Towards an Extensible Proof Assistant that can be Analyzed

Claudio Sacerdoti Coen
(on an ongoing work with E. Tassi & D. Miller)

University of Bologna

17 Jul 2014 - Wien (QED20)

This talk

1. State of the art
2. Towards a domain specific language to write proof assistants
3. Conclusions and future work

State of the art

The Calculus of (Co)Inductive Constructions (CIC)

A λ-calculus typed à la Church featuring

- Dependent types: terms (proofs) can occur in types
- Computation: a function expecting a type T_1 accepts an argument of type T_2 if they are the same up to reduction
- Inductive types
- Functions defined by structural recursion

How are Coq/Matita/Agda implemented?

Curry-Howard

Curry-Howard:

1. Formulas are types
2. Proofs are terms (and are portable)
3. Proof checking is type checking (and require reductions)

Kernel:

1. Implements proof checking: yes/no result
2. Shields the rest of the system
3. Irrelevant for the user

Terms:

1. Highly redundant.
   Example: $[1;2]$ is $(\mathit{cons}\ \mathbb{N}\ 1\ (\mathit{cons}\ \mathbb{N}\ 2\ (\mathit{nil}\ \mathbb{N})))$

Curry-Howard for Partial Proofs

Curry-Howard for partial proofs:

1. Partial proofs are partial terms (terms with holes/placeholders/metavariables)
2. Metavariables used for:
   - open goals/conjectures
   - omitted redundant information (unique and computable)
   - omitted non redundant information (e.g. middle term in transitivity)
3. "Type inference" (refinement): type checks and alters the term, imposing constraints on metavariables
   Example (sufficient constraints):
   $(cons\ ?\ 1\ (cons\ ?\ 2\ (nil\ ?)))$: all ? must be $\mathbb{N}$
4. Metavariable instantation = proof progress

Curry-Howard for Partial Proofs

Constrained metavariables:

1. Fully constrained $\Rightarrow$ can be istantiated
2. Partially constrained
   - 2.1 Sufficiently constrained $\Rightarrow$ postponed, a proof obligation
   - 2.2 Not sufficiently constrained
     - 2.2.1 Failure
     - 2.2.2 Let the user suggest an instantiation

Example (sufficient constraints): assume

$$\cdot/\cdot : \mathbb{Q} \to \forall d : \mathbb{Q}.d \neq 0 \to \mathbb{Q}$$

Checking $\lambda x : \mathbb{Q}.\lambda y : \mathbb{Q}.\lambda H : y * x \neq 0.P(x/_? y)$ forces

$$x : \mathbb{Q}, \quad y : \mathbb{Q}, \quad H : y * x \neq 0 \quad \vdash \quad ? : y \neq 0$$

Curry-Howard for Partial Proofs

Constrained metavariables:

1. Fully constrained $\Rightarrow$ can be istantiated
2. Partially constrained
   - 2.1 Sufficiently constrained $\Rightarrow$ postponed, a proof obligation
   - 2.2 Not sufficiently constrained
     - 2.2.1 Failure
     - 2.2.2 Let the user suggest an instantiation

Example (insufficient constraints): assume

$M : \mathbb{N}$

$f : (\textbf{match}\ ?\ \textbf{with}\ \mathit{true}\ \Rightarrow \mathbb{N} \mid \mathit{false}\ \Rightarrow \mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$

Checking $f\ M$ forces

$$\textbf{match}\ ?\ \textbf{with}\ \mathit{true}\ \Rightarrow \mathbb{N} \mid \mathit{false}\ \Rightarrow \mathbb{N} \times \mathbb{N} \simeq \mathbb{N}$$

Curry-Howard for Partial Proofs

Refiner:

1. Implements type inference/constraining
2. Input: a partial term; Output: constraints (goals)
3. Responsible for the perceived intelligence of the system
   - Non stupid: infer the uniquely constrained metavariables
   - Intelligent: autonomously pick the right choice when constraints are not strict enough

E.g. Type classes: $size\ ?\ [[1]; [2; 3]]$ where

$size : \forall S : SIZEABLE.carr\ S \to \mathbb{N}$
$SIZEABLE = \{carr : Type;\quad size : carr \to \mathbb{N}\}$

Constraint: $carr\ ? \simeq [[1]; [2; 3]]$. Many solutions! What is right?

Teaching an intelligent system

An intelligent system is a system you can teach to!

Unification hints:

$$\frac{? := \{carr = \mathbb{N}; size = \lambda\_.1\}}{carr\ ? \simeq \mathbb{N}}$$

$$\frac{carr\ ?_2 \simeq X \qquad ? := \{carr = list\ X; size = sum\_over\_list\ (size\ ?_2)\}}{carr\ ? \simeq list\ X}$$

The system is extended with knowledge from the library!

More Teaching

Major recent advancements in Coq/Matita via teaching:

1. Unification hints
   1.1 To infer data (cfr. canonical structures)
   1.2 To infer proofs (cfr. Gonthier's mathematical components)
2. Canonical structures (special case of unification hints)
3. Coercions: how to turn any value of type T_1 into a T_2

$$\frac{f : X \longrightarrow Y}{set\_of\_list\ f : list\ X \longrightarrow set\ Y}$$

4. Non uniform coercions: how to turn a specific value x of type T_1 into a value y of type T_2

$$\frac{}{\mathbb{N} : Type \mapsto \{\mathbb{N}, 0, +\} : Monoid}$$

5. User definable tactics (e.g. LTac)

The BIG question

What is the semantics of user extension languages?

