

# HO unification from object language to meta language

Enrico Tassi

enrico.tassi@inria.fr

Université Côte d'Azur, Inria

France

Davide Fissore

davide.fissore@inria.fr

Université Côte d'Azur, Inria

France

## ABSTRACT

Specifying and implementing a logic from scratch requires significant effort. Logical Frameworks and Higher Order Logic Programming Languages provide dedicated, high-level Meta Languages (ML) to facilitate this task in two key ways: 1) variable binding and substitution are simplified when ML binders represent object logic ones; 2) proof construction, and even proof search, is greatly simplified by leveraging the unification procedure provided by the ML. Notable examples of ML are Elf [12], Twelf [13],  $\lambda$ Prolog [9] and Isabelle [19] which have been utilized to implement various formal systems such as First Order Logic [4], Set Theory [11], Higher Order Logic [10], and even the Calculus of Constuctions [3].

The object logic we are interested in is Coq's [17] Dependent Type Theory (DTT), for which we aim to implement a unification procedure  $=_o$  using the ML Elpi [2], a dialect of  $\lambda$ Prolog. Elpi comes equipped with the equational theory  $=_\lambda$ , comprising  $\eta\beta$  equivalence and higher order unification restricted to the pattern fragment [8]. We want  $=_o$  to feature the same equational theory as  $=_\lambda$  but on the object logic DTT. Elpi also comes with an encoding for DTT that works well for meta-programming [16, 15, 6, 5]. Unfortunately this encoding, which we refer to as  $\mathcal{F}_o$ , "underuses"  $=_\lambda$  by restricting it to first-order unification problems only. To address this issue, we propose a better-behaved encoding,  $\mathcal{H}_o$ , demonstrate how to map unification problems in  $\mathcal{F}_o$  to related problems in  $\mathcal{H}_o$ , and illustrate how to map back the unifiers found by  $=_\lambda$ , effectively implementing  $=_o$  on top of  $=_\lambda$  for the encoding  $\mathcal{F}_o$ .

We apply this technique to the implementation of a type-class [18] solver for Coq [17]. Type-class solvers are proof search procedures based on unification that back-chain designated lemmas, providing essential automation to widely used Coq libraries such as Stdpp/Iris [7] and TLC [1]. These two libraries constitute our test bed.

## KEYWORDS

Logic Programming, Meta-Programming, Higher-Order Unification, Proof Automation

### ACM Reference Format:

Enrico Tassi and Davide Fissore. XXXX 2024. HO unification from object language to meta language. In *YYY*. ACM, New York, NY, USA, 5 pages. <https://doi.org/ZZZZZZZZZZZZ>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*Conference'17, July 2017, Washington, DC, USA*

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

<https://doi.org/ZZZZZZZZZZZZ>

## 1 INTRODUCTION

Specifying and implementing a logic from scratch requires significant effort. Logical Frameworks and Higher Order Logic Programming Languages provide dedicated, high-level Meta Languages (ML) to facilitate this task in two key ways: 1) variable binding and substitution are simplified when ML binders represent object logic ones; 2) proof construction, and even proof search, is greatly simplified by leveraging the unification procedure provided by the ML. Notable examples of ML are Elf [12], Twelf [13],  $\lambda$ Prolog [9] and Isabelle [19] which have been utilized to implement various formal systems such as First Order Logic [4], Set Theory [11], Higher Order Logic [10], and even the Calculus of Constuctions [3].

The object logic we are interested in is Coq's [17] Dependent Type Theory (DTT), and we want to code a type-class [18] solver for Coq [17] using the Coq-Elpi [16] meta programming framework. Type-class solvers are unification based proof search procedures that combine a set of designated lemmas in order to providing essential automation to widely used Coq libraries.

As the running example we take the Decide type class, from the Stdpp [7] library. The class identifies predicates equipped with a decision procedure. The following three designated lemmas (called Instances in the type-class jargon) state that: 1) the type  $\text{fin } n$ , of natural numbers smaller than  $n$  is finite; 2) the predicate  $\text{nfact } n \text{ nf}$ , linking a natural number  $n$  to its prime factors  $\text{nf}$ , is decidable; 3) the universal closure of a predicate has a decision procedure if the predicate has and if its domain is finite.

