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 [18] which have been utilized to implement various formal systems such as First Order Logic [4], Set Theory [11] and even Higher Order Logic [10].

The object logic we are interested in is 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. Unfortunately the natural encoding of DTT in Elpi, 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 natural encoding $\mathcal{F}_o$.

We apply this technique to the implementation of a type-class [17] solver for Coq [16]. Type-class solvers are proof search procedures based on back-chaining designated lemmas, providing essential automation to widely used Coq libraries such as Stdpp/Iris [6] 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, 8 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 [18] which have been utilized to implement various formal systems such as First Order Logic [4], Set Theory [11] and even Higher Order Logic [10].

The object logic we are interested in is Dependent Type Theory (DTT), and we want to code a type-class [17] solver for Coq [16]. Type-class solvers are proof search procedures based on back-chaining designated lemmas, providing essential automation to widely used Coq libraries such as Stdpp/Iris [6] and TLC [1]. These two libraries constitute our test bed.

For example, these 3 designated lemmas (Instances in the TC slang) are supposed to be animated, backward chained, by the TC solver. fin n is the type of numbers smaller than n, nfact the number of prime factors and Finite/Decision properties designated as classes what are automatically proved by Coq using the designated instances.

```
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) → Deci
```

for example here

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

A natural encoding (others are discussed section xx) uses c for constants, all for forall, and then app and lam for the lambda calculus part.

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

unfortunately this does not work since the last rule cannt backchain on the encoded goal

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

indeed this unif problem has no solution

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

in this paper we study a more sophisticated encoding

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

This time Pm is HO for elpi, an the extra link is there to explain how to get P out of it.

```
app[c"nfact", y, c"3"] = Pm y
Pm = y\app[c"nfact", y, c"3"]
P = lam (app[c"fin",c"7"]) a\ app[c"nfact", a, c"3"]
```

We also see that this is not trivial, because in the rec call we have now a beta redex.

1.1 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 a 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 he eq theory of DTT which is much stringer than the on of 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.

2 INTRODUCTION

related [3]

Meta programming is a powerful programming style that enable the incorporation of an object language (`ol`) into a meta language (`ml`). This control over the `ol` lies at the core of meta-programming, facilitating a spectrum of operations on the object language. Some of the principal program manipulations range from interpretation to compilation.

Thanks to meta programming, it is feasible to express the equational theory of a theorem prover, as indicated in [**X**, **Y**, **Z**], with the `ml` serving as the foundational infrastructure for this ecosystem. The process of proving a theorem, that is solving a query using the jargon of logic programming, consist in verifying if a proposition is a logical consequence of a given set of assumption. The benefit of using a prologish verification style is that the proof resolution can be automatized thanks to different search strategies, such as a depth first search with backtracking. One prominent manifestation of this automated proof process is type-class resolution[17, 15]. Type classes serve as a typing structure to introduce ad-hoc polymorphism in functional languages. In `coq`, the use of type classes has increasingly become a programming style. Numerous libraries, such as *stdpp* and *iris* are built upon this machinery.

The motivating example for this article is our investigation for an alternative type-class solver for `coq` in `elpi` through the `coq-elpi` plugin. This plugin gives a deep embedding of `coq` terms into `elpi` syntax, but challenges arise whe attempting to solve type-class goals represented into their corresponding `elpi` representation.

citation to elpi

In particular, `coq` terms are encoded with the following (very simplified) data type

```
kind term type.
type const gref -> term.
type app list term -> term.
type fun (term -> term) -> term.
type prod (term -> term) -> term.
```

where a `coq` constant is a gref[1] inside the `const` constructor, the `coq` application is embedded in the `app` node, having the head and the arguments of the application expressed as a list of terms. Lambda abstractions are translated into the node `fun`, binding a term to another term. Finally forall quantification on the form `∀x,F` in the `ol` are translated into `prod x\ F` within the `ml`.

