

# 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 [13], Twelf [14],  $\lambda$ Prolog [10] and Isabelle [20] which have been utilized to implement various formal systems such as First Order Logic [5], Set Theory [12], Higher Order Logic [11], and even the Calculus of Constuctions [4].

The object logic we are interested in is Coq's [18] Dependent Type Theory (DTT), for which we aim to implement a unification procedure  $\approx_o$  using the ML Elpi [3], a dialect of  $\lambda$ Prolog. Elpi's equational theory comprises  $\eta\beta$  equivalence and comes equipped with a higher order unification procedure  $\approx_\lambda$  restricted to the pattern fragment [9]. We want  $\approx_o$  to be as powerful as  $\approx_\lambda$  but on the object logic DTT. Elpi also comes with an encoding for DTT that works well for meta-programming [17, 16, 7, 6]. Unfortunately this encoding, which we refer to as  $\mathcal{F}_o$ , “underuses”  $\approx_\lambda$  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  $\approx_\lambda$ , effectively implementing  $\approx_o$  on top of  $\approx_\lambda$  for the encoding  $\mathcal{F}_o$ .

We apply this technique to the implementation of a type-class [19] solver for Coq [18]. 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 [8] 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, 13 pages. <https://doi.org/ZZZZZZZZZZZZ>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).  
Conference'17, July 2017, Washington, DC, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
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 [13], Twelf [14],  $\lambda$ Prolog [10] and Isabelle [20] which have been utilized to implement various formal systems such as First Order Logic [5], Set Theory [12], Higher Order Logic [11], and even the Calculus of Constuctions [4].

The object logic we are interested in is Coq's [18] Dependent Type Theory (DTT), and we want to code a type-class [19] solver for Coq [18] using the Coq-Elpi [17] 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 [8] 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 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 con  string -> tm.               % constants
```

Following standard  $\lambda$ Prolog [10] 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 `<∀y:t, nfact y 3>`:

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

Unfortunately this translation of rule (r3) uses the predicate `P` 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) given below

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

we obtain an unsolvable unification problem (p): the two lists of terms have different lengths!

```
app[con"nfact", y, con"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 P A, finite A, (r3a)
  pi x\ decision (app[P, x]).
```

Since `Pm` is an higher-order unification variable of type `tm -> tm`, with `x` in its scope, the unification problem (p') admits one solution:

```
app[con"nfact", y, con"3"] = Pm y (p')
Pm = x\ app[con"nfact", x, con"3"] % assignment for Pm
A = app[con"fin", con"7"] % assignment for A
```

After unifying the head of rule (r3a) with the goal, Elpi runs the premise `link Pm P A` that is in charge of bringing the assignment for `Pm` back to the domain `tm` of Coq terms:

```
P = lam A a\ app[con"nfact", a, con"3"]
```

This simple example is sufficient to show that the encoding we seek is not trivial and does not only concern the head of rules, but the entire sequence of unification problems that constitute the execution of a logic program. In fact the solution for `P` above generates a (Coq)  $\beta$ -redex in the second premise (the predicate under the `pi w\`):

