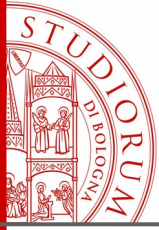


# Towards Extensible Algorithmic Mathematical Knowledge

*Claudio Sacerdoti Coen*  
<[claudio.sacerdoticoen@unibo.it](mailto:claudio.sacerdoticoen@unibo.it)>

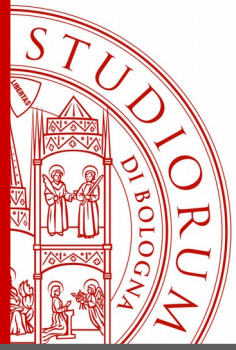
*Bialystok (PL), 26/07/16*



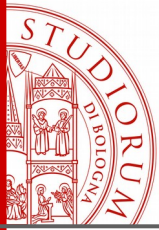
# Plan of the Talk

---

- Status of algorithmic knowledge in mathematical libraries and interactive theorem provers
- Algorithmic knowledge in user space: a language proposal
- Work in progress and achievements



# Status of algorithmic knowledge in mathematical libraries and interactive theorem provers



# Algorithmic Mathematical Knowledge

---

“Algorithmic” knowledge is everywhere!

- In the large (quantifier elimination, Grobner bases, Gaussian elimination, division alg.)
- In the small (when/how to apply a lemma, what to recur on, how to disambiguate symbols, ...)

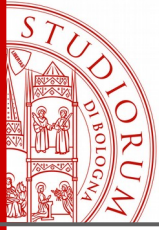


# Algorithmic Mathematical Knowledge

---

“Algorithmic” knowledge is impenetrable in ITPs!

- In the large (tactics, decision procedures)
- In the small (inner mechanisms + user extensions in ad-hoc languages)



# Algorithmic Mathematical Knowledge

---

“Algorithmic” knowledge is hidden or fuzzy in rigorous mathematics!

- In the large (pseudocode/actual code on ad-hoc data structures)
- In the small (shamefully omitted from papers/books)

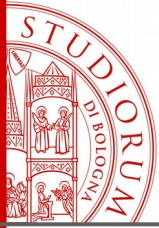


# Algorithmic Mathematical Knowledge

---

“Algorithmic” knowledge is forgotten in MKM libraries!

- In the large (code and data can be encoded, but in ad-hoc way and lacking operational semantics)
- In the small



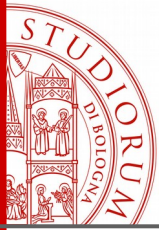
# AMK and AITP

---

Can AITP (Artificial Intelligence + Theorem Proving) recover AMK?

- In the large: no
- In the small: partially
  - how to use a lemma: OK
  - how to interpret a statement: :-)



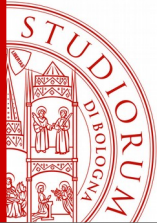


# CICM = MKM + Calculemus

---

How did we forget Calculemus in MKM libraries?

- Language choice
- Performance issues
- Lack of content
- What small AMK to put in? (limit case: parsing, type-checking, proof checking, ...)



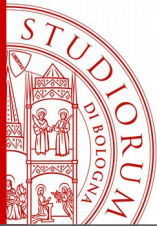
# Major Issues

---

- Performance issues

In a library:

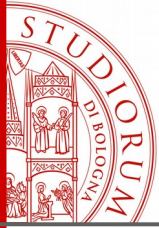
- Reference implementation only  
(What is the algorithm?  
How is knowledge used?)
- Performance is not an issue  
(the code can be reimplemented as long as the  
reference implementation/spec is given)



# Major Issues

---

- Lack of content
  - impossible to get from rigorous math?  
(and what applications?)
    - Plenty of sources from ITPs and CAS  
(with immediate applications to ITPs)  
but not immediately usable
      - Low level encoding/language
      - Focus on performance

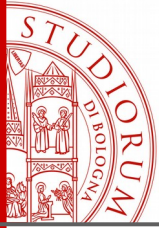


# Major Issues

---

- Language choice

Second part of the talk!