Let's take the following example from the *stdpp* library:

```
Instance forall_dec A (P: A → Prop):
  Finite A →
  ∀x, Decision (P x) →
  Decision (∀x, P x).
```

```
tc-Decision (prod x\ app[P, x]
  % premise for Finite A
  % premise for ∀x, Decision (
```

In our type-class solver, the `forall_dec` instance of `coq` (on the left) is compiled into the `elpi` rule (on the right). This compilation, derived directly from the type of `forall_dec`, conceals certain unification properties that are accepted by the unification algorithm of the `ol` but rejected by the unification algorithm of the `ml`. The primary issue of this compilation lies in how the subterm `∀x, P x` is rendered in the `ml` rule: `P` is a higher-order variable and it sees the binder `x`. However, the rule in the `ml` interprets `P` as a first-order variable, not seeing `x`. Additionally, this subterm is nested within the `app` constructor, introducing a structural challenge to the unification process.

```
tc-Decision (prod x\ const `nat)                         (1)
```

```
tc-Decision (prod x\ app[const `f, const `y, x])         (2)
```

The two examples above, already expressed in the syntax of the `ml`, depict two terms that would successfully unify in the `ol`, but encounter failure in the `ml`. Specifically, the first query fails because the `ml` is not able to unify ``const `nat`` with the term `app[P, x]` due to their distinct rigid heads.

The second goal refers to an approximation of the unification algorithm of the `ol`. Spiegare l'approssimazione. E che attraverso il nostro framework è possibile "spiegare" l'algorithm di unification usato da ogni `ol` nel ml desiderato.

maybe there's something to cite

Based on the definition provided earlier, the terms ``app[const `f, const `y, x]`` and `app[P, x]` are expected to unify with substitution $\theta := \{P \mapsto$ ``app [const `f, const `y]``$\}$. However this unification is not possible in the `ml`. This difficulty arises because, even though the two terms share the same same head (the `app` node), their corresponding lists have different lengths.

something to do here...

[1] a gref is the opaque type for `coq` identifier in `elpi`

mettere da qualche parte anche eta e beta

An immediate approach to address these unification problems would be to adopt a lazy strategy and construct highly general rules for each instance in the database. This approach would result in a rule with the following structure:

```
tc-Decision X :-
  ol.unify X (prod x\ app[P, x]),
  % premise for Finite A
  % premise for ∀x, Decision (P x)
```

In this second implementation, any query to tc-Decision will unify with the head of the rule above. The first premise ensures that the arguments unify with what we expect from the instance definition. Notably, in this case, the unification algorithm of the ol is leveraged successfully addressing the previously mentioned unification problems.

However, this approach has two main drawbacks. Firstly, the communication between the ol and the ml may potentially be slow due to the transformation of terms from one language to the other . Secondly, a lack of clauses discrimination based on the head of the clause is no longer possible. For example, any indexing algorithm designed to filter rules becomes ineffective in this scenario.

here I refer to constr->elpi and elpi->constr

Finally

3 INTRO

we are interested in using a meta language in LF style to write automation, proof search. In various works... they achieve that for a OL which is simpler than the LF, the equational theory is included in the one of the ML. This is exploited to piggy back on the unif of the ML. the peculiarity of our setting is that the OL has a richer equational theory of the ML, eg beta eta zeta bla bla. Moreover is HO logic, so quantifies over functions, so unif variable range on that too. We want to piggy back on the ML unif whenever the problem fits in its domain, eg pattern fragment. this is important for practical purposes.

3.1 in a nutshell

example, a rule for theorem

$$forallf : A -> B, ..blaf.. -> prove(\forall x, f x)blag$$

```
type app ...
prove (forall x\ app F x) :- ... bla F ...
```

on goal

```
prove (forall x\ app (app g x) x))
```

would fail since

```
F != app g x
```

.

of course one wants to avoid

```
prove P S S'' :- ol-unif P S (forall x\ app "F" x) S', ... assignment "F" S' F, bla F S' ...
```

Now, ML has HO variables

```
type lam ..
prove (forall x\ F x) :- ... bla F ...
```

this time

```
F x != app (app g x) x
```

has solution

```
F = a\ app (app g x) x
```

but F is not a term so bla needs to be adapted,

```
prove (forall x\ F x) :- ... bla (lam F) ...
bla g.
```

this is too simplistic since

```
g != lam x\ app g x
```

3.2 contribution

- prover for HO OL in ML that uses unif
- eta beta
- test on stdpp and TLC

4 INTRODUCTION

Meta programming [5] is a programming technique in which a program can treat an other program as its data. This latter program is called object language (ol for short), while the former is called meta language (ml for short). At the heart of meta programming lays the necessity of representing terms of the ol in the ml so that a wide set of program manipulations ranging from interpretation to compilation.