```
decision (app[lam A (a\ app[con"nfact", a, con"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[con"nfact", a, con"3"]), x] =
app[con"nfact", N, NF]
```

The root cause of the problems we sketched in the running example is that the unification procedure  $\approx_\lambda$  of the meta language is not aware of the equational theory of the object logic, even if both theories include  $\eta\beta$ -conversion and admit most general unifiers for unification problems in the pattern fragment  $\mathcal{L}_\lambda$  [9].

*Contributions.* In this paper we discuss alternative encodings of Coq in Elpi (Section 2), then we identify a minimal language  $\mathcal{F}_0$  in which the problems sketched here can be fully described. We then detail an encoding `comp` from  $\mathcal{F}_0$  to  $\mathcal{H}_0$  (the language of the

meta language) and a decoding `decomp` to relate the unifiers bla bla.. TODO citare Teyjus.

## 2 PROBLEM STATEMENT

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 proposition unfolding and fix-point 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  $\mathcal{L}_\lambda$  [9]. We call this unification procedure  $\approx_o$ .

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

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

$x \backslash f \ x$	$\approx_\lambda$	$f$
$\text{lam } A \ x \backslash \text{app}[\text{con}"f", x]$	$\approx_o$	$\text{con}"f"$
$\text{lam } A \ x \backslash \text{app}[\text{con}"f", x]$	$\neq_\lambda$	$\text{con}"f"$
$P \ x$	$\approx_\lambda$	$x$
$\text{app}[P, x]$	$\approx_o$	$x$
$\text{app}[P, x]$	$\neq_\lambda$	$x$

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

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

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

To state precisely the problem we solve we need a  $\mathcal{F}_0$  representation of DTT terms and a  $\mathcal{H}_0$  one. We call  $=_o$  the equality over ground terms in  $\mathcal{F}_0$ ,  $=_\lambda$  the equality over ground terms in  $\mathcal{H}_0$ ,  $\approx_o$  the unification procedure we want to implement and  $\approx_\lambda$  the one provided by the meta language. TODO extend  $=_o$  and  $=_\lambda$  with reflexivity on uvars.

We write  $t_1 \approx_\lambda t_2 \mapsto \sigma$  when  $t_1$  and  $t_2$  unify with substitution  $\sigma$ , we write  $\sigma t$  for the application of the substitution to  $t$ ,  $\sigma \subseteq \sigma'$  when  $\sigma$  is more general than  $\sigma'$ , and we assume that the unification of our meta language is correct:

$$t_i \in \mathcal{L}_\lambda \Rightarrow t_1 \approx_\lambda t_2 \mapsto \rho \Rightarrow \rho t_1 =_\lambda \rho t_2 \quad (1)$$

$$t_i \in \mathcal{L}_\lambda \Rightarrow \rho t_1 =_\lambda \rho t_2 \Rightarrow \exists \rho', t_1 \approx_\lambda t_2 \mapsto \rho' \wedge \rho' \subseteq \rho \quad (2)$$

We illustrate a compilation  $\langle s \rangle \mapsto (t, m, l)$  that maps a term  $s$  in  $\mathcal{F}_0$  to a term  $t$  in  $\mathcal{H}_0$ , a variable mapping  $m$  and list of links  $l$ . The variable map connects unification variables in  $\mathcal{H}_0$  with variables in  $\mathcal{F}_0$  and is used to “decompile” the assignment,  $\langle \sigma, m, l \rangle^{-1} \mapsto \rho$ . Links represent problematic sub-terms which are linked to the unification variable that stands in their place in the compiled term. These links are checked for or progress XXX improve....

We represent a logic program *run* in  $\mathcal{F}_0$  as a list *steps*  $p$  of length  $N$ . Each made of a unification problem between terms  $S_{p_l}$  and  $S_{p_r}$  taken from the set of all terms  $\mathcal{S}$ . The composition of these steps starting from the empty substitution  $\rho_0$  produces the final substitution  $\rho_N$ .<sup>1</sup> The initial here  $\rho_0$  is the empty substitution

$$\begin{aligned} \text{fstep}(\mathcal{S}, p, \rho) &\mapsto \rho'' \stackrel{\text{def}}{=} \rho S_{p_l} \approx_o \rho S_{p_r} \mapsto \rho' \wedge \rho'' = \rho \cup \rho' \\ \text{frun}(\mathcal{S}, N) &\mapsto \rho_N \stackrel{\text{def}}{=} \bigwedge_{p=1}^N \text{fstep}(\mathcal{S}, p, \rho_{p-1}) \mapsto \rho_p \end{aligned}$$

We simulate each run in  $\mathcal{F}_0$  with a run in  $\mathcal{H}_0$  as follows. Note that  $\sigma_0$  is the empty substitution.

$$\begin{aligned} \text{hstep}(\mathcal{T}, p, \sigma, \mathbb{L}) &\mapsto \sigma'' \stackrel{\text{def}}{=} \\ &\sigma \mathcal{T}_{p_l} \approx_\lambda \sigma \mathcal{T}_{p_r} \mapsto \sigma' \wedge \text{check}(\mathbb{L}, \sigma \cup \sigma') \mapsto \sigma'' \\ \text{hrun}(\mathcal{S}, N) &\mapsto \rho_N \stackrel{\text{def}}{=} \\ &\mathcal{T} \times \mathbb{M} \times \mathbb{L} = \{(t_j, m_j, l_j) | s_j \in \mathcal{S}, \langle s_j \rangle \mapsto (t_j, m_j, l_j)\} \\ &\bigwedge_{p=1}^N \text{hstep}(\mathcal{T}, p, \sigma_{p-1}, \mathbb{L}) \mapsto \sigma_p \\ &\langle \sigma_N, \mathbb{M}, \mathbb{L} \rangle^{-1} \mapsto \rho_N \end{aligned}$$

Here *hstep* is made of two sub-steps: a call to  $\approx_\lambda$  (on the compiled terms) and a call to check on the set of links. We claim the following:

PROPOSITION 2.1 (SIMULATION).  $\forall \mathcal{S}, \forall N$

$$\text{frun}(\mathcal{S}, N) \mapsto \rho_N \Leftrightarrow \text{hrun}(\mathcal{S}, N) \mapsto \rho_N$$

That is, the two executions give the same result. Moreover:

PROPOSITION 2.2 (SIMULATION FIDELITY). *In the context of* *hrun*, *we have that*  $\forall p \in 1 \dots N$

$$\text{fstep}(\mathcal{S}, p, \rho_{p-1}) \mapsto \rho_p \Leftrightarrow \text{hstep}(\mathcal{T}, p, \sigma_{p-1}, \mathbb{L}) \mapsto \sigma_p$$

In particular this property guarantees that a *failure* in the  $\mathcal{F}_0$  run is matched by a failure in  $\mathcal{H}_0$  at the same step. We consider this property very important from a practical point of view since it guarantees that the execution traces are strongly related and in turn this enables a user to debug a logic program in  $\mathcal{F}_0$  by looking at its execution trace in  $\mathcal{H}_0$ .

We can define  $s_1 \approx_o s_2$  by specializing the code of *hrun* to  $\mathcal{S} = \{s_1, s_2\}$  as follows:

$$\begin{aligned} s_1 \approx_o s_2 &\mapsto \rho \stackrel{\text{def}}{=} \\ &\langle s_1 \rangle \mapsto (t_1, m_1, l_1) \wedge \langle s_2 \rangle \mapsto (t_2, m_2, l_2) \\ &t_1 \approx_\lambda t_2 \mapsto \sigma' \wedge \text{check}(\{l_1, l_2\}, \sigma') \mapsto \sigma'' \wedge \\ &\langle \sigma'', \{m_1, m_2\}, \{l_1, l_2\} \rangle^{-1} \mapsto \rho \end{aligned}$$

PROPOSITION 2.3 (PROPERTIES OF  $\approx_o$ ).

$$s_i \in \mathcal{L}_\lambda \Rightarrow s_1 \approx_o s_2 \mapsto \rho \Rightarrow \rho s_1 =_o \rho s_2 \quad (3)$$

$$s_i \in \mathcal{L}_\lambda \Rightarrow \rho s_1 =_o \rho s_2 \Rightarrow \exists \rho', s_1 \approx_o s_2 \mapsto \rho' \wedge \rho' \subseteq \rho \quad (4)$$

$$\rho s_1 =_o \rho s_2 \Rightarrow \rho' \subseteq \rho \Rightarrow \rho' s_i \in \mathcal{L}_\lambda \Rightarrow \rho' s_1 \approx_o \rho' s_2 \quad (5)$$

Properties 3 and 4 state, respectively, that in  $\mathcal{L}_\lambda$  the implementation of  $\approx_o$  is correct, complete and returns the most general unifier.

Property 5 states that  $\approx_o$ , hence our compilation scheme, is resilient to unification problems outside  $\mathcal{L}_\lambda$  solved by a third party. We believe this property is of practical interest since we want the user to be able to add heuristics via hand written rules to the ones

<sup>1</sup>If the same rule is used multiple time in a run we just consider as many copies as needed of the terms composing the rules, with fresh unification variables each time

obtained by our compilation scheme. A Typical example is the following problem ( $q$ ) that is outside  $\mathcal{L}_\lambda$ :

$$\begin{aligned} \text{app} [F, \text{con} "a"] &= \text{app} [\text{con} "f", \text{con} "a", \text{con} "a"] \quad (q) \\ F &= \text{lam } x \backslash \text{app} [\text{con} "f", x, x] \quad (h) \end{aligned}$$

Instead of rejecting it our scheme accepts it and guarantees that if ( $h$ ) is given (after the compilation part of the scheme, as a run time hint) then ...

## 2.1 The intuition in a nutshell

A term  $s$  is compiled in a term  $t$  where any “problematic” subterm  $tp$  is replaced by a unification variable  $h$  and an accessory link, that represent a suspended unification problem  $h \approx_\lambda tp$ . As a result  $\approx_\lambda$  is well behaved on  $t$ , that is it captures  $=_o$ . We now define “problematic” formally.

Definition 2.4 ( $\diamond \eta$ ).  $\diamond \eta = \{t | \exists \rho, \rho t \text{ is an eta expansion}\}$

An example of term  $t$  in  $\diamond \eta$  is  $\lambda x. \lambda y. F y x$  since the substitution  $\rho = \{F \mapsto \lambda a. \lambda b. fba\}$  makes  $\rho t = \lambda x. \lambda y. fxy$  that is the eta long form of  $f$ .

Definition 2.5 ( $\diamond \beta$ ).  $\diamond \beta = \{X t_1 \dots t_n | t \notin \mathcal{L}_\lambda\}$ .

An example of  $t$  in  $\diamond \beta$  is  $Fa$  for a constant  $a$ . Note however that an oracle could provide an assignment  $\rho = \{F \mapsto \lambda x. x\}$  that makes the resulting term fall outside of  $\diamond \beta$ .

Definition 2.6 (Subterm  $\mathcal{P}(t)$ ).

$$\begin{aligned} t &\in \mathcal{P}(t) \\ t &= f t_1 \dots t_n \Rightarrow \mathcal{P}(t_i) \subseteq \mathcal{P}(t) \wedge f \in \mathcal{P}(t) \\ t &= \lambda x. t' \Rightarrow \mathcal{P}(t') \subseteq \mathcal{P}(t) \end{aligned}$$

We write  $\mathcal{P}(X) = \bigcup_{t \in X} \mathcal{P}(t)$ .

Definition 2.7 (Normal form). Given a set of terms  $X \subseteq \mathcal{H}_0$ ,

$$\text{normal}(X) = \mathcal{P}(X) \cap (\diamond \beta \cup \diamond \eta) = \emptyset$$

We write  $\sigma X = \{\sigma t | t \in X\}$ .

PROPOSITION 2.8 (NORMAL FORM PRESERVATION).  $\forall \mathcal{T}, \forall \mathbb{L}, \forall p, \forall \sigma, \forall \sigma'$

$$\text{normal}(\sigma \mathcal{T}) \wedge \text{hstep}(\mathcal{T}, p, \sigma, \mathbb{L}) \mapsto \sigma' \Rightarrow \text{normal}((\sigma \cup \sigma') \mathcal{T})$$

In particular this guarantees that is we start from normal terms we never introduce eta-long or non-beta-normal terms in  $\sigma'$ .

Note that proposition 2.8 does not hold for  $\approx_o$  since decompilation can introduce (actually restore) terms in  $\diamond \eta$  or  $\diamond \beta$ .

## 3 ALTERNATIVE ENCODINGS AND RELATED WORK

Paper [2] introduces semi-shallow.

Our encoding of DTT may look “semi shallow” since we use the meta-language lambda abstraction but not its application (for the terms of type  $\text{tm}$ ). A fully shallow encoding unfortunately does not fit our use case, although it would make the running example work:

