Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)

Enrico Tassi

Elpi: an extension language for Coq

Metaprogramming Coq in the Elpi λProlog dialect

Enrico Tassi
Université Côte d'Azur, Inria
France
Enrico.Tassi@inria.fr

Abstract

Elpi is dialect of λProlog that can be used as an extension language for Coq. It lets one define commands and tactics in a high level programming language tailored to the manipulation of syntax trees containing binders and existentially quantified meta variables.

Keywords Coq, λProlog, extension language, metaprogramming

1 Introduction

Extension languages are key to the development of software. From text editors to video games, high level extension languages contribute to the success of software by letting *users* improve the software and adapt it to their needs. Coq is no exception: it comes with an extension language for tactics called $\mathcal{L}_{tac}$ [1] that proved to be extremely useful, well beyond the "small automation" target envisioned by its author. Despite its success $\mathcal{L}_{tac}$ has many limitations: it can define only tactics, it has no data structures, its binding discipline and evaluation order are unclear, etc... As a consequence alternative approaches to extend Coq (or similar systems) were proposed in recent years [3, 4, 6].

We present a Coq plugin embedding Elpi [2], a dialect of λProlog [5] designed to be used as an extension language for software doing symbolic computations on syntax trees containing binders and existentially quantified meta variables (evars).

Unlike most of the alternatives cited above Elpi departs from the idea of meta programming Coq in itself (i.e. in Gallina) in favor of a domain specific, higher level, language that hides to the programmer De Buijn indexes and most of the intricacies of evars. It also provides first class search with backtracking and means to control it (e.g. the cut operator).

2 Elpi

The interpreter for Elpi (that stands for Embeddable Lambda Prolog Interpreter) is developed by C. Sacerdoti Coen and E. Tassi [2]. Its aim is to provide a programming platform to study the so-called elaborator component of an interactive prover as Coq or Matita. It implements a dialect of λProlog where not only the binders of the meta language (as in HOAS), but also its unification variables, can be used to model the object language ones.

We present Elpi by escalating λProlog's "hello world" example (a type checker for $\lambda_{\rightarrow}$) to an elaborator manipulating open terms.

```
type arr  ty -> ty -> ty.              % the arrow type constructor
type lam  (term -> term) -> term.   type app  term -> term -> term.
of (app F X) T :- of F (arr S T), of X S.          % clause for app
of (lam F) (arr S T) :- pi x\ of x S => of (F x) T. % clause for lam
```

Remark that there is no term constructor for variables: λProlog provides operators to introduce fresh constants (names) and to augment the program with hypothetical clauses (clauses about fresh constants).

For example, to the query "which is the type of $\lambda x \lambda y.x$?", written `of (lam x\lam y\ x) Ty`, Elpi answers `Ty = arr S1 (arr S2 S1)` by using the second clause twice. Operationally each time a clause is used its variables (capital letters) are made fresh and the head is unified with the query. In our example the second clause's head, `of (lam F1) (arr S1 T1)`, is unified with the query giving `F1 = x\lam y\ x` and `Ty = arr S1 T1`. Then a fresh constant c_1 is created by the `pi` operator, and the program is augmented with the hypothetical clause `(of` c_1 `S1)` by the `=>` operator. The sub query is about `(F1` c_1`)`, but since `F1` is a function such a term is equivalent to `(lam y\` c_1`)`, i.e. the body of the first lambda where the bound variable has been replaced by c_1. The second clause is used again: this time we have a fresh c_2, a new clause `(of` c_2 `S2)` about it, `T1` gets `arr S2 T2` and consequently `Ty` becomes `arr S1 (arr S2 T2)`. Finally we perform the sub query `(of` c_1 `T2)`. At that stage there is no choice but to use the `(of` c_1 `S1)` hypothetical clause, and unification imposes `T2 = S1`, i.e. the output type to coincide with the type of the first input. This yields `Ty = arr S1 (arr S2 S1)`

2.1 A dialect of λProlog with constraints

When a syntax tree is partial, i.e. it contains unassigned unification variables, λProlog programs naturally perform search: the generative semantics inherited from Prolog enumerates all possible instantiations until one that fits is found. Unfortunately this is not (always) welcome when manipulating partial terms: To the query `of (lam x\ lam y\ F x y) Ty` Elpi replies by finding an arbitrary solution for `F`:

```
F = x\ y\ y                Ty = arr S1 (arr S2 S2)
```

This solution is generated because the sub query `(of (F` c_1 c_2`) T2)` finds a possible solution in the hypothetical clause `(of` c_2 `S2)`.