Meta programming has various application such as ... where thanks to meta programming it is possible to represent the logic of a language into a formal and formally verify the wanted properties. On the other hand, it is possible to embed a logic programming language into another so that some tasks can be delegated to ml.

The latter situation motivates our works, since we are implementing a type-class solver for the ol coq in the ml elpi (a variant λ-prolog). A type class [17, 15] is a typing structure allowing to introduce ad hoc polymorphism in functional languages. We call «instance» an implementation of a type class. The resolution of a type-class problem can be viewed as a logic program where type classes represent predicates parametrized by their arguments and where instances are rules for those predicates.

esempietto in cui l'HO di elpi non risolve un problem HO del linguaggio oggetto FO. Equazione XX

```
(* HO unif *)
Instance forall_dec A (P: A → Prop) `{Finite A} `{∀x, Decision (F
  Decision (∀x, P x).
Instance and_dec P `{Decision P} Q `{Decision Q}:  Decision (P /\



(* FO approx *)
Instance decide_rel A B (R: A → B → Prop) `{RelDecision R} :
  forall x y, Decision (R x y).
Instance decide_eq_nat : RelDecision (@eq nat).
Check _ : Decision (@eq nat 2 3)


Instance _ : Inj add.
        Inj F -> Inj (fun x => F (G x))

        (fun x => add x)


  Decision (∀x, P x).
```

4.1 Related work and alternative approaches

LF e PL with binders (sempre HOAS).

citare FO logic et similia fatti in LP, twelf. qui l'unification dell'OL e' facile. cercare se in twelf hanno fatto un po' di ordine superiore.

isabelle fa la stessa cosa, OL e HOL, che matcha il sistema di tipi. qui l'unificazione e' la stessa, e fanno un ecoding shallow (senza app).

Noi abbiamo un OL piu complicato, i tipi non matchano, serve un nodo app per due ragioni: meta programmazione e arieta variabile

(equazioni XX senza app e lam) XXX IDEA XXX

5 MAIN IDEA: FO ENCODING - HO ENCODING

```
kind fo_tm type.
type fo_app list fo_tm -> fo_tm.
type fo_lam (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 assmt).
type equal subst -> tm -> tm -> o.
```

```
typeabbrev subst list (option tm).
type equal subst -> tm -> tm -> o.
```

le due sintassi, XX tradotto, unif passa, bisogna riportare la soluzione nel mondo fo

this should hold in both encoding

```
forall t1 t2 s, unif t1 t2 nil s -> equal s t1 t2
```

we have unif only in the HO one, where equal is weaker (but unif is stronger)

segnatura di decomp e unif, equalfo (fa beta, eta, deref)

```
% 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
```

equal fo is congruence, deref, beta, eta (for the OL). equal ho is congruence, deref (for the OL). unif (fo or ho) should cover pattern fragment, i.e. well behaved unif.

ho, and unif ho, is the LF/ML language/unif. here we describe fo and ho with a deeply embedded syntax, but the idea is that uv can be mapped to an actual unif variable.

of course we could write a unif for the ho that had beta and eta, but if we use the one of the LF it can't have eta beta of the OL baked in.

5.1 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.

```
prove (app[c "decidable", all x\ P x]) Proof :- ho-link P P', prov
```

qui P nel paper diventa uv N [x] per un certo N.

6 RECOVERING HO

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

fallisce perche le liste non hanno la stessa lunghezza.

compile

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

```
type comp fo.tm -> ho.tm -> list link -> list link -> ho.subst -
comp (fo.c X)   (ho.c X) L L S S.
comp (fo.app [fo.uv N|Argsss]) TT L L3 S S3 :- %!,
  % TODO: here split-pf to enter dist. names into N
  % split-pf Argsss [] PF NPF,
  split-pf Argsss [] PF [], NPF = [], % TODO: compile to (uv N L
  print "In PF" PF NPF,
  if (NPF = []) (TT = ho.uv M PF1) (TT = ho.app [ho.uv M PF1 | N
  % pattern-fragment Args,
  fold4 comp PF PF1 L L1 S S1,
  fold4 comp NPF NPF1 L1 L2 S1 S2,
  ho.new S2 M S3,
  % TODO: maybe len can be given by split-pf
  len PF Len,
  L3 = [link N M Len| L2].
% TODO: if don't want to modify unif, we compile `fo.app [fo.c f
% `ho.app [ho.app[f, c0, c0], ho.c a]
comp (fo.app A) (ho.app A1) L L1 S S1 :- fold4 comp A A1 L L1 S
comp (fo.lam F) (ho.lam F1) L L1 S S1 :-
  (pi x y\ (pi A S\ comp x y A A S S) => comp (F x) (F1 y) L L1
comp (fo.uv N) (ho.uv M []) L [link N M z|L] S S1 :- ho.new S M
```