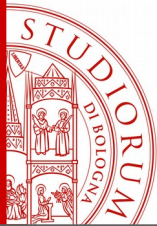


# Major Issues

---

- What small AMK to put in?

Let's analyze a few sources of AMK in ITPs.



# AMK in the small: Coercions

- Coercion  $X : A \rightarrow Y : B$  (Y output)

HOW to promote an  $X : A$  to a  $Y : B$

$$x : A \rightarrow F x : B$$

---

$$X : \text{nat} \rightarrow \text{int\_of\_nat } X : \text{int}$$

---

$$L : \text{list } A \rightarrow \text{map } A \ B \ (\lambda x. Fx) : \text{list } B$$

$$A : \text{Type} \rightarrow G : \text{SemiGroup}$$

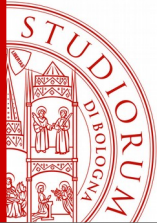
$$B : \text{Type} \rightarrow H : \text{SemiGroup}$$

---

$$\text{nat} : \text{Type} \rightarrow (\text{nat}, 0, 1, +, *) : \text{SemiGroup}$$

---

$$A \times B : \text{Type} \rightarrow G \times H : \text{SemiGroup}$$



# AMK in the small: Canonical Structures

---

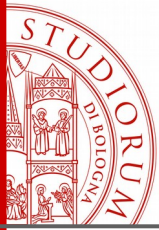
- Associate/extract functions/lemmas from types (e.g. algebraic structures)

– Without:

plus\_comm:  $\forall n, m: \text{nat}. n + m = m + n$

Z\_plus\_comm:  $\forall x, y: \text{int}. x + y = y + x$

union\_comm:  $\forall U: \text{Type}. \forall A, B: P(U). A \cup B = B \cup A$



# AMK in the small: Canonical Structures

- Associate/extract functions/lemmas from types (e.g. algebraic structures)

Class CommMagma =

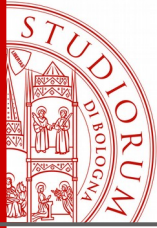
{ C : Type

;  $\star : C \rightarrow C \rightarrow C$

; comm :  $\forall x, y : C. x \star y = y \star x$

}

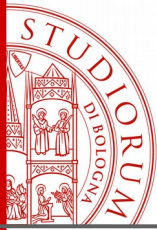




# AMK in the small: Canonical Structures

- Associate/extract functions/lemmas from types (e.g. algebraic structures)

```
Instance NatPlus : CommMagma =  
{ C = nat  
; * = +  
; comm = ... /* open proof obligation */  
}
```



# AMK in the small: Canonical Structures

to prove  $2 + n = n + 2$   
apply lemma comm

- Comm :

$\forall M : \text{CommMagma.}$

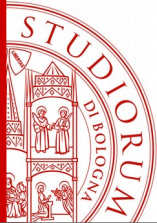
$\forall x, y : M. C. \ x \ M. \star \ y = y \ M. \star \ x$

- Unification problem :

$(2 + n = n + 2) \cong (x \ M. \star \ y = y \ M. \star \ x)$

i.e. find a CommMagma M s.t.

$M.C = \text{nat} \quad \wedge \quad M.\star = +$

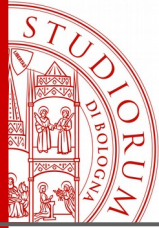


# AMK in the small: Canonical Structures

---

Gonthier's MathComp/Feith-Thompson proof heavily based on canonical structures

- Less library pollution
- More structure in libraries via inheritance
- No need to remember/find lemmas
- User controlled proof search via parameterized instances
- Robust (??)