```
finite (fin N).
decision (nfact N NF).
decision (all A x \ P x) :- finite A, pi x \ decision (P x).
```

There are two reasons for dismissing this encoding. The first one is that in DTT it is not always possible to adopt it since the type system of the meta language is too weak to accommodate terms with a variable arity, like the following example:

```
Fixpoint arr T n := if n is S m then T -> arr T m else T.
Definition sum n : arr nat n := ...
Check sum 2 7 8 : nat.
Check sum 3 7 8 9 : nat.
```

The second reason is the encoding for Coq is used for meta programming the system, hence it must accommodate the manipulation of terms that are now known in advance (not even defined in Coq) without using introspection primitives such as Prologs's functor and arg.

In the literature we could find a few related encoding of DTT. TODO In [4] 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.

TODO This other paper [15] should also be cited.

None of the encodings above provide a solution to our problem.

#### 4 PRELIMINARIES: $\mathcal{F}_0$ AND $\mathcal{H}_0$

In order to reason about unification we provide a description of the  $\mathcal{F}_0$  and  $\mathcal{H}_0$  languages where unification variables are first class terms, i.e. they have a concrete syntax. We keep these languages minimal, for example, we omit the all quantifier of DTT we used in the example in Section 1 together with the type notation of terms carried by the lam constructor.

```
kind fm type.          kind tm type.
type fapp list fm -> fm. type app list tm -> tm.
type flam (fm -> fm) -> fm. type lam (tm -> tm) -> tm.
type fcon string -> fm.   type con string -> tm.
type fuva nat -> fm.      type uva nat -> list tm -> tm.
```

Figure 1:  $\mathcal{F}_0$  and  $\mathcal{H}_0$  language

In the case of  $\mathcal{F}_0$  unification variables fuva have no explicit scope: the arguments of an higher order unification variable are via the fapp constructor. For example in the statement of the instance forall\_dec the term  $P \ x$  is represented as fapp[fuva  $N$ ,  $x$ ], where  $N$  is a memory address and  $x$  is a bound variable.

In  $\mathcal{H}_0$  the representation of  $P \ x$  is instead uva  $N \ [x]$ . We say that the unification variable uva  $N \ L$  is in  $\mathcal{L}_\lambda$  iff distinct  $L$  holds.

```
type distinct list A -> o.
distinct [].
distinct [X|XS] :- name X, not(mem X XS), distinct XS.
```

The name builtin predicate tests if a term is a bound variable.<sup>2</sup> The compiler ?? needs to support terms outside  $\mathcal{L}_\lambda$  for practical reasons, so we don't assume all out terms are in  $\mathcal{L}_\lambda$  but rather test. **what??**

In both languages unification variables are identified by a natural number, which can be seen as a memory address. The memory and its associated operations are described below:

<sup>2</sup>one could always load name  $x$  for every  $x$  under a pi and get rid of the name builtin

```
typeabbrev memory A (list (option A)).
type set? nat -> memory A -> A -> o.
type unset? nat -> memory A -> o.
type assign nat -> memory A -> A -> memory A -> o.
```

If a memory cell is none, then the corresponding unification variable is not set. assign sets an unset cell to the given value.

Since in  $\mathcal{H}_0$  unification variables have a scope, their solution needs to be abstracted over it in order to enable the instantiation of a single solution to different scopes. On the contrary solutions to  $\mathcal{F}_0$  variables are plain terms.