decomp ...

7 RECOVERING ETA

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

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

8 RECOVERING BETA

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

9 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 slegare le due F e poi unificare le soluzioni tra di loro

10 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,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)
```

11 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Even though type-class resolution is the motivating example of this paper, we provide a general framework allowing to solve reproduce the same unification properties of the `ol` into the `ml`. In other word, if two terms unify in the `ol`, then they still unify in the `ml`.

In the following, we consider the `ol` being able to quantify over higher-order variables and accepts $\eta - \beta$-reductions. The same unification properties are considered valid for the `ml`.

There exist two different ways to encode the `ol` in the `ml`, we can either deep embed the `ol` such that any term of the `ol` is represented with a corresponding predicate. For example, if `f` is a function of type `A -> B` in the `ol`, then the `ml` has the predicate `p` defined as `type p A' -> B' -> o`, where `A'` and `B'` are types corresponding respectively to `A` and `B` in the `ol`. In a theorem prover like `coq`, we can translate theorems like the following statement

```
forall F X, p (f X) (fun x => g x (F x)).
```
(3)

where `p, f` and `g` are defined constants of the language, into

```
p (f X) (x\ g x (F x)).
```
(4)

However, even if this encoding is quite appealing since it allows to mirror enough straightforwardly the terms of the `ol`, we loose the possibility the manipulate the terms of the `ol` into the `ml`. In other words, we have no syntax allowing to know if the current term is a constant, an application, a lambda abstraction and so on. This is mainly due to the absence of a syntax in the encoding of the `ol` terms. Moreover, another motivation for using syntax to represent terms of the `ol` is that the typing system of the `ol` could potentially be more expressive than the typing system of the `ml`[2].

[2]This is the case for `coq` wrt `elpi`, since in we have no immediate way to encode the dependent types of `coq` into `elpi`

To simplify the understanding of our encoding, in the following code snippet we give the typing schema of the `ol` terms represented into terms of the `ml`.

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

In particular, the type `tm` is the type of the terms of the `ol`. The function applications of the `ol` are represented as a list of `tm` prefixed by the constructor `app`. The `lam` constructor, represent lambda abstractions of the `ol` binding a `tm` into an other `tm`. Constants as strings inside the constructor `c`. Lastly, unification variables are integers inside the constructor `uv`, where the integer is the index of the current variable wrt a list of optional `tm`, standing for the substitution mapping of the `ol`.

This second encoding of the `ol` into our `ml` translate eq. (3) into the term:

```
app[c "p", app[c "f", uv 0], lam x\ app[c "g", x, app[uv 1, x]]].
```
(5)

This second encoding of the `ol` terms is now structured and as a drawback we are restricting the unification of the `ol`, that is, terms that originally unify at the `ol` level, do not unify in the `ml`.

For example, let a and b two defined constants and let's try to unify the `ol` term

```
p (f a) (fun x => g x b)
```
(6)

corresponding to

```
app[c "p", app[c "f", c "a"], lam x\ app[c "g", x, c "b"]].
```
(7)

with eq. (3) (corresponding to eq. (5)). The unification of the `ml` is able to instantiate `uv 0` (cf `X`) to `c "a"`, but we are no longer capable to unify the sub-term `app[uv 1, x]` (cf `F`) with `c "b"`.

The result of this translation of terms inside the `ml` causes a certain lack of powerfulness while symbolizing higher-order variables. Recall that we are considering a `ml` capable to deal with higher-order variables, however, the sub-term `app[uv 1, x]` is not expressed into the canonical form where a higher-order variable of the `ml` is in the pattern fragment [7], i.e. a variable applied to distinct names. Therefore, we need to preprocess the received unification problem $t_1 = t_2$ by

12 TERM COMPILATION

In order to present the the compilation of the `ol` terms, so that higher order unification can be performed, we need a second and more powerful representation of the `ol` terms so that variables have a scope. This specification is shown in the code snippet below.

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

In particular a `ml.uv` term is meant as a unification variable of the meta-language. Therefore, the unification between

