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], λProlog [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 λProlog. 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, 4 pages. https://doi.org/ZZZZZZZZZZZZ

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

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], λProlog [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 `fin n`, of natural numbers smaller than `n` is finite; 2) the predicate `nfact n nf`, linking a natural number `n` to its prime factors `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).              (* r1 *)
Instance nfact_dec n nf : Decision (nfact n nf). (* r2 *)
Instance forall_dec A P : Finite A →              (* r3 *)
  ∀x:A, Decision (P x) → Decision (∀x:A, P x).
```

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

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

The encoding of DTT provided by Elpi, that we will discuss at length later in section **??** and **??**, is an Higher Order Abstract Syntax (HOAS) datatype `tm` featuring (among others) the following constructors:

```
type lam  tm -> (tm -> tm) -> tm.     % lambda abstraction
type app  list tm -> tm.              % n-ary application
type all  tm -> (tm -> tm) -> tm.     % forall quantifier
type c    string -> tm.               % constants
```

Following standard λProlog [9] the concrete syntax to abstract, at the meta level, an expression `e` over a variable `x` is `x\ e`, and square brackets denote a list of terms separated by comma. As an example we show the encoding of the Coq term "$\forall y$: t.nfact y 3":

```
all (c"t") y\ app[c"nfact", y, c"3"]
```

We now illustrate the encoding of the three instances above as higher order logic programming rules: capital letters denote rule parameters; `:-` separates the rule's head from the premises; `pi w\ p` introduces a fresh nominal constant `w` for the premise `p`.

```
finite (app[c"fin", N]).                             % r1
decision (app [c"nfact", N, NF]).                    % r2
decision (all A x\ app[P, x]) :- finite A,           % r3
  pi w\ decision (app[P, w]).
```

Unfortunately this direct translation of rule (r3) uses the predicate `P` essentially as a first order term for the meta language (its type is `tm`). If we try to backchain the rule (r3) on the encoding of the goal (g) above:

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

we fail because of this "higher order" unification problem (for DTT) is phrased as a first order unification problem in the meta language: the two lists of terms have different lengths!

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

In this paper we study a more sophisticated encoding of Coq terms allowing us to rephrase the problematic rule (r3) as follows:

```
decision (all A x\ Pm x) :- link Pm A P, finite A,   % r3a
  pi x\ decision (app[P, x]).
```

Since `Pm` is an higher order unification variable with `x` in its scope, the unification problem (pa) admits one solution:

```
app[c"nfact", y, c"3"] = Pm y                       % pa
Pm = x\ app[c"nfact", x, c"3"]        % assignment for Pm
A = app[c"fin", c"7"]                 % assignment for A
```

After unifying the head of rule (r3a) with the goal Elpi runs the premise `link Pm A P` that is in charge of bringing the assignment for `Pm` (that has type `tm -> tm`) 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, since the solution for `P` above generates a (Coq) β-redex in the second premise (the predicate under the `pi w`):

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

In turn this redex prevents the rule (r2) 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]
```

The root cause of the problems we face is that the unification procedure $=_\lambda$ 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].

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

2 PROBLEM STATEMENT AND ALTERNATIVE ENCODINGS

The equational theory of Coq's Dependent Type Theory is very rich. In addition to the usual $\eta\beta$-equivalence for functions, terms (hence types) are compared up to definition unfolding and fixpoint unrolling. Still, for efficiency and predictability reasons, most form of automatic proof search employ a unification procedure that captures a simpler one, just $\eta\beta$, and that solves higher order problems restricted to the pattern fragment $\mathcal{L}_\lambda$ [8]. We call this unification procedure $=_o$.

The equational theory of the meta language Elpi that we want to use to implement a form of proof automation is strikingly similar, since it it comprises $\eta\beta$ (for the meta language functions), and the unification procedure $=_\lambda$ solves higher order problems in $\mathcal{L}_\lambda$.

