Towards Extensible Algorithmic Mathematical Knowledge

Claudio Sacerdoti Coen
<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” 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” knowledge is impenetrable in ITPs!

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


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 \( Y : B \)

\[
\begin{align*}
X : \text{nat} & \rightarrow \text{int} \_\_\_ \text{nat} X : \text{int} \\
L : \text{list} A & \rightarrow \text{map} A B (\lambda x. Fx) : \text{list} B \\
A : \text{Type} & \rightarrow G : \text{SemiGroup} \\
B : \text{Type} & \rightarrow H : \text{SemiGroup} \\
nat : \text{Type} & \rightarrow (\text{nat},0,1,+,*) : \text{SemiGroup} \\
A \times B : \text{Type} & \rightarrow G \times H : \text{SemiGroup}
\end{align*}
\]
AMK in the small: Canonical Structures

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

  – Without:

    ```
    plus_comm: ∀n,m: nat. n + m = m + n
    Z_plus_comm: ∀x,y: int. x + y = y + x
    union_comm: ∀U: Type. ∀A,B: P(U). A ∪ B = B ∪ A
    ```
AMK in the small: Canonical Structures

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

Class CommMagma =
{ C : Type
; ⋆ : C → C → C
; comm : ∀ x,y : C. x ⋆ y = y ⋆ 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 */
}
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.
  $M.\ C. = \text{nat} \ \land \ M. \star = +$
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

\[
\begin{align*}
M &= \{ \text{nat, +, … } \} \\
M.C &\cong \text{nat} \\
M &= \{ \text{nat, +, … } \} \\
M.* &\cong + \\
G.C &\cong A \\
H.C &\cong B \\
M &= G \times H \\
M.C &\cong A \times B \\
\langle \text{S1 } \rangle &\cong E \\
\langle \text{S2 } \rangle &\cong F \\
S &= \text{Plus}(\text{S1},\text{S2}) \\
\langle S \rangle &\cong E + F \\
\text{quoting expressions, i.e. reflexion}
\end{align*}
\]
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)

\[
\text{real_tac, } (\Gamma, P \parallel X, \Delta \vdash G) \Rightarrow \text{cases } (X \geq 0 \lor X < 0) ; \text{real_tac}
\]

\[
\langle S \rangle \equiv E \quad \langle T \rangle \equiv F
\]

\[
\text{real_tac, } (\Gamma \vdash E = F) \Rightarrow \text{rew (normalize_correct S)} ; \text{rew (normalize_correct T)} ; \text{real_tac}
\]

\[
\text{normalize_correct : } \forall S: \text{syntax. } \langle \eta S \rangle = \langle S \rangle
\]
Algorithmic knowledge in user space: a language proposal
One Language to Bind Them All

- Binders, scope, α-conversion, capture avoiding substitution
- Metavariables, scope, non capture avoiding instantiation
  E.g.: $\forall x. \exists M, N. \forall y. (M \ y \equiv x + y \land N \equiv x + y)$
  $M := \lambda y. x + y$
  $N$ 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?
λProlog

- Terms: λ-abstraction \( x \setminus t \) (highest prec)
  - all binders via HOAS integral 0 10 x \( x \times x \)
    lam x \( \setminus \) app x x
  - capture avoiding substitution via \( \beta \)

reduces_to (app (lam F) T) (F T).
reduces_to (app M1 N) (app M2 N) :- reduces_to M1 M2.

?- reduces_to (app (lam x \( \setminus \) app f x) y) O.
O := f y
\(\lambda\text{Prolog}\)

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

\[
\begin{align*}
Q & ::= \sigma X \backslash Q \mid Q,Q \mid Q;Q \mid x \, t \ldots \, t \\
| & \quad \pi x \backslash Q \mid C \Rightarrow Q \\
C & ::= \pi x \backslash C \mid C,C \mid x \, t \ldots \, t \\
| & \quad Q \Rightarrow C
\end{align*}
\]

- \(\pi x \backslash Q\) : introduces a new eigenvar \(x\)
- \(C \Rightarrow Q\) : assumes \(C\) to prove \(Q\)
Simply Typed λ-calculus in λProlog

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

\[
\begin{array}{c}
\Gamma \vdash M : A \to B \\
\Gamma \vdash N : A
\end{array}
\Rightarrow
\Gamma \vdash MN : B
\]