```
ml.lam x\ ml.lam y\ ml.uv 1 [x, y]
```

and

```
ml.lam x\ ml.lam y\ ml.uv 0 [x]
```

is supposed to procedure of the substitution `ml.lam x\ ml.lam y\ ml.uv 2 [x]` for `uv 0` and the substitution `ml.lam x\ ml.uv 2 [x]`) for `uv 1`.

Moreover, if `ml.uv` stands for meta-variables, the `app` and the `lam` constructors are the nodes for the terms of the `ol`. Therefore, we cannot claim that `ml.lam x\ ml.app [ml.c "f", x]` and `ml.c "f"` unify, since, even though the first is the η-expansion of the second, the `ml` does not know how to $\eta\beta$-reduce terms of the `ol`.

In our encoding, we explicitly encode the meta-variables with the `ml.uv` constructor. This is because we prefer to have the full control of the `ml`, including the meta-variables instantiation. This way we are able to concretely touch the substitution performed by the `ml`. In a further section, we show that there is no difference between our custom `ml` language and any other `ml`. Of course, a full control on the unification behind meta-variable assmt ask to drag the substitution mapping of the `ml` and update it each time a variable is refined.

The compilation phase is quite straightforward, each constructor of type `tm` is mapped to its corresponding version of type `ml.tm`. A slight different approach is taken in the case of terms of the form `app [uv N | L]`, where the term is translated into `tm.un M L`, that is, a new meta-variable `M` with scope `L`.

This latter term transformation is untying the original variable `N` of the `ol` from the compiled term in the `ml`. This means that when `M` is instantiated into the `ml`, we need to transfer the substitution to the `ol`. In order to bridge instantiation of meta-variables with the `ol` variables, an ad hoc `link` is crafted between the two variables.

A link, `type link nat -> nat -> link`, takes two integers: the first stands for the index of variables in the `ol` and the second is the index of the meta-variables.

For example, if we take back the example in eq. (7), and want to compile it, we obtain the new term:

```
app[c "p", app[c "f", c "a"], lam x\ app[c "g", x, c "b"]].
```
(8)

—————————— END ——————————

12.1 First-order unification

Just as an introduction, we briefly show some small example of unification between terms with only first-order unification variables. This way, we would like the reader to become familiar between the communication of the two languages.

Let's take as an example the following unification problem in the obj. lang.:

$$f\ x\ 1 \stackrel{\tau}{=} f\ Y\ Z \tag{9}$$

where f, x and 1 are defined constant and Y and Z are both unification variables. By convention we use upper case letter for quantified variables. Moreover, for this first representation we do not really focus on the type of the manipulated objects, since they do not condition the unification algorithm.

It is quite evident that a valid substitution for eq. (9) is $\theta = \{Y \mapsto `x', Z \mapsto `1'\}$. Now let's consider the same problem translated in the meta language.

$$app[`f', `x', `1'] \stackrel{\tau}{=} app[`f', Y, Z] \tag{10}$$

The unification of these terms is again quite simple since it is sufficient to do a simple matching sub-term by sub-term so that variables can be instantiated. We can therefore note that the same substitution θ will be produced.

12.2 Higher-order unification

The unification problem treated before was enough easy to be correctly understood by both language representation. We want now to go a bit further and reason with a more complex problem where a variable is a function of higher-order.

We propose two different higher-order unification problem in the following equations where, in the former we have rigid-flexible unification and in the latter we have a flexible-flexible unification.

$$f\ x\ 1 \stackrel{\tau}{=} F\ x \tag{11}$$

$$G\ x\ y \stackrel{\tau}{=} H\ y\ x \tag{12}$$

The two substitutions for the previous examples are $\theta_1 = \{F \mapsto fun\ x \Rightarrow f\ x\ 1\}$ and $\theta_2 = \{H \mapsto fun\ y\ x \Rightarrow G\ x\ y\}$. We can note that to be in the pattern fragment, a functional variable should be applied to distinct names.

If we translate the problem before in the meta language, the unification problems showed above become

$$app[`f',\ `x',\ `1'] \stackrel{\tau}{=} app[F,\ `x'] \tag{13}$$

$$app[G,\ `x',\ `y'] \stackrel{\tau}{=} app[H,\ `y',\ `x'] \tag{14}$$

Now, the new unification problems are no more expressed in the logic of the meta language and, therefore, in both cases, unification fails. The procedure we can adopt in order to transform a higher-order unification problem of the object language into the logic of the meta language is to transform the entry of the problem in a problem which can be understood by the meta language. The procedure is made of two steps:

(1) In the first place, we need to recognize the structure of the pattern fragment expressed in the term received in entry. This means that we need to find all the sub-terms of the form 'app[X | L]', where 'X' is a flexible variable and 'L' is a list of distinct names.
(2) For any sub-term representing a higher-order unification in the object language, we build a fresh variable 'X'' such that the names 'L' are not in the scope of 'X'', we call 'X'' the twin variable of 'X'.
(3) We solve the new goal where each pattern fragment problem is replaced with a problem using twin variables and after each of these problems, we add a new premise linking these twin variables. The linking is done using the following criteria: for

each abstraction in the resulting term 'X'', unify recursively 'X' to a lambda abstraction in the object language.

The previous algorithm can be applied to eqs. (11) and (12) to provide the wanting solution. In particular, eq. (11) is transformed into the unification problem:

$$f\ x\ 1 \stackrel{\tau}{=} F'\ x,\ ho-link\ F'\ F \tag{15}$$

$$G'\ x\ y \stackrel{\tau}{=} H'\ y\ x, ho-link\ G'\ G, ho-link\ H'\ H \tag{16}$$

For instance, the former unification problem produce the substitution $\theta_1 = \{F' \mapsto (x \backslash\ f\ x\ 1)\}$. The ho-link function is then applied to transform the substitution of F' into the corresponding term of the object langue: $F \mapsto fun\_\ \_\ (x \backslash\ f\ x\ 1)$ which correspond to the term $fun\ x \Rightarrow f\ x\ 1$. The latter unification problem gives the substitution $\theta_2 = \{H' \mapsto (y\ x \backslash G'\ x\ y)\}$ in the meta language. The first ho-link simply unify G to G' since G' is flexible, whereas H is mapped to $fun\ y\ x \Rightarrow G'\ x\ y$.

The role of the ho-link is not only to instantiate the higher-order variable F of the object language when F is flexible and the twin variable in the meta language is rigid. It may happen that F has already been partially instantiated. The unification problem below gives such an example in the object language:

$$G\ x\ y \stackrel{\tau}{=} H\ y\ x, H\ x\ y \stackrel{\tau}{=} x \tag{17}$$

producing the following substitution $\theta = \{G \mapsto (fun\ x\ y \Rightarrow x); F \mapsto (fun\ x\ y \Rightarrow y)\}$. This unification problem is translated into:

$$\begin{aligned} &G'\ x\ y \stackrel{\tau}{=} H'\ y\ x,\ ho-link\ G'\ G,\ ho-link\ H'\ H \\ &H''\ x\ y \stackrel{\tau}{=} x,\ ho-link\ H''\ H \end{aligned} \tag{18}$$

The first line produces the same substitution as before: $\theta = \{H' \mapsto (x\ y \backslash G'\ y\ x)\}$, where G' is unified with G and H is instantiated to «$fun\ x\ y \Rightarrow G'\ y\ x$». While executing the second line, we see H'' instantiated to the function $(x\ y \backslash x)$ in the meta language. The last ho-link is charged to link H'' with H but this time H has already been partially instantiated. In particular the call to this ho-link is as follows: $ho-link\ (x\ y \backslash x)\ (prod\ \_\ \_\ (x \backslash\ prod\ \_\ \_\ (y \backslash\ app[G', x, y])))$.

Since the two arguments have rigid heads, we start to traverse both terms recursively by eating each lambda-abstraction. At the end of this procedure, the remaining sub-terms are now x and $app[G', x, y]$,

in our code this example though the eta mess error

13 HO UNIFICATION IN TYPED LANGUAGES

TODO: ho-link need the type of original term to produce a typed term in the object language, example: f x 1 = F x => type of F = (A -> Prof) if type of f is (A -> nat -> Prop)

14 PROOF AUTOMATION FROM COQ TO ELPI

TODO: representing a logic programming language into an other: compile rules keeping higher order unification

Term	ol	ml
Constant	a	'a'
Application	f a_1 a_2 ... a_n	app['f', 'a_1', 'a_2', ..., 'a_n']
Abstraction	fun (x : T) ⇒ f x	fun 'x' T (x\app['f', 'x'])
Variable	X	X

Table 1: ol terms to ml terms representation

14.1 Dealing with FO *non-syntactical* unification

14.2 Dealing with HO unification

A TC IN COQ

For instance, if XXX is the type class representing the AAA, then ZZZ and WWW are instances for XXX. In the code snippet below, we give such implementation in coq.