In spite of the similarity the link between $=_\lambda$ and $=_o$ is not trivial, since the abstraction and application term constructors the two unification procedure deal with are different. For example

`x\ f x`	$=_\lambda$	`f`
`lam A x\ app[c"f", x]`	$=_o$	`c"f"`
`lam A x\ app[c"f", x]`	$\neq_\lambda$	`c"f"`
`P x`	$=_\lambda$	`x`
`app[P, x]`	$=_o$	`x`
`app[P, x]`	$\neq_\lambda$	`x`

One could ignore this similarity, and "just" describe the object language unification procedure in the meta language, that is crafting a `unif` predicate to be used as follows in rule (r3):

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

This choice would underuse the logic programming engine provided by the metalanguage since by removing any datum from the head of rules indexing degenerates. Moreover the unification procedure built in the meta language is likely to be faster than one implemented in it, especially if the meta language is interpreted as Elpi is.

To state precisely the problem we solve we need a $\mathcal{F}_o$ representation of DTT terms, a $\mathcal{H}_o$ one. Then for any problem $s_1 =_o s_2$ for s_i in $\mathcal{F}_o$, we get a $t_1 =_\lambda t_2$ for s_i in $\mathcal{F}_o$ and a $L = \bigwedge_i \text{link } s_i\ t_i$ such that:

$$s_1 =_o s_2 \Leftrightarrow t_1 =_\lambda t_2 \wedge L$$
$$s_1 \neq_o s_2 \Rightarrow t_1 \neq_\lambda t_2 \vee \neg L$$

where comp $t_i = (s_i, \text{link } s_i\ t_i)$ and link is a predicate allowed to use $=_o$ on ground terms (hence equal, not unif)

These properties allow us to simulate a unification based backward search on DTT by using $=_\lambda$. In particular any unification that would work in DTT will work, and whenever a unification in DTT would have failed either the corresponding $=_\lambda$ or the link fails.

This grants that the trace of the logic program performing the search not only gives the same result, but also takes the same paths, that is it fails as early as possible.

2.1 Alternative encodings and related work

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.

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 the objet language within the meta language, we start by formally describing the two languages. Employing meta-programming for this purpose, fig. 1 presents the type `tm` containing the constructors for term application, lambda abstraction and constants. Moreover, in order to represent unification variables, we need to give two different constructors.

In the case of the OL, variables have no scope. For example, the subterm `P x` from the instance forall_dec. Here, `P` is an higher order variables with type `A → Prop` and `x` is a name bound to `P`. However, at the meta level, the translation of `P x` becomes `app[P, x]`, that is, `P` cannot reference `x`. In our encoding, unification variables are encoded as integer corresponding to memory addresses, and the constructor for unification variables in the OL is `fo_uv`. This constructor is prefixed with `fo` since the variable, having no scope, is a first-order variable.

On the other hand, in the case of the ML, we want to use its unification algorithm to make variable assignment. Since the ML is an Higher Order Programming Language, we represent unification variables with the `uv` constructor, which, this time, can see a list of terms. Of course some attention should be payed when dealing with this constructor, since we have to certify each time that an `uv i` remain in the pattern fragment, that is, the list of term in the scope of `i` is a list of distinct names. Finally, the following code

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

todo: explain better

illustrates the `assmt` representing variable assignment in the ML.

The memory of the two languages are represented with lists of substitutions. In particular, the substitution of the OL, called `fo_subst`, is made by optional terms such that, if the substitution is `none`, then the variable is not instantiated. Note that the variables in the `fo_subst` have always the `fo_uv` constructor. On the other hand, the ML substitution is an optional assignment and in that assignment, variables are considered to have the `uv` constructor.

A key property needed in unification is being able to verify if two terms are equal. This is kind of a structural equality verification between two terms, where variable dereferencing is performed when the variable is assigned. A sketch of the equality function is given for the OL language in fig. 2. Though, this equality relation over terms of a language can be powered by other reduction rules depending equational theory being considered. In our case, the OL terms are equal under $\eta\beta$ redex. This mean that new rules for those two redexes are added in the implementation of `fo_equal`.

