HO unification from object language to meta language

Enrico Tassi, Davide Fissore

Month XX, 202Y

Abstract

Unification is a fundamental process behind goal resolution in logic programming. Two terms t_1 and t_2 unify up to $\eta\beta$-reduction if it exist a substitution θ such that $\theta t_1 = \theta t_2$. A substitution is a mapping from unification variables to terms and it is the most general, if forall substitution θ', such that $\theta' t_1 = \theta' t_2$, it exists a substitution θ'' such that $\theta' = \theta\theta''$. Unification has been widely studied in the 90s [here the refs]. In this paper we want to give a framework to reason about unification in meta-programming.

1 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.

1.1 in a nutshell

example, a rule for theorem

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

```
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
```

1.2 contribution

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

2 Introduction

Meta programming [3] 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. Furthermore, since a logic program is a set of clauses, the behavior of the `ol` can be modified `in corsa` by simply adding new rules inside the database of the meta program.

translate in english

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 [1, 4] 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 (P x)} :
  Decision (∀x, P x).
Instance and_dec P `{Decision P} Q `{Decision Q}:  Decision (P /\ Q).
```

```
(* 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).
```

2.1 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

3 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.
```

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

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

proprieta desiderate, serve equalho (fa deref)

3.1 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', prove (... P'
↪   ...).
```

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

4 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 -> ho.subst ->
↪   o.
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) + link-fo-app
  print "In PF" PF NPF,
  if (NPF = []) (TT = ho.uv M PF1) (TT = ho.app [ho.uv M PF1 | NPF1]),
  % 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, c0, c0, ho.c
↪   a]` into
% `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 S1.
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 S S1).
comp (fo.uv N) (ho.uv M []) L [link N M z|L] S S1 :- ho.new S M S1.

decmop...
```

5 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 ..

6 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.

7 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

8 recoving 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)
```

9 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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)).        (1)
```

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

```
p (f X) (x\ g x (F x)).        (2)
```

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`[1].

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 Equation (1) into the term:

`app[c "p", app[c "f", uv 0], lam x\ app[c "g", x, app[uv 1, x]]].` (3)

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)` (4)

corresponding to

`app[c "p", app[c "f", c "a"], lam x\ app[c "g", x, c "b"]].` (5)

with Equation (1) (corresponding to Equation (3)). 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 [2], i.e. a variable applied to distinct names. Therefore, we need to preprocess the received unification problem $t_1 = t_2$ by (i) compiling the terms into a terms t'_1 and t'_2 understandable by the `ml` (ii) finding a valid substitution for t'_1 and t'_2 (iii) giving back a valid substitution θ for the `ol`, such that θ is the most general unifier for T_1 and t_2 in the logic of the `ol`.

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

10 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 assignment 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 Equation (5), 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"]].        (6)
```

——— END ———

10.1 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.:

$$f\ x\ 1 \overset{\tau}{=} f\ Y\ Z \tag{7}$$

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 Equation (7) is $\theta = \{Y \mapsto `x', Z \mapsto `1'\}$. Now let's consider the same problem translated in the meta language.

$$app[`f', `x', `1'] \overset{\tau}{=} app[`f', Y, Z] \tag{8}$$

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.

10.2 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.

$$f\ x\ 1 \overset{\tau}{=} F\ x \tag{9}$$

$$G\ x\ y \overset{\tau}{=} H\ y\ x \tag{10}$$

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

$$app[`f',\ `x',\ `1'] \overset{\tau}{=} app[F,\ `x'] \tag{11}$$

$$app[G,\ `x',\ `y'] \overset{\tau}{=} app[H,\ `y',\ `x'] \tag{12}$$

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 Equations (9) and (10) to provide the wanting solution. In particular, Equation (9) is transformed into the unification problem:

$$f\ x\ 1 \overset{\mathcal{T}}{=} F'\ x,\ ho-link\ F'\ F \tag{13}$$

$$G'\ x\ y \overset{\mathcal{T}}{=} H'\ y\ x, ho-link\ G'\ G, ho-link\ H'\ H \tag{14}$$

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:

$$G\ x\ y \overset{\mathcal{T}}{=} H\ y\ x, H\ x\ y \overset{\mathcal{T}}{=} x \tag{15}$$

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:

$$\begin{aligned} G'\ x\ y \overset{\mathcal{T}}{=} H'\ y\ x,\ ho-link\ G'\ G,\ ho-link\ H'\ H \\ H''\ x\ y \overset{\mathcal{T}}{=} x,\ ho-link\ H''\ H \end{aligned} \tag{16}$$

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

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

11 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)

12 Proof automation from `coq` to `elpi`

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

12.1 Dealing with FO *non-syntactical* unification

12.2 Dealing with HO unification

References

[1] 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.

[2] 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.

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

[4] 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.

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 = y.

Instance decide_rel: forall (A B : Type) (R : A -> B -> Prop),
  RelDecision R -> forall (x : A) (y : B), Decision (R x y). Admitted.
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])`.