```
typeabbrev fsubst (memory ftm).
kind assmt type.
type abs (tm -> assmt) -> assmt.
type val tm -> assmt.
typeabbrev subst (memory assmt).
```

We call fsubst the memory of  $\mathcal{F}_0$ , while we call subst the one of  $\mathcal{H}_0$ . Both have the invariant that they are not cyclic, TODO explain. Other invariant: the terms in ho\_subst never contains eta and beta expansion

#### 4.1 Equational theory and Unification

here we give the functions/signatures to express the properties 3-5 in the problem statement

Together with the description of the terms of the language, we need some auxiliary functions to perform operations like term equality, unification, dereferencing. There predicates are supposed to be implemented in the OL and the ML following respectively their specification. In the following few paragraphs we give a brief proposition of them.

*Term dereferencing:  $\rho s$  and  $\sigma t$ .* Since in our encoding we explicitly carry a substitution we need to define the operation that applies it to a term. Its proposition takes care to normalize (flatten) applications, for example it turns (app [app [con "f", con "a"], con "b"]) into (app [con "f", con "a", con "b"]).

dereference variables of the two languages. This is particularly useful to check if two terms in the OL (resp. in the ML) are equal. The constructors representing rigid terms, namely app, lam and con, make the dereferencing procedure to recurse over the their subterms. The code below display this behavior for the dereferencing performed by the OL:

```
type fder fsubst -> fm -> fm -> o.
fder S (fapp A) (fapp B) :- map (fder S) A B.
fder S (flam F) (flam G) :-
  pi x\ fder S x x => fder S (F x) (G x).
fder _ (fcon C) (fcon C).
fder S (fuva N) R :- set? N S T, fder S T R.
fder S (fuva N) (fuva N) :- unset? N S.
```

```
type fderef fsubst -> fm -> fm -> o.          ( $\rho s$ )
fderef S T R :- fder S T T', napp T' R.
```

```
type napp fm -> fm -> o.
```

TODO explain napp. We use the cut operator to keep the code compact. It is possible to rewrite the rule for application traversal



```

465 type ( $=_{\lambda}$ ) tm  $\rightarrow$  tm  $\rightarrow$  o.
466 app A  $=_{\lambda}$  fapp B :- map ( $=_{\lambda}$ ) A B.
467 lam F  $=_{\lambda}$  flam G :- pi x\ x  $=_{\lambda}$  x  $\Rightarrow$  F x  $=_{\lambda}$  G x.
468 con C  $=_{\lambda}$  fcon C.
469 uva N A  $=_{\lambda}$  fuva N B :- map ( $=_{\lambda}$ ) A B.

```

Figure 2: Equal predicate ML

so that it is mutually exclusive with the first one, but that requires a rather verbose analysis of the head of  $A$ .

The corresponding code for  $\mathcal{H}_0$  is similar, we only show the last two rules that differ in a substantial way:

```

476 type deref subst  $\rightarrow$  tm  $\rightarrow$  tm  $\rightarrow$  o. (σt)
477 deref S (app A) (app B) :- map (deref S) A B.
478 deref S (lam F) (lam G) :-
479   pi x\ deref S x x  $\Rightarrow$  deref S (F x) (G x).
480 deref _ (con C) (con C).
481 deref S (uva N A) R :- set? N S F, move F A T, deref S T R.
482 deref S (uva N A) (uva N B) :- unset? N S, map (deref S) A B.
483 type move assignment  $\rightarrow$  list tm  $\rightarrow$  tm  $\rightarrow$  o.
484 move (abs Bo) [H|L] R :- move (Bo H) L R.
485 move (val A) [] A :- !.
486 move (val (uva N A)) L (uva N X) :- std.append A L X.

```

TODO: no need to napp, see the beta section. Note that when the substitution  $S$  maps a unification variable  $N$  to an assignment  $F$  we

....  
 TODO: invariant: variables in subst are always fully applied, and length of scope is the arity of the HO variable.

Important!!! A different reasoning is to be addressed to the variables of the ML. Firstly, a meta variable cannot appear in the app node as the first element of the list, we will explain why in section 5

```

496 type ( $=_o$ ) ftm  $\rightarrow$  ftm  $\rightarrow$  o. (=o)
497 fapp A  $=_o$  fapp B :- map ( $=_o$ ) A B.
498 flam F  $=_o$  flam G :- pi x\ x  $=_o$  x  $\Rightarrow$  F x  $=_o$  G x.
499 fcon C  $=_o$  fcon C.
500 fuva N  $=_o$  fuva N.
501 flam F  $=_o$  T :- (ηl)
502   pi x\ beta T [x] (R x), x  $=_o$  x  $\Rightarrow$  F x  $=_o$  R x.
503 T  $=_o$  flam F :- (ηr)
504   pi x\ beta T [x] (R x), x  $=_o$  x  $\Rightarrow$  R x  $=_o$  F x.
505 fapp [flam X | L]  $=_o$  T :- beta (flam X) L R, R  $=_o$  T. (βl)
506 T  $=_o$  fapp [flam X | L] :- beta (flam X) L R, T  $=_o$  R. (βr)

```

*Term equality:*  $=_o$  vs.  $=_{\lambda}$ . We can test if two terms are equal following the equational theory of the language being considered. In ?? we provide an implementation of the  $=_o$  predicate. The first four rules check if the two terms are equal regarding the structure of the current node, that is, two terms are equal if they have same head and if recursively each subterm is two by two equal. Moreover, since the theory of the OL accepts  $\eta$ - and  $\beta$ -equivalence, then we implement the corresponding rules.

The equality relation for the ML, accepts  $\eta\beta$ -equivalence between terms of the ML. Recall that  $\text{abs } x \backslash f \ x$ , is a valid  $\eta$  expansion of the function  $f$  and that  $\text{lam } x \backslash \text{app}[f, x]$  is not that equivalent to  $f$  at meta level. However, since we are interested in using the unification procure of the ML, by eq. (1), we can use the  $\approx_{\lambda}$  relation to test, when needed if two terms are equal in the ML.

*Term unification:*  $\approx_o$  vs.  $\approx_{\lambda}$ . The last but not least important relation we should take care of before presenting our full algorithm aiming to unify terms of the OL in the ML and provide the substitution produced in the ML to the OL, is term unification. This procedure is a more powerful version of the equal predicate, since unification checks if two terms can be equal be assigning unification variables. In our representation, variable assignment (or refinement) is performed by modifying the corresponding substitution mapping. We will not give an implementation of  $\approx_o$ , since we are giving an implementation of it using our algorithm, see ??.