```
Inductive sig (A : Type) (P : A -> Prop) : Type := ...

Class Decision (P : Prop) := decide : {P} + {not P}.
Class RelDecision {A B: Type} (R : A -> B -> Prop).
Class ProofIrrel (A : Type) : Prop := proof_irrel (x y : A) : x

Instance decide_rel: forall (A B : Type) (R : A -> B -> Prop),
  RelDecision R -> forall (x : A) (y : B), Decision (R x y). Adm
Instance True_pi : ProofIrrel True. Admitted.
Instance sig_eq_dec: forall (A : Type) (P : A -> Prop),
  (forall x, ProofIrrel (P x)) -> RelDecision (@eq A) ->
  RelDecision (@eq (sig A P)). Admitted.
```

This small set of instances after a first phase of compilation is translated into the following elpi rules:

explain compilation of pred and inst?

```
type tc-Decision term -> term -> o.
type tc-RelDecision term -> term -> term -> term -> o.
type tc-ProofIrrel term -> term -> o.

tc-ProofIrrel (`True`) (`True_pi`).
tc-Decision (app [R, X, Y])
  (app [`decide_rel`, A, B, R, P, X, Y]) :-
  tc-RelDecision A B R P.
tc-RelDecision (app [`sig`, A, P])
  (app [`sig`, A, P])
  (app [`eq`, app [`sig`, A, P]])
  (app [`sig_eq_dec`, A, P, P1, P2]) :-
  pi-decl c0 `x` A =>
    tc-ProofIrrel (app [P, c0]) (app [P1, c0]),
  tc-RelDecision A A (app [`eq`, A]) P2.
```

In this paper we do not really want to explain how the translation of the class/instances is performed in our ml, we prefer to focus our attention on unification of terms of the ol in our ml. Although, in table 1, we provide a simple subset of the typing system used to represent the term of the ol in the ml.

Type-class resolution starts from a query, that is a class applied to some arguments. This coq term is translated into a term of the ml and the search for a solution in the database is started. However, it may happen that the term representation in the ml may hide some unification properties that are true in the ol. In the example above, the goal `Decision (@eq T a b)` for some `a` and `b` unifies with `Decision (R x y)` in the `ol` but not in its meta representation. Similarly, the goal `RelDecision (@eq (sig T ?P))` where `?P`, under the hypothesis `RelDecision (@eq nat)`, will trying to apply the rule for `sig_eq_dec`, we fall into an higher order unification problem, where `P` is applied to the local name `x`. However, the corresponding rule in the `ml` exploit a first order variable `P`. Therefore, after the refinement of the goal to `sig_eq_dec`, the resolution immediately fail to solve the premise `tc-ProofIrrel (app [P, c0]) (app [P1, c0])`.

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%5C_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] Patricia Hill and J Gallagher. “Meta-Programming in Logic Programming”. In: (Sept. 1994). DOI: 10.1093/oso/9780198537922.003.0010.

[6] 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.

[7] Dale Miller. “A logic programming language with lambda-abstraction, function variables, and simple unification”. In: *Extensions of Logic Programming*. Ed. by Peter Schroeder-Heister. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 253–281. ISBN: 978-3-540-46879-0.

[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 Wenzel. *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 Benzmüller. “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] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-71067-7.

[16] The Coq Development Team. *The Coq Reference Manual – Release 8.18.0*. https://coq.inria.fr/doc/V8.18.0/refman. 2023.

[17] 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.

[18] 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.