**Instance** `fin_fin n : Finite (fin n)`.

**Instance** `nfact_dec n nf : Decision (nfact n nf)`.

**Instance** `forall_dec A P : Finite A  $\rightarrow$`

`$\forall x:A, \text{Decision } (P \ x) \rightarrow \text{Decision } (\forall x, P \ x)$` .

Under this context the type-class solver is able to prove the the following statement automatically by back-chaining the three instances.

**Check** `_ : Decision (forall y: fin 7, nfact y 3)`.

The encoding of DTT provided by Elpi, that we will discuss at length later in sections 2 and 3, features the following term constructors:

**kind** `tm type`.

**type** `lam tm  $\rightarrow$  (tm  $\rightarrow$  tm)  $\rightarrow$  tm`. % lambda abstraction

**type** `app list tm  $\rightarrow$  tm`. % n-ary application

**type** `all tm  $\rightarrow$  (tm  $\rightarrow$  tm)  $\rightarrow$  tm`. % forall quantifier

**type** `c string  $\rightarrow$  tm`. % constants

Following this term encoding the three instances are represented by the following rules:

`finite (app [c "fin", N])`.

`decision (app [c "nfact", N, NF])`.

`decision (all A x\ app [P, x]) :- finite A,`

`pi x\ decision (app [P, x])`.

Unfortunately this direct translation of the instances considers the

TODO:  
ex-  
plain  
HOAS

TODO:  
ex-  
plain  
pi,  
cons

predicate  $P$  as a first order term. If we try to backchain the third rule on the encoding of the goal above:

```
decision (all (app[c"fin", c"7"]) y\
  app[c"nfact", y, c"3"]).
```

we fail because of this “higher order” unification problem (in DTT) is phrased as a first order unification problem in the meta language.

```
app[c"nfact", y, c"3"] = app[P, y]
```

In this paper we study a more sophisticated encoding of Coq terms allowing us to rephrase the problematic rule as follows:

```
decision (all A x\ Pm x) :- link Pm A P, finite A,
  pi x\ decision (app[P, x]).
```

This time  $Pm$  is a higher order unification variable (of type  $tm \rightarrow tm$ ).

The resulting unification problem is now:

```
app[c"nfact", y, c"3"] = Pm y
```

That admits one solution:

```
Pm = y\ app[c"nfact", y, c"3"]
A = app[c"fin", c"7"]
```

Elpi succeeds in the application of the new rule and then runs the premise  $link\ Pm\ A\ P$  that is in charge of bringing the assignment back to the domain of Coq terms (the type  $tm$ ):

```
P = lam A a\ app[c"nfact", a, c"3"]
```

This simple example is sufficient to show that the encoding we seek is not trivial. Indeed the solution for  $P$  generates a (Coq)  $\beta$ -redex in the second premise (under the  $pi$ ):

```
decision (app[lam A (a\ app[c"nfact", a, c"3"]), x])
```

In turn the redex prevents the second rule to backchain properly since the following unification problem has no solution:

```
app[lam A (a\ app[c"nfact", a, c"3"]), x] =
app[c"nfact", N, NF]
```

This time the root cause is that the unification procedure of  $=_A$  of the meta language is not aware of the equational theory of the object logic  $=_o$ , even if both theories include  $\eta\beta$ -conversion and admit most general unifiers for problems in the pattern fragment [8].

In this paper we discuss alternative encodings of Coq in Elpi 2, then we identify a minimal language  $\mathcal{F}_0$  in which the problems sketched here can be fully described. We then detail an encoding  $comp$  from  $\mathcal{F}_0$  to  $\mathcal{H}_o$  (the language of the meta language) and a decoding  $decomp$  to relate the unifiers bla bla..