```

523 type ( $\approx_{\lambda}$ ) tm  $\rightarrow$  tm  $\rightarrow$  subst  $\rightarrow$  subst  $\rightarrow$  o.

```

On the other hand, unification in the ML needs to be defined. In fig. 5, we give an implementation of  $\approx_{\lambda}$  but that is actually what our meta language provides as a builtin.

This predicate has four arguments, the two terms  $t_1$  and  $t_2$  to unify, the old substitution map  $\rho_1$ , and the new substitution map  $\rho_2$ , with the invariant  $\rho_1 \subseteq \rho_2$ . The first three rules unify terms with same rigid heads, and call the unification relation on the sub-terms. If  $t_1$  (resp.  $t_2$ ) is an assigned variables,  $t_1$  is dereferenced to  $t'_1$  (resp.  $t'_2$ ) and the unification is called between  $t'_1$  and  $t_2$  (resp.  $t_1$  and  $t'_2$ ). If both terms are unification variables, we test that their arguments are in the pattern fragment, we allocate a new variable  $w$  in  $\rho_1$  such that  $w$  is the pruning of the arguments of  $t_1$  and  $t_2$ , we assign both  $t_1$  and  $t_2$  to  $w$  and return the new mapping  $\rho_2$  containing all the new variable assignment. Finally, if only one of the two terms is an unification variable  $v$ , after having verified that  $v$  does not occur in the other term  $t$ , we bind  $v$  to  $t$  and return the new substitution mapping.

OLD

A key property needed in unification is being able to verify if two terms are equal wrt a given equational theory. This relation allow to compare terms under a certain substitution mapping, so that any time a variable  $v$  is assigned in a subterm, a dereferencing of  $v$  is performed. After variable dereferencing, the test for equality is continued on the new-created subterm.

The base equality function over terms can be defined as follows:

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.

In the following section we explain how we deal with term (de)compilation and links between unification variables.

## 5 BASIC COMPILATION $\mathcal{F}_0$ TO $\mathcal{H}_0$

In this section we describe a basic compilation scheme that we refine later, in the following sections.

The objective of the compilation is to recognize the higher-order variables available in  $\mathcal{H}_0$  when expressed in a first order way in  $\mathcal{F}_0$ . The compiler also generates a list of links that are used to bring back the substitution from  $\mathcal{H}_0$  to  $\mathcal{F}_0$  and allocates in the memory a cell for each variable.

same  
or  
⊇  
or  
⊆

```

581 kind link type.
582 type link nat -> nat -> nat -> subst. % link Fo Ho Arity
583 typeabbrev links list link.
584 type comp fm -> tm -> links -> links -> subst -> subst -> o.
585 comp (fcon X) (con X) L L S S.
586 comp (flam F) (lam G) K L R S :- pi x y\
587   (pi A S\ comp x y L L S S) => comp (F x) (G y) K L R S.
588 comp (fuva M) (uva N []) K [link M N z|K] R S :- new R N S.
589 comp (fapp[fuva M|A]) (uva N B) K L R S :- distinct A, !,
590   fold4 comp A B K R R,
591   new R N S, len A Arity,
592   L = [link N M Arity | K].
593 comp (fapp A) (app B) K L R S :- fold4 comp A B K L R S.

```

Note that link carries the arity (number of expected arguments) of the variable.

say  
when  
this is  
needed

```

596 type solve-links links -> links -> subst -> subst -> o.
597 solve-links L L S S.
598
599 Then decomp
600
601 type decompile links -> subst -> fsubst -> o.
602 decompile L S O :-
603   map (_\r\l = none) S O1, % allocate empty fsubst
604   (pi N X\ knil N X :- mem L (link X N _) ; N = X) =>
605   decomp1 S L O1 O.
606 type knil nat -> nat -> o.
607
608 type decomp1 links -> subst -> fsubst -> o.
609 decomp1 S [] [].
610 decomp1 S [link _ N _|L] O P :- unset? N S X,
611   decomp1 S L O P.
612 decomp1 S [link M N _|L] O P :- set? N S X,
613   decomp-assignment S X T, assign M O (some T) O1,
614   decomp1 S L O1 P.
615
616 type decomp-assignment subst -> assignment -> fm -> o.
617 decomp-assignment S (abs F) (flam G) :-
618   pi x y\ decomp-tm S x y => decomp-assignment S (F x) (G y).
619 decomp-assignment S (val T) T1 :- decomp S T T1.

```

TODO  
link  
TODO  
nuove  
subst  
TODO:  
code  
unif

```

620
621 type decomp subst -> tm -> fm.
622 decomp _ (con C) (fcon C).
623 decomp S (app A) (app B) :- map (decomp S) A B.
624 decomp S (lam F) (flam G) :-
625   pi x y\ decomp S x y => decomp S (F x) (G y).
626 decomp S (uva N A) R :- set? N S F,
627   move F A T, decomp S T R.
628 decomp S (uva N A) R :- unset? N S,
629   map (decomp S) A B, knil N M, napp (fapp[fuva M|B]) R.
630
631 Now unif
632
633 type (≈o) fm -> fm -> subst -> subst -> o.
634 (X ≈o Y) S S1 :-
635   fderef S X X0, fderef S Y Y0,
636   comp X0 X1 [] S0 [] L0,
637   comp Y0 Y1 S0 S1 L0 L1,
638   (X1 ≈λ Y1) [] HS0,
639

```

(norm)  
(compile)  
(unify)

```

solve-links L1 L2 HS0 HS1, (link)
decompile L2 HS1 S1. (decompile)

```

## 5.1 Prolog simulation

Allows us to express the properties. we take all terms involved in a search (if a rule is used twice we simply take a copy of it), we compile all of them, and then we pick the unification problems among these terms and step through them.