# AMK in the small: Unification Hints

- Canonical Structures as special cases of Unification Hints

$$M = \{ \text{nat}, +, \dots \}$$

---

$$M.C \cong \text{nat}$$

$$M = \{ \text{nat}, +, \dots \}$$

---

$$M.\star \cong +$$

$$G.C \cong A$$

$$H.C \cong B$$

$$M = G \times H$$

---

$$M.C \cong A \times B$$

$$\langle\langle S1 \rangle\rangle \cong E$$

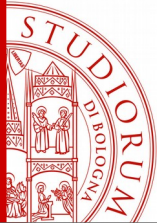
$$\langle\langle S2 \rangle\rangle \cong F$$

$$S = \text{Plus}(S1, S2)$$

---

$$\langle\langle S \rangle\rangle \cong E + F$$

quoting  
expressions,  
i.e. reflexion



# AMK in the small: User Space Tactics

---

- Tactics in ML/Haskell/Java/...
  - They deal with representation details (encoding of binders, metavariables, n-ary vs binary application, types in terms, ...)
  - Low level, error prone
  - Hard to maintain
  - Only for power users
  - Non portable (knowledge lost in libraries)



# AMK in the small: User Space Tactics

---

- User level language for tactics (LTac)

!

---

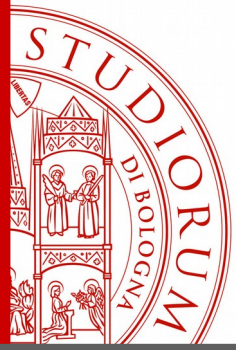
$\text{real\_tac}, (\Gamma, P \mid X, \Delta \vdash G) \Rightarrow \text{cases } (X \geq 0 \vee X < 0) ; \text{real\_tac}$

$(( S )) \cong E \quad (( T )) \cong F$

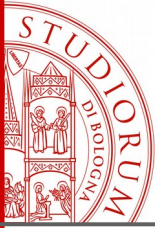
---

$\text{real\_tac}, (\Gamma \vdash E = F) \Rightarrow$   
 $\text{rew } (\text{normalize\_correct } S) ; \text{rew } (\text{normalize\_correct } T) ; \text{real\_tac}$

$\text{normalize\_correct} : \forall S: \text{syntax}. (( \eta S )) = (( S ))$



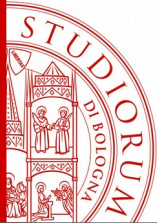
# Algorithmic knowledge in user space: a language proposal



# One Language to Bind Them All

- Binders, scope,  $\alpha$ -conversion, capture avoiding substitution
- Metavariables, scope, non capture avoiding instantiation  
E.g.:  $\forall x. \exists M, N. \forall y. (M \ y \cong x+y \wedge N \cong x + y)$   
 $M := \lambda y. x + y$   
 $N$  no solution
- Declarative + control (i.e. Prolog like)
- Minimalist

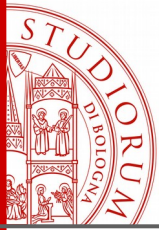




# Logical Framework?

---

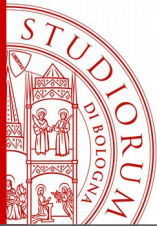
- Great for encoding logics and typing rules
  - Binders etc. for free via Higher Order Abstract Syntax / Lambda Tree Syntax
- Not enough for “general purpose” programming
  - Rabe’s generic type/proof checker extended via Java code
  - No “first class” metavariables
    - No partial terms and proofs



# Declarative Languages

---

- Most AMK is naturally in rule form
- Simple logical semantics (omitting control)
- Non deterministic
- Minimalist, smaller design space
  
- What about binders and metavariables?