## 2 ALTERNATIVE ENCODINGS

Our choice of encoding of DTT may look weird to the reader familiar with LF, since used a shallow encoding of classes and binders, but not of the “lambda calculus” part of DTT. Here a more lightweight encoding that unfortunately does not fit our use case

```
finite (fin N).
decision (nfact N NF).
decision (all A x\ P x) :-
  (pi x\ decision (P x)), finite A.
```

but in DTT this is not always possible and not handy in our use case, since the arity of constants is not fixed.

```
Fixpoint narr T n :=
  if n is S m then T -> narr T m else T.
Definition nsum n : narr nat (n+1).
```

```
Check nsum 2 8 9 : nat.
```

```
Check nsum 3 7 8 9 : nat.
```

moreover we use the same encoding for meta programming, or even just to provide hand written rules. We want to access the syntax of OL, so our embedding cannot be that shallow. We want to keep it shallow for the binders, but we need the  $c$ ,  $app$  and  $lam$  nodes.

Another alternative

```
decision X :- unif X (all A x\ app[P, x]),
  (pi x\ decision (app[P, x])), finite A.
```

gives up all half of what the ML gives us. Moreover even if  $unif$  here embodies the eq theory of DTT which is much stronger than the one of the ML, we don’t need it. According to our experience eta beta suffice, but HO is needed.

Note that this [3] is related and make the discrepancy between the types of ML and DTT visible. In this case one needs 4 application nodes. Moreover the objective is an encoding of terms, proofs, not proof search. Also note the  $conv$  predicate, akin to the  $unif$  we rule out.

This other paper [14] should also be cited.

## 3 LANGUAGES DESCRIPTION

In order to reason about unification of the terms of an objet language within a meta language, we start by formally describing the two languages. Employing meta-programming for this purpose, fig. 1 presents, on the left, the typing structure of object language terms along with the signature of the  $eq\_fo$  function.

In this encoding, the  $fo\_tm$  type stands for the first order representation of the object language terms. We model unification variables as integers corresponding to memory addresses. The memory is represented as list of optional terms (the  $subst\_fo$  type abbreviation). If the cell  $i$  is none, then the variable  $i$  is not instantiated, otherwise it has already been assigned to some value.

Because of our translation of the terms of the object language into the meta language outlined in section 1, the variables of the object language have no scope. An illustration of this observation can be found by analyzing deeply the subterm  $P\ x$  from the instance  $forall\_dec$ . Here,  $P$  is an higher order variables with type  $A \rightarrow Prop$  and  $x$  is a name bound to  $P$ . However, at the meta level, the translation of  $P\ x$  becomes  $app[uv\ 1, x]$ , that is,  $uv\ 1$  cannot reference  $x$ .

As our objective is to handle higher-order variables, a viable solution involves addressing this issue through the compilation of the term received as input into a more expressive version. This can be achieved by employing a second type structure for the terms of the OL, as outlined in the second column of fig. 1. The primary difference lies in the fact that a unification variable (always identified by an integer) now takes a list of terms representing its scope. The abstractions of the ML are symbolized using the constructor  $abs$  of the type  $assmt$ . The constructor  $val$  is utilized to contain terms of the object language, and in this context, there is no need for an application node in the ML.

The equality relation over terms under a given substitution mapping is possible thanks to the  $equal$  predicate. There exist two version of this predicate:  $equal\_fo$  and  $equal\_ho$ . Their implementation is given in fig. 2.

```

kind fo_tm type.
type fo_app list fo_tm -> fo_tm.
type fo_lam (fo_tm -> fo_tm) -> fo_tm.
type fo_uv nat -> fo_tm.
type fo_c string -> fo_tm.

kind tm type.
type app list tm -> tm.
type lam (tm -> tm) -> tm.
type uv nat -> list tm -> tm.
type c string -> tm.

kind assmt type.
type abs (tm -> assmt) -> assmt.
type val tm -> assmt.

typeabbrev subst list (option fo_tm). typeabbrev subst list (option assmt).
type eq_fo subst -> fo_tm -> fo_tm -> o. type eq subst -> tm -> tm -> o.