```

type pick list A -> (pair nat nat) -> (pair A A) -> o.
pick L (pr X Y) (pr TX TY) :- nth X L TX, nth Y L TY.

type prolog-fo list fm -> list (pair nat nat) -> subst -> o.
prolog-fo Terms Problems S :-
  map (pick Terms) Problems FoProblems,
  fold4 (≈o) FoProblems [] S.

```

```

type step-ho (pair tm tm) -> links -> links -> subst -> subst -> o.
step-ho (pr X Y) L0 L1 S0 S2 :-
  (X1 ≈λ Y1) S0 S1,
  solve-links L0 L1 S1 S2.

```

```

type prolog-ho list fm -> list (pair nat nat) -> subst -> o.
prolog-ho Terms Problems S :-
  fold4 comp Terms HoTerms [] L0 [] HS0,
  map (pick HoTerms) Problems HoProblems,
  fold4 step-ho HoProblems L0 L HS0 HS,
  decompile L HS S.

```

the property is that if a step for Fo succeeds then the Ho one does, and if Fo fails then the Ho fails ()

## 5.2 Example

OK

```

Terms [ flam x\ fapp[fcon"g",fapp[fuva z, x]]
      , flam x\ fapp[fcon"g", fcon"a"] ]
Problems = [ pr z (s z) ] % λx.g(Fx) = λx.ga
lam x\ app[con"g",uva z [x]] ≈o lam x\ app[con"g", con"a"]
link z z (s z)
HS = [some (abs x\con"a")]
S = [some (flam x\fcon a)]

```

KO

```

Terms [ flam x\ fapp[fcon"g",fapp[fuva z, x]]
      , flam x\ fapp[fcon"g", fcon"a"] ]
Problems = [ pr 0 1 % A = λx.x
            , pr 2 3 ] % Aa = a
lam x\ app[con"g",uva z [x]] ≈o lam x\ app[con"g", con"a"]
link z z (s z)
HS = [some (abs x\con"a")]
S = [some (flam x\fcon a)]
lam x\ app[f, app[X, x]] = Y,
lam x\ x[] = X.

```

**TODO: Goal:**  $s_1 \approx_o s_2$  is compiled into  $t_1 \approx_\lambda t_2$

**TODO: What is done:** uvars  $fo_{uv}$  of OL are replaced into uvars  $ho_{uv}$  of the ML

**TODO: Each  $fo_{uv}$  is linked to an  $ho_{uv}$  of the OL**

**TODO: Example needing the compiler v0 (tra l'altro lo scope è ignorato):**

```
lam x\ app[con"g", app[uv 0, x]] ≈o lam x\ app[con"g", c"a"]
```

**TODO: Links used to instantiate vars of elpi**

**TODO: After all links, the solution in links are compacted and given to coq**

**TODO: It is not so simple, see next sections (multi-vars, eta, beta)**

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.

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 same 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.

decomp che mappa abs verso lam **TODO: An other example:**

```
lam x\ app[f, app[X, x]] = Y, (lam x\ x) = X.
```

## 6 USE OF MULTIVARS

Se il termine iniziale è della forma

```
app[con"xxx", (lam x\ lam y\ Y y x), (lam x\ f)]
=
app[con"xxx", X, X]
```

allora se non uso due X diverse non ho modo di recuperare il quoziente che mi manca.

a sto punto consideriamo liste di problemi e così da eliminare sta xxx senza perdita di generalità (e facciamo problemi più corti, e modellizziamo anche la sequenza)

### 6.1 Problems with $\eta$

**TODO: The following goal necessita v1 (lo scope è usato):**

```
X = lam x\ lam y\ Y y x, X = lam x\ f
```

**TODO: The snd unif pb, we have to unif lam x\ lam y\ Y x y with lam x\ f**

**TODO: It is not doable, with the same elpi var**

Invarianti: A destra della eta abbiamo sempre un termine che comincia per  $\lambda x.bla$

La deduplicate eta: - viene chiamata che della forma [variable] -> [eta1] e [variable] -> [eta2] (a destra non c'è mai un termine con testa rigida) - i due termini a dx vengono unificati con la unif e uno dei due link viene buttato -> si mettono più lambda - dopo l'unificazione rimane un link [variabile] -> [etaX] e quando risolviamo un eta-link, allora dobbiamo eta-espandere qb il termine di sinistra per poterlo unificare con quello di dx.

### 6.2 Problems with $\beta$

$\beta$ -reduction problems ( $\diamond\beta$ ) appears any time we deal with a subterm  $t = Xt_1 \dots t_n$ , where  $X$  is flexible and the list  $[t_1 \dots t_n]$  in not in  $\mathcal{L}_\lambda$ . This unification problem is not solvable without loss of generality, since there is not a most general unifier. If we take back the example given in section 2.1, the unification  $Fa = a$  admits two solutions for  $F$ :  $\rho_1 = \{F \mapsto \lambda x.x\}$  and  $\rho_2 = \{F \mapsto \lambda_.a\}$ . Despite this, it is possible to work with  $\diamond\beta$  if an oracle provides a substitution  $\rho$  such that  $\rho t$  falls again in the  $\mathcal{L}_\lambda$ .

On the other hand, the  $\approx_\lambda$  is not designed to understand how the  $\beta$ -redexes work in the onject language. Therefore, even if we know that  $F$  is assigned to  $\lambda x.x$ ,  $\approx_\lambda$  is not able to unify  $Fa$  with  $a$ . On the other hand, the problem  $Fa = G$  is solvable by  $\approx_\lambda$ , but the final result is that  $G$  is assigned to  $(\lambda x.x)a$  which breaks the invariant saying that the substitution of the meta language always contain only terms in normal form.

The solution to this problem is to modify the compiler such that any sub-term  $t$  considered as a potential  $\beta$ -redex is replaced with a hole  $h$  and a new dedicated link, called `link- $\beta$` .