# $\lambda$ Prolog

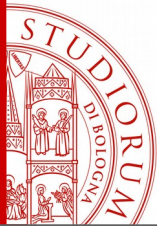
- Terms:  $\lambda$ -abstraction  $x \backslash t$  (highest prec)
  - all binders via HOAS  $\text{integral } 0 \ 10 \ x \backslash x * x$   
 $\text{lam } x \backslash \text{app } x \ x$
  - capture avoiding substitution via  $\beta$

$\text{reduces\_to } (\text{app } (\text{lam } F) T) (F T).$

$\text{reduces\_to } (\text{app } M1 N) (\text{app } M2 N) \text{ :- reduces\_to } M1 M2.$

?-  $\text{reduces\_to } (\text{app } (\text{lam } x \backslash \text{app } f x) y) O.$

$O \text{ := } f y$

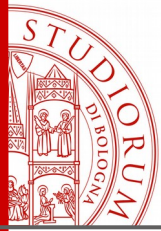


# $\lambda$ Prolog

- Largest possible fragment of intuitionistic logic that has complete Prolog-like proof search

$$Q ::= \text{sigma } X \setminus Q \mid Q, Q \mid Q; Q \mid x \text{ t } .. \text{ t}$$
$$\mid \text{pi } x \setminus Q \mid C \Rightarrow Q$$
$$C ::= \text{pi } x \setminus C \mid C, C \mid x \text{ t } .. \text{ t}$$
$$\mid Q \Rightarrow C$$

- $\text{pi } x \setminus Q$  : introduces a new eigenvar  $x$
- $C \Rightarrow Q$  : assumes  $C$  to prove  $Q$

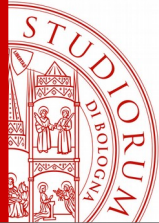


# Simply Typed $\lambda$ -calculus in $\lambda$ Prolog

$\text{typ (app M N) B :-}$   
 $\text{typ M (A --> B), typ N A.}$   $\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$

$\text{typ (lam M) (A --> B) :-}$   
 $\text{pi x \ typ x A => typ (M x) B.}$   $\frac{\Gamma, x:A \vdash M[x/y] : B}{\Gamma \vdash \lambda y. M : A \rightarrow B}$

$$\frac{x:A \in \Gamma}{\Gamma \vdash x.: A}$$



# Simply Typed $\lambda$ -calculus in $\lambda$ Prolog

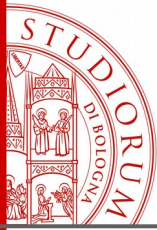
$\text{typ (app M N) B :-}$   
 $\text{typ M (A --> B), typ N A.}$

$\text{typ (lam M) (A --> B) :-}$   
 $\text{pi x \ typ x A => typ (M x) B.}$

$\text{typ g A} \quad \text{?- typ (lam f \ lam y \ app f y) T}$   
 $\text{typ g A, typ x C} \quad \text{?- typ (lam y \ app g y) B}$   
 $\text{typ g A, typ x C} \quad \text{?- typ (app g x) D}$   
 $\text{typ g A, typ x C} \quad \text{?- typ g (A' \to D), typ x A'}$

$T := A \text{ --> } B$   
 $B := C \text{ --> } D$   
 $A := A' \text{ --> } D$   
 $A' := C$

Answer:  $T := (C \text{ --> } D) \text{ --> } C \text{ --> } D$



# FO Prover in $\lambda$ Prolog

proves ( $\_$ , true).

proves (Gamma, or F G) :- proves Gamma F.

proves (Gamma, or F G) :- proves Gamma G.

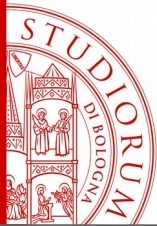
proves (Gamma, forall F) :-  $\pi x \setminus$  proves (Gamma, F x).

proves (Gamma, exists F) :-  $\sigma X \setminus$  proves (Gamma, F X).

proves (Gamma, Q) :-

pick Gamma (or F G) Delta, /\* Gamma = Delta + or F G \*/

proves [F|Delta] Q, proves [G|Delta] Q.