```

Figure 1: Language description

```

% deref
equal S (uv N) T1 :- assigned? N S T, equal S T T1.
equal S T1 (uv N) :- assigned? N S T, equal S T1 T.
equal S (app [uv N|A]) T1 :- assigned? N S F, beta F A T, equal S T T1.
equal S T1 (app [uv N|A]) :- assigned? N S F, beta F A T, equal S T1 T.
% congruence
equal S (app L1) (app L2) :- forall2 (equal S) L1 L2.
equal S (lam F1) (lam F2) :- pi x\ equal S x x => equal S (F1 x) (F2 x).
equal _ (c X) (c X).
equal _ (uv N) (uv N).
% beta
equal S (app [lam X | TL]) T :- beta (lam X) TL T', equal S T' T.
equal S T (app [lam X | TL]) :- beta (lam X) TL T', equal S T T'.
% eta
equal S (lam F) T :- not (T = lam _),
  pi x\ beta T [x] (T' x), equal S (lam F) (lam T').
equal S T (lam F) :- not (T = lam _),
  pi x\ beta T [x] (T' x), equal S (lam T') (lam F).


---


% deref
equal S (uv N A) T1 :- assigned? N S F, beta-abs F A T, equal S T T1.
equal S T1 (uv N A) :- assigned? N S F, beta-abs F A T, equal S T1 T.
% congruence
equal S (app L1) (app L2) :- forall2 (equal S) L1 L2.
equal S (lam F1) (lam F2) :- pi x\ equal S x x => equal S (F1 x) (F2 x).
equal _ (c X) (c X).
equal _ (uv N A1) (uv N A2) :- forall2 (equal S) A1 A2. %fixme in the code

```

Figure 2: Term equality

The first tests if two terms are equal in the object language, that is, given a substitution  $\theta$ , is it true that the two terms are equal? Note that no variable instantiation is done: unification is not performed by the `equal` predicate. Therefore, a unification variable  $i$  equals to another term  $t$  if  $t$  is a variable with same index  $i$ , or if  $i$  is assigned to  $t'$  in  $\theta$  and `equal`  $t' t$ . Moreover, since `equal_fo` represents the equality of the OL, and since the OL accept term equality up to  $\eta\beta$ , then `equal_fo` also quotient over these reductions.

The `equal_ho` predicate tests if two terms are equal in the world of the meta languages. This equality function is not capable to understand the  $\eta\beta$  reductions over terms of the object language. The ML sees a term of the type `tm` as atom on which only structural

equality can be performed. It is only possible to dereference a variable if assigned.

### 3.1 Compilation: `fo_tm` $\rightarrow$ `tm`

- how we transform an `fo_tm` in `tm` - the role of links

### 3.2 Unification in ML

- we accept HO unif with PF

### 3.3 Term de-compilation

TODO:  
say  
HO is  
same  
as ML

### 3.4 implementation

l'HO encoding e' esattamente lambda Prolog/elpi, il compilatore in pratica potrebbe essere scritto in un meta language, qui lo si presenta in elpi stesso. HO e FO in questo paper sono deep embedded in elpi per parlarne, ma in pratica il nostro solever, prendere XX, scrivere la clausola compilata.

### 4 RECOVERING ETA

```
q (all x\ F x) = q (all x\ app[f,x]) / N p f = p F
F = fun a => app [f,a] ----> F = f
```

l'utente da p su f, mentre l'istanza pe q forza F a fun ..

### 5 RECOVERING BETA

```
q (all x\ F x) = q (all x\ app[f,x,x]) / N p1 (app[f,a,a]) = p1 (app[f,a])
F = fun y => app [f,y,y] ----> (app[F,a]) / N app[f, a, a].
```

qui la sintesi di F puo generare un beta redex, quindi ci mettiamo p1 F1, e decomp beta F [a] F1.

### 6 RECOVERING ETA-BETA WITHIN UNIFICATION (NON LINEAR VARIABLES)

se i problemi di cui sopra avvengono nello stesso termine

```
q2 (all x\ F x) (app[F,a]) = q2 (all x\ app[f,x,x]) (app[f,a,a])
```

bisogna legare le due F e poi unificare le soluzioni tra di loro

### 7 HEURISTIC / BINARY APP

fo approx / sub pattern fragment

```
p (all x\ app[F,x,a]) (app[F,b]) = p (all x\ app[f,x,x,a]) (app[f,b])
p (all x\ G x) F' =
G = x\ f x x a
F = lam x\ f x x
F' = (app[f,b,b])
link (F a) F'
link G F
(app (app F x) a) = (app (f x x) a)
```