```
type link-beta tm -> tm -> link.
```

This link carries two terms, the former representing the variable  $h$  for the new created hole and the latter containing the subterm  $t$ . As for the `link- $\eta$` , we will call  $h$  and  $t$  respectively the left *lhs* and the right *rhs* hand side of the `link- $\beta$` .

At compile time, a subterm is  $\diamond\beta$ , if it has the shape `fapp[fuva N | L]` and distinct  $L$  does not hold. In that case,  $L$  is split in two sublist `PF` and `NPF` such that former is the longest prefix of  $L$  such that distinct `PF` holds. `NPF` is the list such that append `PF NPF L`. The *lhs* is set to a new variable named  $M$  with `PF` in scope whereas

the  $rhs$  is given by the term  $\text{app}[\text{uva } N' \text{ PF } | \text{ NPF}]$  where the  $\mathcal{H}_0$  variable identified by  $N'$  is mapped to the  $\mathcal{F}_0$  variable named  $N$ .

After its creation, a link- $\beta$  remain suspended until the head of  $rhs$  is instantiated by the oracle (see eq. (5)). In this case,  $rhs$  is  $\beta$ -reduced to a new term, say  $t'$ .  $t'$  is either a term in  $\mathcal{L}_\lambda$ , in which case  $t'$  is unified with the  $lhs$ , otherwise, the link remain suspended and no progress is performed. Another way to wake a link- $\beta$  up is when the  $lhs$  is a term  $T$  and  $rhs$  has the shape  $\text{app}[\text{uva } N \text{ PF } | \text{ NPF}]$  and some of the arguments in the  $\text{NPF}$  list become names. This is possible after the resolution of other links. In this case, the list  $L$  obtained by the concatenation between  $\text{PF}$  and  $\text{NPF}$  is split again in to lists  $\text{PF}'$  and  $\text{NPF}'$ . If  $\text{PF}$  is not that same as  $\text{PF}'$ , then we can 1) remove the current link- $\beta$ , 2) create a new link- $\beta$  between  $T$  and  $\text{app}[\text{uva } N' \text{ PF}' | \text{ NPF}']$  and 3) create a new link- $\eta$  between the variables  $N$  and  $N'$ .

An example justifying this last link manipulation is given by the following unification problem:

```
f = flam x\ flam y\ fapp[F, (X x), y] % f = λx.λy.F(Xx)y
```

from the substitution  $\rho = \{X \mapsto \lambda x.x\}$ .

**TODO: The following goal:  $x = \text{lam } x\ x$ ,  $\text{app}[x, a] = a$**

**TODO: We use links-beta**

### 6.3 Tricky examples

```
triple ok (@lam x\ @app[@f, @app[@X, x]]) @Y,
triple ok @X (@lam x\ x),
triple ok @Y @f

% ok! 22 [
%   triple ok (@lam x\ @lam y\ @app[@Y, y, x]) @X,
%   triple ok (@lam x\ @f) @X,
% ].
```

## 7 FIRST ORDER APPROXIMATION

**TODO: Coq can solve this:  $f \ 1 \ 2 = x \ 2$ , by setting  $X$  to  $f \ 1$**

**TODO: We can re-use part of the algo for  $\beta$  given before**

## 8 UNIF ENCODING IN REAL LIFE

**TODO: Il ML presentato qui è esattamente elpi**

**TODO: Il OL presentato qui è esattamente coq**

**TODO: Come implementiamo tutto ciò nel solver**

## 9 RESULTS: STDPP AND TLC

**TODO: How may rule are we solving?**

**TODO: Can we do some perf test**

## 10 CONCLUSION

## 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, Claudio Sacerdoti Coen, and Enrico Tassi. “Implementing HOL in an Higher Order Logic Programming Language”. In: *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. LFMTTP '16. Porto, Portugal: Association for Computing Machinery, 2016. ISBN: 9781450347778. DOI: 10.1145/2966268.2966272. URL: <https://doi.org/10.1145/2966268.2966272>.
- [3] Cvetan Dunchev et al. “ELPI: Fast, Embeddable,  $\lambda$ Prolog 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\\_32](http://dx.doi.org/10.1007/978-3-662-48899-7_32).
- [4] 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.
- [5] 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.
- [6] Davide Fissore and Enrico Tassi. “A new Type-Class solver for Coq in Elpi”. In: *The Coq Workshop 2023*. Bialystok, Poland, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- [7] Benjamin Grégoire, Jean-Christophe L  chenet, and Enrico Tassi. “Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin’s data types with Coq-Elpi”. In: *CPP '23: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. Boston MA USA, France: ACM, Jan. 2023, pp. 167–181. DOI: 10.1145/3573105.3575683. URL: <https://inria.hal.science/hal-03800154>.
- [8] 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.
- [9] 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.
- [10] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. DOI: 10.1017/CBO9781139021326.
- [11] 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.
- [12] 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>.
- [13] 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.



- [14] 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.
- [15] Colin Rothgang, Florian Rabe, and Christoph Benz Müller. “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.
- [16] Enrico Tassi. “Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq”. In: *ITP 2019 - 10th International Conference on Interactive Theorem Proving*. Portland, United States, Sept. 2019. DOI: 10.4230/LIPIcs.CVIT.2016.23. URL: <https://inria.hal.science/hal-01897468>.
- [17] Enrico Tassi. “Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi  $\lambda$ Prolog dialect)”. In: *The Fourth International Workshop on Coq for Programming Languages*. Los Angeles (CA), United States, Jan. 2018. URL: <https://inria.hal.science/hal-01637063>.
- [18] The Coq Development Team. *The Coq Reference Manual – Release 8.18.0*. <https://coq.inria.fr/doc/V8.18.0/refman>. 2023.
- [19] 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>.
- [20] 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.

**APPENDIX**

Note that  $(a \text{ infix } b) \text{ c d}$  de-sugars to  $(\text{infix}) \text{ a b c d}$ .

