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, 3 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).
Instance nfact_dec n nf : Decision (nfact n nf).
Instance forall_dec A P : Finite A →
  ∀x:A, Decision (P x) → Decision (∀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 -> (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 this term encoding the three instances are represented by the following rules:

TODO: explain HOAS

```
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: explain 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 an higher order unification variable (of type tm -> 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) β-redex in the second premise (under the pi x):

```
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 $=_\lambda$ 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].

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..

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 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.
```

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

todo: explain better

```
                    # Common code
                    kind tm type.
                    type app list tm -> tm.
                    type lam (tm -> tm) -> tm.
                    type c string -> tm.
# OL code                                   # ML code
type fo_uv  nat -> tm.                      type uv  nat -> list tm -> tm.
typeabbrev fo_subst list (option 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

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 $\supseteq$ 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' /\ decomp s' s.
```

Math formula

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

4 COMPILATION: TM → TM