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 $\simeq_o$ using the ML Elpi [2], a dialect of λProlog. Elpi's equational theory comprises $\eta\beta$ equivalence and comes equipped with a higher order unification procedure $\simeq_\lambda$ restricted to the pattern fragment [8]. We want $\simeq_o$ to be as powerful as $\simeq_\lambda$ 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" $\simeq_\lambda$ 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 $\simeq_\lambda$, effectively implementing $\simeq_o$ on top of $\simeq_\lambda$ 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

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) below:

```
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 $\simeq_\lambda$ of the meta language is not aware of the equational theory of the object logic $\simeq_o$, 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 $\simeq_o$.

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 $\simeq_\lambda$ solves higher order problems in $\mathcal{L}_\lambda$.

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

```
x\ f x                  ≃λ   f
lam A x\ app[c"f", x]   ≃o   c"f"
lam A x\ app[c"f", x]   ≠λ   c"f"
P x                     ≃λ   x
app[P, x]               ≃o   x
app[P, x]               ≠λ   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 and a $\mathcal{H}_o$ one. We call $=_o$ the equality over ground terms in $\mathcal{F}_o$, $=_\lambda$ the equality over ground terms in $\mathcal{H}_o$, $\simeq_o$ the unification procedure we want to implement and $\simeq_\lambda$ the one provided by the meta language. TODO extend $=_o$ and $=_\lambda$ with reflexivity on uvars.

We write $t_1 \simeq_\lambda t_2 \mapsto \sigma$ when t_1 and t_2 unify with substitution σ, we write σt for the application of the substitution to t and we assume that the unification of our meta language is correct:

$$t_1 \simeq_\lambda t_2 \mapsto \sigma \Rightarrow \sigma t_1 =_\lambda \sigma t_2$$

We illustrate a compilation $\langle s \rangle \mapsto (t, l)$ that maps a term s in $\mathcal{F}_o$ to a term t in $\mathcal{H}_o$ and a list of links l. The links connect unification variables in $\mathcal{H}_o$ with variables in $\mathcal{F}_o$ and are used to decompile the assignment, $\langle \sigma, l \rangle^{-1} \mapsto \rho$.

Given

$$\langle s_1 \rangle \mapsto (t_1, l_1) \wedge \langle s_2 \rangle \mapsto (t_2, l_2)$$

we define

$$s_1 \simeq_o s_2 \mapsto \rho \stackrel{def}{=} t_1 \simeq_\lambda t_2 \mapsto \sigma \wedge \langle \sigma, l_1 + l_2 \rangle^{-1} \mapsto \rho$$

We write $s \in \mathcal{L}_\lambda$ if all unif variables in s are applied to distinct bound variables.

$$t_1 \simeq_o t_2 \mapsto \rho \Rightarrow \rho t_1 =_o \rho t_2 \quad (1)$$

$$s_i \in \mathcal{L}_\lambda \wedge \exists \rho, \rho s_1 =_o \rho s_2 \Leftrightarrow s_1 \simeq_o s_2 \mapsto \rho' \subseteq \rho \quad (2)$$

$$\forall \rho, \rho s_1 =_o \rho s_2 \Rightarrow \rho t_1 \simeq_o \rho t_2 \quad (3)$$

the first one is that we are correct

the second one is that we are complete and mgu iff the problems are nice

the third one is still broken since we did not define how to apply a fo subst to a ho term but it morally means that any valid substitution for s1 s2 cannot break the unification of terms t1 t2 obtained by compiling s1 and s2 before knowing the subst.

These properties allow us to simulate a unification based backward search on DTT by using $\simeq_\lambda$, in a faithful way: 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, we present in fig. 2 a new type for the terms of the OL.

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

Figure 2: Common terms

say we do not care about types

This encoding is very similar to the one introduced in section 1, except for the `all` constructor. We explicitly neglect it since blabla. Moreover, since we are working with unification variables we need to introduce a new constructor.

In the case of the OL, variables have no scope. For example, the subterm `P x` from the instance forall_dec has `P` an higher order variables with type `A → Prop` and `x` as bound variable. However, at the meta level, the translation of `P x` becomes `app[P, x]`, that is, `P` cannot reference `x`. In our encoding, we represent the variables of the OL in the following way:

```
type fo_uv nat -> tm.
```

In particular, a variable of the OL is identified by a integer refereeing a memory address. This constructor is prefixed with `fo` since the variable, having no scope, is a first-order variable. Finally, the memory of the OL is depicted by the following type abbreviation:

```
typeabbrev fo_subst list (option tm).
```

which is a list of optional terms. If a cell is `none`, then the variable corresponding to this cell is not instantiated.

On the other hand, the terms of the ML have exactly the same shape has the terms of fig. 2. However, since the ML is an Higher Order Programming Language, unification unification variables must have a scope. We represent them in the following way:

```
type ho_uv nat -> list tm -> tm.
```

Of course some attention should be payed when dealing with `ho_uv`, since we have to certify that an `uv i` remain in the pattern fragment, that is, the list of term in the scope of `i` is always a list of distinct names.

Another important implementation should be provided in order to represent the lambda abstractions and the applications of the ML. As outlined in section 2, «`x\ f x`» has not the same semantic as «`lam x\ app[f, x]`», since, even though, the two terms represent the same concept, the former is the lambda abstraction of `f` applied to the binder in ML, whereas the latter is the same term representation but wrt the OL. We already have defined the nodes `app` and `lam` for the OL, therefore, in the following code snippet, we give their representation in the ML.

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

In particular, the node `abs` stands for the lambda abstraction. The node `val` contains terms of the object language. The node for variable application is not really necessary, since, using the syntax of the ML, it is defined by putting two terms of the ML side by side. For example, the terms «`x\ f x`» becomes «`abs x\ f x`».

The memory of the ML, is defined with the following type abbreviation:

```
typeabbrev ho_subst list (option tm).
```

```
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 1: Term equality

with the invariant that the `tm` inside the cell never contains the node `fo_uv`. Note that, reciprocally, the `fo_subst` does not contain any node of the form `ho_uv`.

3.1 Term equality

OLD

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. 1. 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 $\supseteq$ or $\subseteq$

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' /\ links ... /\ 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 of distinct names L, then this list becomes the scope of the variable. 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 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?

As an example, let's study the following unification problem (a slightly modified version from section 1):

```
lam x\ app[c"decision", app[c"nfact", x, c"3"]] ≃o
  lam x\ app [c"decision", app[uv 0, x]]
```

we have the main unification problem where the nested `app` nodes have lists of different lengths making the unification to fail. The compilation of these terms produces a new unification problem with the following shape:

```
lam x\ app[c"decision", app[c"nfact", x, c"3"]] ≃λ
  lam x\ app [c"decision", uv 1 [x]]
```

The main difference is the replacement of the subterm `app[uv 0, x]` of the OL with the subterm `uv 0 [x]`. Variable indexes are chosen by the ML, that is, the index `0` for that unification variable of the OL term has not the sam meaning of the index `0` in the ML. There exists two different substitution mapping, one for the OL and one for the ML and the indexes of variable point to the respective substitution.

- 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