```

1161  type ( $=_o$ )  $ftm \rightarrow ftm \rightarrow o$ .                                ( $=_o$ )
1162  fapp A  $=_o$  fapp B :- map ( $=_o$ ) A B.                                1219
1163  flam F  $=_o$  flam G :-  $\pi x \backslash x =_o x \Rightarrow F x =_o G x$ .            1220
1164  fcon C  $=_o$  fcon C.                                                1221
1165  fuva N  $=_o$  fuva N.                                                1222
1166  flam F  $=_o$  T :-                                                ( $\eta_l$ )
1167      $\pi x \backslash \text{beta } T [x] (R x), x =_o x \Rightarrow F x =_o R x$ .        1223
1168  T  $=_o$  flam F :-                                                ( $\eta_r$ )
1169      $\pi x \backslash \text{beta } T [x] (R x), x =_o x \Rightarrow R x =_o F x$ .        1224
1170  fapp [flam X | L]  $=_o$  T :- beta (flam X) L R, R  $=_o$  T. ( $\beta_l$ )      1225
1171  T  $=_o$  fapp [flam X | L] :- beta (flam X) L R, T  $=_o$  R. ( $\beta_r$ )      1226
1172                                                                    1227
1173  type beta fm  $\rightarrow$  list fm  $\rightarrow$  fm  $\rightarrow o$ .                            1228
1174  beta A [] A.                                                    1229
1175  beta (flam F) [H | L] R :- subst F H B,                        1230
1176     beta B L R. % since F could be  $x \backslash \text{app}[x] \_$  and H be  $\text{lam } \_$ 
1177  beta (fapp A) L (fapp X) :- append A L X.                      1231
1178  beta (fuva N) L (fapp [fuva N | L]).                            1232
1179  beta (fcon H) L (fapp [fcon H | L]).                            1233
1180                                                                    1234
1181  type subst (fm  $\rightarrow$  fm)  $\rightarrow$  fm  $\rightarrow$  fm  $\rightarrow o$ .                      1235
1182  subst F H B :- napp (F H) B. % since (F H) may generate (app[app _|_])
1183                                                                    1236
1184  type napp fm  $\rightarrow$  fm  $\rightarrow o$ .                                        1237
1185  napp (fcon C) (fcon C).                                         1238
1186  napp (flam F) (flam G) :-  $\pi x \backslash \text{napp } x x \Rightarrow \text{napp } (F x) (G x)$ .
1187  napp (fapp[fapp L|M]) R :- !, append L M N, napp (fapp N) R.
1188  napp (fapp[X]) R :- !, napp X R.
1189  napp (fapp A) (fapp B) :- map napp A B.
1190  napp (fuva N) (fuva N).
1191                                                                    1239
1192  Figure 3: Full implementation of the  $=_o$  predicate for  $\mathcal{F}_o$ 
1193                                                                    1240
1194                                                                    1241
1195                                                                    1242
1196                                                                    1243
1197                                                                    1244
1198                                                                    1245
1199                                                                    1246
1200                                                                    1247
1201                                                                    1248
1202                                                                    1249
1203                                                                    1250
1204                                                                    1251
1205                                                                    1252
1206                                                                    1253
1207                                                                    1254
1208                                                                    1255
1209                                                                    1256
1210                                                                    1257
1211                                                                    1258
1212                                                                    1259
1213                                                                    1260
1214                                                                    1261
1215                                                                    1262
1216                                                                    1263
1217                                                                    1264
1218                                                                    1265
1219                                                                    1266
1220                                                                    1267
1221                                                                    1268
1222                                                                    1269
1223                                                                    1270
1224                                                                    1271
1225                                                                    1272
1226                                                                    1273
1227                                                                    1274
1228                                                                    1275
1229                                                                    1276

```

```

1277 type ( $\approx_\lambda$ ) tm -> tm -> subst -> subst -> o.
1278 % Congruence
1279 (app A  $\approx_\lambda$  app B) R S :- fold2 ( $\approx_\lambda$ ) A B R S.
1280 (lam F  $\approx_\lambda$  lam G) R S :- pi x\ (pi S\ (x  $\approx_\lambda$  x) S S) => (F x  $\approx_\lambda$  G x) R S.
1281  $\approx_\lambda$  (con C) (con C) S S.
1282 % deref
1283 (uva N A  $\approx_\lambda$  T) R S :- set? N S F, move F A T1, (T1  $\approx_\lambda$  T) R S.
1284 (T  $\approx_\lambda$  uva N A) R S :- set? N S F, move F A T1, (T  $\approx_\lambda$  T1) R S.
1285 % flex-flex
1286 (uva N A  $\approx_\lambda$  uva M B) S S3 :- unset? M, unset? N,
1287   distinct A, distinct B,
1288   new S W S1, prune W Args1 B Ass,
1289   assign N S1 Ass S2, assign M S2 Ass S3.
1290 % assignment
1291 (uva N A  $\approx_\lambda$  T) R S :- distinct A, not (T = uva _ _), not_occ N S T,
1292   bind A T T1, assign N S T1 S1.
1293 (T  $\approx_\lambda$  uva N A) R S :- distinct A, not (T = uva _ _), not_occ N S T,
1294   bind A T T1, assign N S T1 S1.
1295
1296 type distinct list A -> o.
1297 distinct [].
1298 distinct [X|XS] :- name X, not(mem X XS),
1299 distinct XS.
1300
1301 typeabbrev memory A (list (option A)).
1302 type set? nat -> memory A -> A -> o.
1303 set? N S T :- nth N S (some T).
1304 type unset? nat -> memory A -> o.
1305 unset? N S :- nth N S none.
1306 type assign nat -> memory A -> A -> memory A -> o.
1307 assign z [none|M] T [some T|M].
1308 assign (s N) [X|M] T [X|M1] :- assign N M T M1.
1309 kind nat type.
1310 type z nat.
1311 type s nat -> nat.
1312 type nth nat -> list A -> A -> o.
1313 nth z [X|_] X.
1314 nth (s N) [_|L] X :- nth N L X.
1315
1316 type new memory A -> nat -> memory A -> o.
1317 new [] z [none].
1318 new [X|XS] (s N) [X|YS] :- new XS N YS.
1319
1320 type prune .
1321 type move .
1322 type beta.
1323 type bind.
1324 type not_occ.
1325 TODO
1326
1327 type fold2 (A -> A1 -> B -> B -> o) -> list A -> list A1 -> B -> B -> o.
1328 fold2 _ [] [] A A.
1329 fold2 F [X|XS] [Y|YS] A A1 :- F X Y A A0, fold2 F XS YS A0 A1.

```

Figure 4: Implementation of the  $\approx_\lambda$  predicate for  $\mathcal{H}_0$



```

1393  type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A -> list A1 -> B -> B -> C -> C -> o.
1394  fold4 _ [] [] A A B B.
1395  fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0, fold4 F XS YS A0 A1 B0 B1.
1396
1397  type len list A -> nat -> o.
1398  len [] z.
1399  len [_|L] (s X) :- len L X.

```

Figure 5: Implementation of the compiler