makam programs.
- Useful for all sorts of things:
- implementing DSLs (e.g. LM: invertible ML-like language)
- doing parsing (PEG combinators + LM semantic actions) with pretty-printing for free
- (eventually) implementing intermediate languages for Makam and bootstrapping

Parsing.

Invertible functional-style code.

```
forward : lm A B -> (A -> B -> prop) -> prop.
backward : lm A B -> (B -> A -> prop) -> prop.
```

PEG combinators compiled to invertible code.

```
pegcompile : peg A -> lm string A -> prop.
pegparse = forward ○ pegcompile.
pegprint = backward ○ pegcompile.
```

Parsing.

```
pterm ->
    ”λ” id:ident ”.” body:pterm
    { return lam (bind id body) }
  / f:pbaseterm args:rep(pbaseterm)
    { foldl (return app) f args }

pbaseterm ->
    id:ident { lookup id }
  / e:parenthesized(pterm) { e }
```

Guaranteed to produce well-typed, well-bound abstract syntax.

Conclusion.

Summary.

- Makam: a tool to simplify prototype PL implementation.
- Declarative, Prolog-style rules for specifying different aspects of languages.
- Re-use tricky stuff as implemented in the meta-language.
- Higher-order features allow powerful abstractions.
- Surprisingly expressive formalism – still figuring things out!

Current & future work.

- Base language features are fairly stable.
- Doing a profiling and optimization phase.
- Plan to experiment with further type systems using Makam (VeriML, Ur/Web, etc.)

Backup slides.

Existential types.

$$\boxed{\frac{\Delta; \Gamma \vdash e : \tau[\tau'/\alpha]}{\Delta; \Gamma \vdash \langle\, \tau',\ e \,\rangle : \Sigma\alpha.\tau}}$$

```
sigma : (typ -> typ) -> typ.
pack : typ -> expr -> expr.

typeof (pack T’ E) (sigma TF) <-
  typeof E (TF T’).
```

Existential types.

$$\frac{\Delta ; \Gamma \vdash e : \Sigma\alpha.\tau \qquad \Delta,\ \alpha' ; \Gamma,\ x : \tau[\alpha'/\alpha] \vdash e' : \tau' \qquad \alpha' \notin fv(\tau')}{\Delta ; \Gamma \vdash \textbf{let}\ \langle\ \alpha,\ x\ \rangle = e\ \textsf{in}\ e' : \tau'}$$

```
unpack : expr ->
         (typ -> expr -> expr) ->
         expr.

typeof (unpack E EF) T' <-
  typeof E (sigma TF),
  (a:typ -> x:term ->
    typeof x (TF a) ->
    typeof (EF a x) T').
```

Behind the hood: Makam.

Differences with λProlog.

Behind the hood: Makam.

Differences with λProlog.

- Some small practical features: e.g. proper naming through hybrid abstract/concrete variables.

Behind the hood: Makam.

Differences with λProlog.

- Some small practical features: e.g. proper naming through hybrid abstract/concrete variables.
- Implemented in OCaml instead of C.

Behind the hood: Makam.

Differences with λProlog.

- Some small practical features: e.g. proper naming through hybrid abstract/concrete variables.
- Implemented in OCaml instead of C.
- Interpreted* instead of compiled.

Behind the hood: Makam.

Differences with λProlog.

- Some small practical features: e.g. proper naming through hybrid abstract/concrete variables.
- Implemented in OCaml instead of C.
- Interpreted* instead of compiled.
- *In a naive way instead of after N years of research.

Behind the hood: Makam.

Differences with λProlog.

- Some small practical features: e.g. proper naming through hybrid abstract/concrete variables.
- Implemented in OCaml instead of C.
- Interpreted* instead of compiled.
- *In a naive way instead of after N years of research.
- Makam is many times slower.

Behind the hood: Makam.

Differences with λProlog.

- Some small practical features: e.g. proper naming through hybrid abstract/concrete variables.
- Implemented in OCaml instead of C.
- Interpreted* instead of compiled.
- *In a naive way instead of after N years of research.
- Makam is many times slower.
- Still, we plan to use it as a practical tool.

Behind the hood: Makam.

Differences with λProlog.

- Some small practical features: e.g. proper naming through hybrid abstract/concrete variables.
- Implemented in OCaml instead of C.
- Interpreted* instead of compiled.
- *In a naive way instead of after N years of research.
- Makam is many times slower.
- Still, we plan to use it as a practical tool.
- ?!?

?!?

Let's revisit the case with multiple bindings.

```
typeof (letrec F) T' <-
  open F as (Defs, Body) binding xs in
  assumemany typeof xs TS in
  (map typeof Defs TS,
  typeof Body T').
```

?!?

Let's revisit the case with multiple bindings.

```
typeof (letrec F) T' <-
  open F as (Defs, Body) binding xs in
  assumemany typeof xs TS in
  (map typeof Defs TS,
  typeof Body T').
```

?!?

Let's revisit the case with multiple bindings.

```
typeof (letrec F) T' <-
  open F as (Defs, Body) binding xs in
  assumemany typeof xs TS in
  (map typeof Defs TS,
  typeof Body T').

assumemany : (A -> B -> prop) -> list A -> list B
      -> prop -> prop.
assumemany P [] [] Q <- Q.
assumemany P (X :: XS) (T :: TS) Q <-
  (P X T -> assumemany P XS TS Q).
```

?!?

Let's revisit the case with multiple bindings.

```
typeof (letrec F) T' <-
  open F as (Defs, Body) binding xs in
  assumemany typeof xs TS in
  (map typeof Defs TS,
  typeof Body T').

assumemany : (A -> B -> prop) -> list A -> list B
      -> prop -> prop.
assumemany P [] [] Q <- Q.
assumemany P (X :: XS) (T :: TS) Q <-
  (P X T -> assumemany P XS TS Q).
```

Staging in Makam.

Connectives, clauses, etc. are normal terms.

```
and : prop -> prop -> prop.
or : prop -> prop -> prop.
newvar : (A -> prop) -> prop.
newmeta : (A -> prop) -> prop.
assume : prop -> prop -> prop.
...
```

Predicates can compute propositions; we can then use the result normally.

```
invert : prop -> prop -> prop.
invert (and P Q) (and Q’ P’) <-
  invert P P’, invert Q Q’.
```