### REFERENCES

- [1] Arthur Charguéraud. “The Optimal Fixed Point Combinator”. In: *Interactive Theorem Proving*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 195–210. ISBN: 978-3-642-14052-5.
- [2] Cvetan Dunchev et al. “ELPI: Fast, Embeddable, λProlog Interpreter”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis et al. Vol. 9450. 2015, pp. 460–468. doi: 10.1007/978-3-662-48899-7\_32. URL: [http://dx.doi.org/10.1007/978-3-662-48899-7\\_32](http://dx.doi.org/10.1007/978-3-662-48899-7_32).
- [3] Amy Felty. “Encoding the Calculus of Constructions in a Higher-Order Logic”. In: ed. by M. Vardi. IEEE, June 1993, pp. 233–244. doi: 10.1109/LICS.1993.287584.
- [4] Amy Felty and Dale Miller. “Specifying theorem provers in a higher-order logic programming language”. In: *Ninth International Conference on Automated Deduction*. Ed. by Ewing Lusk and Ross Overbeck. 310. Argonne, IL: Springer, May 1988, pp. 61–80. doi: 10.1007/BFb0012823.
- [5] Davide Fissore and Enrico Tassi. “A new Type-Class solver for Coq in Elpi”. In: *The Coq Workshop 2023*. Bialystok, Poland, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- [6] Benjamin Grégoire, Jean-Christophe Lécenot, and Enrico Tassi. “Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin’s data types with Coq-Elpi”. In: *CPP ’23: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. Boston MA USA, France: ACM, Jan. 2023, pp. 167–181. doi: 10.1145/3573105.3575683. URL: <https://inria.hal.science/hal-03800154>.
- [7] RALF JUNG et al. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *Journal of Functional Programming* 28 (2018), e20. doi: 10.1017/S0956796818000151.
- [8] Dale Miller. “Unification under a mixed prefix”. In: *Journal of Symbolic Computation* 14.4 (1992), pp. 321–358. doi: 10.1016/0747-7171(92)90011-R.
- [9] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. doi: 10.1017/CBO9781139021326.
- [10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wen- zler. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7.
- [11] Lawrence C. Paulson. “Set theory for verification. I: from foundations to functions”. In: *J. Autom. Reason.* 11.3 (Dec. 1993), pp. 353–389. ISSN: 0168-7433. doi: 10.1007/BF00881873. URL: <https://doi.org/10.1007/BF00881873>.
- [12] F. Pfening. “Elf: a language for logic definition and verified metaprogramming”. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, California, USA: IEEE Press, 1989, pp. 313–322. ISBN: 0818619546.
- [13] Frank Pfenning and Carsten Schürmann. “System Description: Twelf – A Meta-Logical Framework for Deductive Systems”. In: *Automated Deduction – CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2.
- [14] Colin Rothgang, Florian Rabe, and Christoph Benz Müller. “Theorem Proving in Dependently-Typed Higher-Order Logic”. In: *Automated Deduction – CADE 29*. Ed. by Brigitte Pientka and Cesare Tinelli. Cham: Springer Nature Switzerland, 2023, pp. 438–455. ISBN: 978-3-031-38499-8.
- [15] Enrico Tassi. “Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq”. In: *ITP 2019 - 10th International Conference on Interactive Theorem Proving*. Portland, United States, Sept. 2019. doi: 10.4230/LIPIcs.CVIT.2016.23. URL: <https://inria.hal.science/hal-01897468>.
- [16] Enrico Tassi. “Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)”. In: *The Fourth*

- International Workshop on Coq for Programming Languages*. Los Angeles (CA), United States, Jan. 2018. URL: <https://inria.hal.science/hal-01637063>.
- [17] The Coq Development Team. *The Coq Reference Manual – Release 8.18.0*. <https://coq.inria.fr/doc/V8.18.0/refman>. 2023.
- [18] P. Wadler and S. Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. DOI: 10.1145/75277.75283. URL: <https://doi.org/10.1145/75277.75283>.
- [19] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. “The Isabelle Framework”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7.