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 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 implenetrable 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 → Y : B (Y output)

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

	x : A → F x : B
X : nat → int_of_nat X : int	L : list A → map A B (λx. Fx) : list B

	A : Type → G : SemiGroup B : Type → H : SemiGroup
nat : Type → (nat,0,1,+,*) : SemiGroup	A x B : Type → G x H : SemiGroup

AMK in the small: Canonical Structures

- Associate/extract functions/lemmas from types (e.g. algebraic structures)
  - Without:

    plus_comm: $\forall n,m$: nat. $n + m = m + n$

    Z_plus_comm: $\forall x,y$: int. $x + y = y + x$

    union_comm: $\forall U$: 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
  ; ⋆ : 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 */
  }

AMK in the small: Canonical Structures

to prove $2 + n = n + 2$

apply lemma comm

- Comm :

  $\forall M : CommMagma.$

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

- Unification problem :

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

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

  $M.C = nat \quad \wedge \quad M.\star = +$

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

$$\frac{M = \{ nat, +, \ldots \}}{M.C \cong nat}$$

$$\frac{G.C \cong A \quad H.C \cong B \quad M = G \times H}{M.C \cong A \times B}$$

$$\frac{M = \{ nat, +, \ldots \}}{M.\star \cong +}$$

$$\frac{\llparenthesis S1 \rrparenthesis \cong E \quad \llparenthesis S2 \rrparenthesis \cong F \quad S = Plus(S1,S2)}{\llparenthesis S \rrparenthesis \cong E + F}$$

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)

$$\frac{!}{\text{real\_tac}, (\Gamma, P\ |X|, \Delta \vdash G) \Rightarrow \text{cases}\ (X \geq 0 \lor X < 0)\ ;\ \text{real\_tac}}$$

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

$$\text{normalize\_correct} : \forall S\text{: syntax.}\ ⦅ \eta\ S ⦆ = ⦅ S ⦆$$

Algorithmic knowledge in user space: a language proposal

One Language to Bind Them All

- Binders, scope, α-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?

λProlog

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

```
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 \ app f x) y) O.
O := f y
```

λProlog

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

```
Q ::= sigma X \ Q | Q,Q | Q;Q | x t .. t
      | pi x \ Q | C => Q
C ::= pi x \ C | C,C | x t .. t
      |  Q => C
```

- pi x \ Q : introduces a new eigenvar x
- C => Q : assumes C to prove Q

Simply Typed λ-calculus in λProlog

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

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

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

$$\frac{\Gamma, x{:}A \vdash M[x/y] : B}{\Gamma \vdash \lambda y.\ M : A \to B}$$

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

Simply Typed λ-calculus in λProlog

```
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 (lam f \ lam y \ app f y) T	
typ g A	?- typ (lam y \ app g y) B	T := A --> B
typ g A, typ x C	?- typ (app g x) D	B := C --> D
typ g A, typ x C	?- typ g (A’ → D), typ x A’	A := A’ --> D
		A’ := C

Answer: T := (C --> D) --> C --> D

FO Prover in λProlog

```
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.   lam x \ 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.   lam (s → t) h \ lam s a \ X h a

  h: s → 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 → T) } delayed

- delayed goals are fired when the guard becomes true
  E.g. if   M := λx. 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
  { Gamma1 ?- typ T TY1 } =>
  { Gamma2 ?- typ T TY2 } <=>
  restrict Gamma1 T = restrict Gamma2 T /\
  { ?- TY1 = TY2 }.

- { typ x nat, typ y bool ?- typ (M x y) (nat → A),
  typ z bool, typ w nat ?- typ (M w z) (B → nat) } ==>
  { typ x nat, typ y bool ?- typ (M x y) (nat → A),
  ?- B → nat = nat → 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 := false, but the ! kills it.

Matching mode for λProlog

```
$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) ≅ X t1 .. tn   via M := X, L := [t1,..,tn]

Extensions to λProlog

- 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)