# Partial Objects

- Partial object = object containing metas

E.g.  $\text{lam } x \backslash \text{app } (M \ x) \ N$

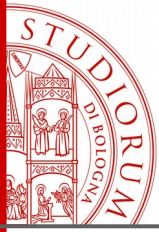
- e.g. omitted (recoverable) types
- e.g. placeholders in epsilon/delta proofs
- e.g. assignable vars/pointers

- Partial proof = proof object with metas

E.g.  $\text{lam } (s \rightarrow t) \ h \backslash \text{lam } s \ a \backslash X \ h \ a$

$$h: s \rightarrow t, a: s \vdash X \ h \ a : t$$





# , $\lambda$ Prolog Diverges on Partial Objects

$\text{typ (app M N) B :-}$   
 $\text{typ M (A --> B), typ N A.}$

$\text{typ (lam M) (A --> B) :-}$   
 $\text{pi x \ typ x A => typ (M x) B.}$

$\text{typ x nat ?- typ (app (M x) x) T.}$

$\text{typ x nat ?- typ (M x) (A \rightarrow T), typ x A.}$

BAD:  $M\ x := \text{app (M' x) (N' x)}$   
 $M' x := \text{app (M'' x) (N'' x)}$   
...



# Constrained Higher Order Logic

---

Proposal: extend  $\lambda$ Prolog with constraints via  $\$delay$ .

$\$delay$  (typ T TY) on flexible T.

typ (app M N) B :-  
 typ M (A --> B), typ N A.

typ (lam M) (A --> B) :-  
 pi x \ typ x A => typ (M x) B.

{ typ x nat ?- typ (M x) (A  $\rightarrow$  T) } is delayed



# Constrained Higher Order Logic

---

\$delay (typ T TY) on flexible T.

{ typ x nat ?- typ (M x) (A → T) } delayed

- delayed goals are fired when the guard becomes true

E.g. if  $M := \lambda x. x$   
then { typ x nat ?- typ x (A → T) } fired



# Constrained Higher Order Logic

- Constrains can be propagated according to user-provided rules (meta-theorems) in CHR style.

E.g. unicity of typing

$$\begin{aligned} & \{ \text{Gamma1} \text{?- typ T TY1} \} \Rightarrow \\ & \{ \text{Gamma2} \text{?- typ T TY2} \} \Leftrightarrow \\ & \text{restrict Gamma1 T} = \text{restrict Gamma2 T} \wedge \\ & \{ \text{?- TY1} = \text{TY2} \}. \end{aligned}$$

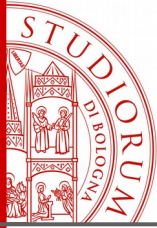
- $\{ \text{typ x nat, typ y bool} \text{?- typ (M x y) (nat} \rightarrow \text{A)},$   
 $\text{typ z bool, typ w nat} \text{?- typ (M w z) (B} \rightarrow \text{nat)} \} \Rightarrow$   
 $\{ \text{typ x nat, typ y bool} \text{?- typ (M x y) (nat} \rightarrow \text{A)},$   
 $\text{?- B} \rightarrow \text{nat} = \text{nat} \rightarrow \text{A} \}$



# Constrained Higher Order Logic

---

- NOTE:
  - Delayed goals are to be matched up to nominal unification
  - Computational expensive (NP)
  - Quest for efficient but expressive fragments (Work in progress)



# , λProlog Cannot Print Partial Objects

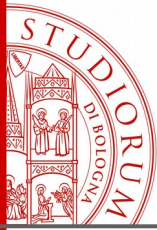
---

```
print (app M N) S12 :-  
  print M S1, print N S2, append [S1," ",S2] S12.
```

...

```
print x "x" ?- print (app (M x) x) S
```

```
M := λy. y /* print instantiates M! */
```

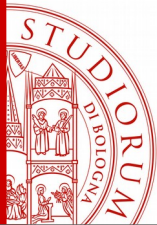


# , λProlog Cannot Match Partial Objects

```
proves (Gamma, Q) :-  
  split Gamma (or F G) Delta,  
  proves [F|Gamma] Q,  
  proves [G|Gamma] Q,  
  !. /* the rule is invertible,  
      never backtrack here */
```

?- proves ([H], not H).

Solution:  $H := \text{false}$ , but the ! kills it.



# Matching mode for $\lambda$ Prolog

`$mode(i,o)` for print.