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

\[
\begin{array}{c}
\Gamma, x : A \vdash M[x/y] : B
\end{array}
\Rightarrow
\Gamma \vdash \lambda y. M : A \to B
\]

\[
\begin{array}{c}
x : A \in \Gamma
\end{array}
\Rightarrow
\Gamma \vdash x : A
\]
Simply Typed λ-calculus in λProlog

\[\text{typ} \ (\text{app} \ M \ N) \ B \ :\ -\ \\
\quad \text{typ} \ M \ (A \rightarrow B), \ \text{typ} \ N \ A.\]

\[\text{typ} \ (\text{lam} \ M) \ (A \rightarrow B) \ :\ -\ \\
\quad \pi \ x \ \text{\ typ} \ x \ A \Rightarrow \text{typ} \ (M \ x) \ B.\]

?- \text{typ} \ (\text{lam} \ f \ \text{\ lam} \ y \ \text{\ app} \ f \ y) \ T

\[\text{typ} \ g \ A \quad \text{?- typ} \ (\text{lam} \ y \ \text{\ app} \ g \ y) \ B\quad T := A \rightarrow B\]
\[\text{typ} \ g \ A, \ \text{typ} \ x \ C \quad \text{?- typ} \ (\text{app} \ g \ x) \ D\quad B := C \rightarrow D\]
\[\text{typ} \ g \ A, \ \text{typ} \ x \ C \quad \text{?- typ} \ g \ (A' \rightarrow D), \ \text{typ} \ x \ A'\quad A := A' \rightarrow D\]
\[\text{Answer: T := (C --> D) --> C --> D} \quad A' := C\]
proves (_, true).

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

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

proves (Gamma, exists F) :- sigma X \ 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. \( \text{lam } x \ \text{\app} (M \ x) \ N \)
  - 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. \( \text{lam } (s \rightarrow t) \ h \ \text{\lam} s \ a \ \text{\X} h \ a \)

  \[ h: s \rightarrow t, a: s |- X h a : t \]
, λProlog Diverges on Partial Objects

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 (app (M x) x) T.
typ x nat ?- typ (M x) (A → T), typ x A.

BAD:  M x := app (M’ x) (N’ x)
       M’ x := app (M” x) (N” x)
...
Constrained Higher Order Logic

Proposal: extend λProlog 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 → T) } is delayed
Constrained Higher Order Logic

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

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

- delayed goals are fired when the guard becomes true

E.g. if $M := \lambda x. x$
then $\{ typ x \text{ nat } ?- \text{ typ x } (A \rightarrow T) \}$ fired
Constrained Higher Order Logic

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

E.g. unicity of typing

\{ \Gamma_1 ?- \text{typ } T \; \text{TY}_1 \} \Rightarrow
\begin{align*}
\{ \Gamma_2 ?- \text{typ } T \; \text{TY}_2 \} & \iff \\
\text{restrict } \Gamma_1 \; T = \text{restrict } \Gamma_2 \; T \land \\
\{ \; - \; \text{TY}_1 = \text{TY}_2 \}.
\end{align*}

- \{ \text{typ } x \; \text{nat}, \; \text{typ } y \; \text{bool} \; - \; \text{typ } (M \; x \; y) \; (\text{nat} \to A), \; \\
\text{typ } z \; \text{bool}, \; \text{typ } w \; \text{nat} \; - \; \text{typ } (M \; w \; z) \; (B \to \text{nat}) \} \Rightarrow \\
\{ \text{typ } x \; \text{nat}, \; \text{typ } y \; \text{bool} \; - \; \text{typ } (M \; x \; y) \; (\text{nat} \to A), \; \\
\; - \; B \to \text{nat} = \text{nat} \to 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)
\textbf{λProlog Cannot Print Partial Objects}

\begin{verbatim}
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! */
\end{verbatim}
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 := false, but the ! kills it.
Matching mode for λProlog

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

\text{print (app M N) S12 :-}
\text{\hspace{1cm} print M S1, print N S2, append [S1," ",S2] S12.}

\text{print @(M,L) S :- … /* do something */}
...

• The first argument is matched, not unified
• @$\equiv X\ t1\ ..\ tn$\ via\ $M := X,\ L := [t1,..,tn]$
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 λProlog 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 λProlog
  - type checker for Automath
  - 40 times slower than compiled OCaml code
  - 3 times slower than interpreted OCaml code
Work in Progress

• Implementation of λProlog 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)