To avoid that, Elpi lets one declare that a predicate should never instantiate one of its arguments (morally an input) and turn a goal into a constraint when the input is flexible.

```
pred of i:term, o:ty.
of (?? as F) T :- declare_constraint (of F T) on F.
```

The query above now results in the following solution, that comprises a (typing) constraint on `F` (the key of the constraint).

`Ty = arr S1 (arr S2 T2)` `of` c_1 `S1, of` c_2 `S2 ?- of (F` c_1 c_2`) T2 on F`

Constraints are resumed as soon as the key is assigned. If `nat` and `bool` are types, then the following query succeeds.

```
Ty = arr nat (arr bool bool), of (lam x\ lam y\ F x y) Ty,
not (F = a\b\a),   % the resumed constraint makes this assignment fail
F = a\b\b          % this one is accepted, since the type matches
```

Finally, Elpi also provides a language of rewriting rules (inspired from Constraint Handling Rules (CHR)) to manipulate the store of constraints (e.g. to keep the constraint store duplicate free).

CoqPL'18, January 13, 2018, Los Angeles, California, United States
2018.

3 HOAS for Gallina terms

For lack of space we omit the declaration of all Gallina's term constructors. Instead we comment the HOAS description of addition over natural numbers.

```
fix `add` 0 (prod `n` (indt "nat") _\ prod `m` (indt "nat") _\ indt "nat")
 add\ lam `n` (indt "nat") n\ lam `m` (indt "nat") m\
        match n (lam `n` (indt "nat") _\ indt "nat")
        [m, lam `p` (indt "nat") p\ app [indc "S", app [add, p, m]]]
```

Global names are shortened for the sake of conciseness and are between double quotes, like `"nat"`, while pretty printing hints are between single quotes such as `` `add` ``. Dummy λ-abstraction is written `_\` (i.e. no variable name). The second argument of `fix` is the position of the decreasing argument. `indt` and `indc` respectively represent inductive types and their constructors. `const "plus"` represents the global constant to which the entire term is associated. Finally that the `lam` node carries extra arguments with respect to Section 2.

3.1 Quotations and anti-quotations

The syntax of terms used above closely reflects the data type of terms internally used by Coq and is quite verbose. Indeed, Coq users rarely interact directly with this data type, thanks to the sophisticated notation engine provided by the system. The notation engine is exposed to Elpi via a system of quotations and anti-quotations. For example the term `(fun x : nat => x + S x)` can be built and assigned to the variable `T` by the following query.

```
T = {{fun x : nat => x + lp:F x}}, F = x\ app[{{S}}, x]
```

Note the anti-quotation `lp:F` that lets one put an Elpi term in the middle of a Coq one.

4 Commands

Elpi comes with APIs to access the environment of Coq, for example to read/write inductive declarations or constants.

As an example we implement a command that counts the number of constructors of an inductive type and defines a constant containing such number as a term of type `nat`.

```
Elpi Command count_constructors <<      % Elpi code in a Coq document
 int->nat 0 {{0}}.
 int->nat N {{S lp:X}} :- M is N - 1, int->nat M X.
 main [IndName, ResName] :-
  coq-locate IndName (indt GR),            % find inductive by name
  coq-env-indt GR _ _ _ _ Kn _,            % get constructors' names
  len Kn N, int->nat N Nnat,               % count and convert to nat
  coq-env-add-const ResName Nnat {{nat}}.  % define new constant
>>.
Elpi count_constructors bool nK_bool.  (* command invocation       *)
Print nK_bool.                         (* prints nK_bool = 2 : nat *)
```

Applications During an internship Luc Chabassier wrote a program to derive boolean comparison functions (and their correctness proofs) from inductive type declarations. It covers inductive types with (even non uniform) parameters, but no indexes (circa 400 LOC). Cyril Cohen implemented a binary parametricity transformation for terms and inductive data types (circa 250 LOC). Derivation of induction principles is work in progress.