`print (app M N) S12 :-`

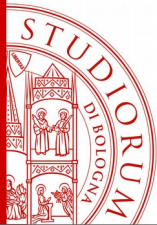
`print M S1, print N S2, append [S1," ",S2] S12.`

`print @(M,L) S :- ... /* do something */`

`...`

- The first argument is matched, not unified
- $@(M,L) \cong X \ t1 \ .. \ tn$  via  $M := X, L := [t1,\dots,tn]$

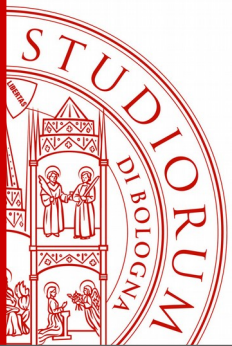




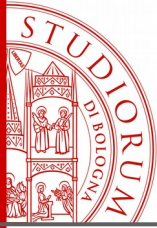
# Extensions to $\lambda$ Prolog

---

- Constraints
  - Hard to be made efficient
  - Preserve the logical semantics
- (Matching) modes, cut
  - Easy to implement
  - Destroy the logical/denotational semantics

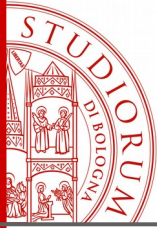


# Work in progress and achievements



# Achievements

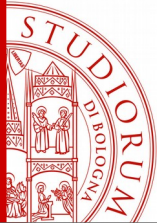
- ELPI (Embedded  $\lambda$ Prolog Interpreter)
  - with C. Dunchev, E. Tassi
  - written in OCaml
  - (almost) backward compatible with Teyjus compiler
  - faster than Teyjus :-)
  - 3100 lines equivalent to 3100 lines of Matita code (binders, reduction, metavariables, unification) (1500 for type-checking in Matita)



# Achievements

---

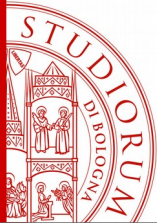
- Grundlagen in ELPI
  - F. Guidi
  - Benchmark for pure  $\lambda$ Prolog
  - type checker for Automath
  - 40 times slower than compiled OCaml code
    - 3 times slower than interpreted OCaml code



# Work in Progress

---

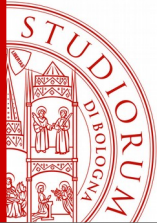
- Implementation of  $\lambda$ Prolog extensions in ELPI
  - With D. Miller, E. Tassi
  - still playing with the syntax/semantics



# Achievements

---

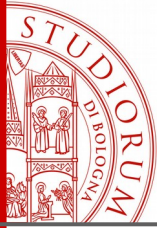
- Implementation of Intuitionistic HOL in ELPI
  - with C. Dunchev
  - kernel, basic tactics, inductive predicate package, basic library (Knaster-Tarski fixed point theorem, natural numbers)
  - Trusted code based:  
HOL-Light: 571  
Ours: 200 / 246
  - WiP on propagation rules (critical for speed)



## Achievements/Future Work

---

- Prototype of a Super LightWeight Matita in old version of ELPI (super slow)
- Core system: type/proof checking, type inference
- AMK in the library: overloading, implicit arguments, unification hints, coercions, tactics
- Under re-implementation for new fast ELPI

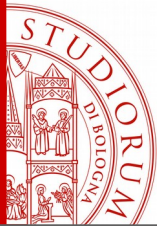


## More Future Work

---

- Fix syntax and semantics of ELPI
- Complete Matita and benchmark against it
- Compile ELPI (efficiently...)
- Reuse AMK from one system in another (Coq to Matita, HOL to Coq, etc.)





# Conclusion

---

- Algorithmic Mathematical Knowledge is anywhere but in our libraries
- Importing library from A to B without the algorithmic knowledge is unsatisfactory
- Constraint Higher Order Logic Programming as a candidate to
  - encode AMK
  - Implement proof assistant (prototypes)