If `fo_equal` is conceived to manage equality between terms of the OL, the same equality predicate in the ML behave slightly different. By the given definition of the ML, the ML allows $\eta\beta$ congruence of terms, but, since the node `app` and `abs` are constructor representing the applications and the abstractions of the OL, these two reduction rules cannot applied on them. We build therefore a predicate `equal` working for terms in the ML which is implemented merely with the rules for the `fo_equal` predicate. For example, if `fo_equal [] (abs x\ [c"f", c]) (c"f")` is true in the object language, `equal [] (abs x\ [c"f", c]) (c"f")` produces a failure.

not true for uv

The solution we are proposing aim to overcome these unification issues by 1) compiling the terms t and u of the OL into an internal version t' and u' in the ML; 2) unifying t' and u' at the meta level instantiating meta variables; 3) decompiling the meta variable into terms of the OL; 4) assigning the variables of the OL with the decompiled version of their corresponding meta variables. We claim that t and u unify if and only if t' and u' unify and that the substitution in the object language is the same as the one returned by the ML.

same or ⊇ or ⊆

Mathematically, we what to prove the following property:

```
forall (t  u : term_ol) (s : subst_ol)
       (t' u': term_ml) (s': subst_ml),
  equal_ol t u s <->
    comp t t' /\ comp u u' /\
    unif_ml t' u' s' /\ decomp s' s.
```

Math formula

In the following section we explain how we deal with term (de)compilation and unification variable linking.

4 COMPILATION

The compilation step is meant to recover the higher-order variables of the OL, expressed in a first order way, by replacing them with higher-order variables in the ML. In particular, every time a variable of the OL is encountered in the original term, it is replaced with a meta variable, and if the OL variable is applied to a list L of terms, then this list becomes the scope of the variable if L is a list of distinct names. For all the other constructors of `tm`, the same term constructor is returned and its arguments are recursively compiled. The predicate in charge for term compilation is: `type comp tm -> tm -> links -> links -> subst -> subst -> o.` Where, we take the term of the OL, produce the term of the ML, take a

```
# Common code
kind tm type.
type app list tm -> tm.
type lam (tm -> tm) -> tm.
type c string -> tm.
```

```
# OL code
type fo_uv  nat -> tm.
typeabbrev fo_subst list (option tm).
```

```
# ML code
type uv  nat -> list tm -> tm.
typeabbrev subst list (option assmt).
```

Figure 1: Language description

```
type fo_equal subst -> tm -> tm -> o.
% deref
fo_equal S (uv N) T1 :- assigned? N S T, fo_equal S T T1.
fo_equal S T1 (uv N) :- assigned? N S T, fo_equal S T1 T.
% congruence
fo_equal S (app L1) (app L2) :- forall2 (fo_equal S) L1 L2.
fo_equal S (lam F1) (lam F2) :- pi x\ fo_equal S x x => fo_equal S (F1 x) (F2 x).
fo_equal _ (c X) (c X).
fo_equal _ (uv N) (uv N).
```

Figure 2: Term equality

list of link and produce a list of new links, take a substitution and return a new substitution.

In particular, due to programming constraints, we need to drag the old subst and return a new one extended, if needed, with the new declared meta-variables.

The following code

```
kind link type.
type link nat -> nat -> nat -> subst.
```

defines a link, which is a relation between to variables indexes, the first being the index of a OL variable and the second being the index of a ML variable. The third integer is the number of term in the scope of the two variables, or equivalently, in a typed language, their arity.

integer or nat?

- how we transform an fo_tm in tm
- the role of links
- decomp - esempio che va in questa semplice rappresentazione (from intro) - esempio che non va, multi-var, eta, beta

5 UNIFICATION IN ML

- we accept HO unif with PF
- need of multiple vars for a single OL var