5 Tactics

The proof term construction engine of Coq is based on existentially quantified metavariables (evar): Proof construction proceeds by assigning to the evar corresponding to a (missing) proof of the current goal a term that can contain new evars representing subgoals.

In the spirit of HOAS, Elpi encodes Coq's evars as λProlog unification variables with a pending typing constraint on them. The proof construction state (`evar_map` in Coq's slang) is hence modelled by the set of constraints. Whenever an evar e gets instantiated with a term t, the corresponding typing constraint is resumed making the assignment fail if the type of t does not match the type of the e. Constraint Handling Rules are used to keep the constraint set duplicate free: Every time two typing constraints are keyed on the same evar, one of the them is turned into an immediate check for compatibility (unification) of the two types.

A tactic invocation on the goal on the left is translated to (roughly) the query on the right

```
T : U          pi t\ decl t `T` U =>
x := V : T      pi x\ def x `x` V t =>
==========       declare_constraint (of (G t x) P) on (G t x),
P                solve (goal [decl t `T` U, def x `x` V t] (G t x) P)
```

where `of` is a predicate implementing an elaborator for Gallina's terms and `solve` is the main entry point for a tactic.

As an example we implement a `pattern-match` predicate that could play the role of $\mathcal{L}_{tac}$'s "match goal with" primitive. We then use such a predicate to implement a tactic that solves the goal only if the context contains two distinct hypotheses that prove the goal.

```
Elpi Tactic example_tac <<              % Elpi code in a Coq document
pred copy i:term, o:term.       % never instantiate the first argument
copy (indc GR) (indc GR).
copy (app L) (app PL) :- map L copy PL.
copy (lam N F) (lam N PF) :- pi x\ copy x x => copy (F x) (PF x).
..
pmatch-hyp (decl X N Ty)  (decl X N PTy)   :- copy Ty PTy.
pmatch-hyp (def X N B Ty) (def X N PB PTy) :- copy B PB, copy Ty PTy.
pattern-match (goal Hyps _ Concl) PHyps PGoal Cond :-
  pmatch Concl PGoal,
  (forall PHyps p\ exists Hyps h\ pmatch-hyp h p),
  Cond.
solve ((goal _ G _) as Goal) :-
  pattern-match Goal [decl X NX T,decl Y NY T] T (not(X = Y)),
  coq-say "Both" NX "and" NY "solve the goal, picking " NX,
  G = X.
>>.
Lemma silly (x y : bool) (A : x = y) (H : True) (B : x = y) : x = y.
Proof. elpi example_tac. Qed.  (* prints: Both A and B solve... *)
```

Here `map`, `forall` and `exists` are just standard list iterators, while `copy` is a standard (*venerable*) λProlog predicate [5, page 199] that, given the mode declaration, implements matching. Remark how the condition `(not (X = Y))` forces Elpi to backtrack, since the first pick would be `NX = NY = `A``.

Applications Cyril Cohen is working on a tactic translating a program into its refined version for the CoqEAL project.

Download Elpi from https://github.com/LPCIC/coq-elpi, profit!

References

[1] David Delahaye. 2002. A Proof Dedicated Meta-Language. *Electr. Notes Theor. Comput. Sci.* 70, 2 (2002), 96–109.

[2] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. 2015. ELPI: fast, Embeddable, λProlog Interpreter. In *Proceedings of LPAR*. Suva, Fiji.

[3] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 34 (Aug. 2017), 29 pages.

[4] Gregory Malecha and Jesper Bengtson. 2016. Extensible and Efficient Automation Through Reflective Tactics. Springer-Verlag, New York, NY, USA, 532–559.

[5] Dale Miller and Gopalan Nadathur. 2012. *Programming with Higher-Order Logic* (1st ed.). Cambridge University Press, New York, NY, USA.

[6] Beta Ziliani, Yann Régis-Gianas, and Jan-Oliver Kaiser. 2017. The Next 700 Safe Tactic Languages. (2017).