Extensions in a ad-hoc, declarative languages of heuristic based undocumented algorithms written in a functional language.

Towards a domain specific language to write proof assistants

What features for the programming language?

Extension languages (wishlist):

1. Declarative flavour
2. Need sophisticated but clear semantics of backtracking
3. Formulae with binders and metavariables as first class objects
4. Clear semantics (for static analysis, e.g. termination)

Refiner implementation (now):

1. Impure functional language
2. Limited local backtracking or super-complex monads
3. Binders and metavariables handled by hand via De Brujin indexes (implementing tactics/heuristics very painful)

Glimpses of a promised land

Why suffering when implementing the refiner too?

Proposal:

Implement the refiner and the user extensions in the same domain specific language

Benefits:

1. greatly simplify the implementation
2. code as close as possible to pen&paper description of the algorithm
3. static checks, possibility to certify the algorithm

Can we afford the slow-down? (open question)

What domain specific language?

A promising one: λProlog

- Developed by Milner, Nadathur et alt.
- Like Prolog, but for a very large fragment of first order logic
- Binders as first class objects (need higher order unification)
- Cuts like in Prolog
- Clear semantics (if cuts are avoided and unification problems are in the decidable pattern calculus)
- Abella: a theorem prover for λProlog programs

Example:
$type\_of\ (\lambda x.M)\ (A \to B) \dashv \forall x.\ type\_of\ x\ A\ \Rightarrow type\_of\ M\ B$

Not the right solution yet

Right language iff allows shallow embedding.

1. Can λProlog binders be used for CIC binders? Yes!
2. Can λProlog metavariables be used for CIC binders? No!
   - 2.1 Query: *type_of* X Y. will trigger full proof search
     $\mathit{type\_of}\ (\lambda x.M)\ (A \to B) \dashv \forall x.\ \mathit{type}_o f\ x\ A \Rightarrow \mathit{type\_of}\ x\ B$
     type_of X Y needs to be DELAYED (turned into a proof obligation)
   - 2.2 Full higher order unification: $X\ 0 \simeq 0 + 0$
     Four solutions: $\{\lambda x.0 + 0, \lambda x.x + 0, \lambda x.0 + x, \lambda x.x + x\}$

     λProlog: delays all flexible rigid unifications outside the decidable pattern fragment
     Coq/Matita: apply heuristics and limited backtracking

Our "λProlog"

- Embeddable in Matita/Coq as DSL
  1. Interpreter implemented in OCaml
  2. Can call primitives implemented in OCaml
     E.g.: kernel and library invocation, pretty-printing, etc.
- New constructs:
  1. Labelled clauses for grafting user extensions in the right place E.g. : $heuristic_1 < rule_3 : clause$
  2. **delay** X P **on** *filter*
     - freezes X to a constant to avoid accidental instantiation (unless X is already frozen)
     - delays the goal P
     - tabulates the goal $P[\bar{X}/\bar{x}] \dashv H_1[\bar{X}/\bar{x}], \ldots, H_n[\bar{X}/\bar{x}]$ where the H_i are the clauses in context filtered by filter and the constants $\bar{x}$ are generalized
       Example: If $X : T$ is delayed, every occurence of X must have type T as well

Our "λProlog"

- New constructs:
  1. **resume** X F
     - 1.1 Thaws the frozen variable X
     - 1.2 Resumes all goals P delayed on X after executing F (to pass information)
       Example: **resume** X $(X = \lambda x.x + x)$
  2. $(\#X\ L)$ matches the application of a frozen variables, binds the argument list to L

Example: implementation of higher order unification

$$\begin{array}{l} \mathit{unify}\ (\#X\ L)\ M \dashv \\ \quad (\forall x.\mathit{bind}\ x\ L \Rightarrow \mathit{abstract}\ M\ (T\ x)), \\ \quad \textbf{resume}\ X\ (X = T) \end{array}$$

where $\mathit{bind}\ x\ 0 \Rightarrow \mathit{abstract}\ (0+0)\ (T\ x)$ is implemented to yield $T\ x = 0 + 0$ or $T\ x = 0 + x$, etc.

Our "λProlog"

- New constructs:
  1. $**delayed** L binds L to the list of open goals (frozen metavariables)
  2. $**schedule** X F resumes all clauses delayed on X after assuming F (to pass information)

Implementation of the proof engine:

1. $**delayed** turns proof obligations into first class objects to show them to the user and implement tacticals
2. $**schedule** to apply a tactic to a given goal

Conclusions and future work

Where are we now

What we have:

- elpi (embedded "λProlog" interpreter)
  - $\approx$ 2500 lines of OCaml
  - embedding friendly (easy to make external calls)
- refinement and unification algorithms for $\lambda\mathrm{P} + \mathbb{N}$, completed with unification hints, coercions, non uniform coercions, quasi linear overloading algorithm, simple tactic engine and top-level, basic tactics in

only 519 lines of elegant "λProlog" code

Future work

- better understanding of new constructs
- general (co)inductives types, universes
- embed in Matita to quantify the performance penalty
  $10\times$ slower would be acceptable,
  cfr. major speed-up from parallelization in UITP
- static analysys of "λProlog" programs
- interactive verification of "λProlog" programs

The End

Thanks for your attention!