

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], λ 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 λ Prolog. Elpi's equational theory comprises $\eta\beta$ equivalence and comes equipped with a higher order unification procedure \approx_λ restricted to the pattern fragment [9]. We want \approx_o to be as powerful as \approx_λ 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_λ 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_λ , effectively implementing \approx_o on top of \approx_λ 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, 11 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.
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], λ 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 λ 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 A P»` 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) β -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_λ 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}_λ [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}_λ [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_λ solves higher-order problems in \mathcal{L}_λ .

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

$x \setminus f \ x$	\approx_λ	f
$\text{lam } A \ x \setminus \text{app}[\text{con} "f", x]$	\approx_o	$\text{con} "f"$
$\text{lam } A \ x \setminus \text{app}[\text{con} "f", x]$	\neq_λ	$\text{con} "f"$
$P \ x$	\approx_λ	x
$\text{app}[P, x]$	\approx_o	x
$\text{app}[P, x]$	\neq_λ	x

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

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

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

To state precisely the problem we solve we need a \mathcal{F}_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_λ 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 σ ; we write σt for the application of the substitution to t , and $\sigma X = \{\sigma t \mid t \in X\}$ when X is a set; we write $\sigma \subseteq \sigma'$ when σ is more general than σ' . 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 ρ_0 produces the final substitution ρ_N .¹ The initial here ρ_0 is the empty substitution

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

We simulate each run in \mathcal{F}_0 with a run in \mathcal{H}_0 as follows. Note that σ_0 is the empty substitution.

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

Here *hstep* is made of two sub-steps: a call to \simeq_λ (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 $s_1 \simeq_o s_2$ by specializing the code of *hrun* to $\mathcal{S} = \{s_1, s_2\}$ as follows:

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

PROPOSITION 2.3 (PROPERTIES OF \simeq_o).

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

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

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

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

Property 5 states that \simeq_o , hence our compilation scheme, is resilient to unification problems outside \mathcal{L}_λ solved by a third party.

¹If 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

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

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

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 p is replaced by a fresh unification variable h and an accessory link that represent a suspended unification problem $h \simeq_\lambda p$. As a result \simeq_λ is “well behaved” on t , meaning it does not contradict $=_o$ (as it would do on “problematic” terms). We now define “problematic” and “well behaved” more formally.

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

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 ($\diamond\beta$). $\diamond\beta = \{Xt_1 \dots t_n \mid Xt_1 \dots t_n \notin \mathcal{L}_\lambda\}$.

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 $\mathcal{P}(t)$).

$$\begin{aligned} t &\in \mathcal{P}(t) \\ t &= ft_1 \dots t_n \Rightarrow \mathcal{P}(t_i) \subseteq \mathcal{P}(t) \wedge f \in \mathcal{P}(t) \\ t &= \lambda x. t' \Rightarrow \mathcal{P}(t') \subseteq \mathcal{P}(t) \end{aligned}$$

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

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

$$\mathcal{W}(X) \Leftrightarrow \forall t \in \mathcal{P}(X), t \notin (\diamond\beta \cup \diamond\eta)$$

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

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

A less formal way to state 2.8 is that *hstep* never “commits” an unneeded λ -abstraction in σ (a λ that could be erased by an η -contraction), nor puts in σ a flexible application outside \mathcal{L}_λ (an application node that could be erased by a β -reduction).

Note that proposition 2.8 does not hold for \simeq_o since decompilation can introduce (actually restore) terms in $\diamond\eta$ or $\diamond\beta$ that were move out of the way (put in \mathbb{L}) by compilation.

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 tm). A fully shallow encoding unfortunately does not fit our use case, although it would make the running example work:

```

349   finite (fin N).
350   decision (nfact N NF).
351   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:

```

357   Fixpoint arr T n := if n is S m then T -> arr T m else T.
358   Definition sum n : arr nat n := ...
359   Check sum 2 7 8 : nat.
360   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.

```

383   kind fm type.           kind tm type.
384   type fapp list fm -> fm. type app list tm -> tm.
385   type flam (fm -> fm) -> fm. type lam (tm -> tm) -> tm.
386   type fcon string -> fm.   type con string -> tm.
387   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}_λ iff `distinct L` holds.

```

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

```

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

²one could always load name x for every x under a pi and get rid of the name builtin

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:

```

407   typeabbrev memory A (list (option A)).
408   type set?   nat -> memory A -> A -> o.
409   type unset? nat -> memory A -> o.
410   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.

```

421   typeabbrev fsubst (memory ftm).
422   kind assmt type.
423   type abs (tm -> assmt) -> assmt.
424   type val tm -> assmt.
425   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: ρs and σt . 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 sub-terms. The code below display this behavior for the dereferencing performed by the OL:

```

454   type fder fsubst -> fm -> fm -> o.
455   fder S (fapp A) (fapp B) :- map (fder S) A B.
456   fder S (flam F) (flam G) :-
457     pi x\ fder S x x => fder S (F x) (G x).
458   fder _ (fcon C) (fcon C).
459   fder S (fuva N) R :- set? N S T, fder S T R.
460   fder S (fuva N) (fuva N) :- unset? N S.
461
462   type fderef fsubst -> fm -> fm -> o.
463   fderef S T R :- fder S T T', napp T' R.

```

(ρs)


```

465 type (=λ) tm -> tm -> o.
466 app A =λ fapp B :- map (=λ) A B.
467 lam F =λ flam G :- pi x\ x =λ x => F x =λ G x.
468 con C =λ fcon C.
469 uva N A =λ fuva N B :- map (=λ) A B.

```

Figure 2: Equal predicate ML

```

473 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 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:

```

480 type deref subst -> tm -> tm -> o. (σt)
481 deref S (app A) (app B) :- map (deref S) A B.
482 deref S (lam F) (lam G) :-
483   pi x\ deref S x x => deref S (F x) (G x).
484 deref _ (con C) (con C).
485 deref S (uva N A) R :- set? N S F, move F A T, deref S T R.
486 deref S (uva N A) (uva N B) :- unset? N S, map (deref S) A B.
487 type move assignment -> list tm -> tm -> o.
488 move (abs Bo) [H|L] R :- move (Bo H) L R.
489 move (val A) [] A :- !.
490 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 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

```

500 type (=o) ftm -> ftm -> o. (=o)
501 fapp A =o fapp B :- map (=o) A B.
502 flam F =o flam G :- pi x\ x =o x => F x =o G x.
503 fcon C =o fcon C.
504 fuva N =o fuva N.
505 flam F =o T :- (ηl)
506   pi x\ beta T [x] (R x), x =o x => F x =o R x.
507 T =o flam F :- (ηr)
508   pi x\ beta T [x] (R x), x =o x => R x =o F x.
509 fapp [flam X | L] =o T :- beta (flam X) L R, R =o T. (βl)
510 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 η - and β -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 η 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 $=_\lambda$ relation to test, when needed if two terms are equal in the ML.

Term unification: \approx_o vs. \approx_λ . 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 by 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 ??.

```

538 type (≈λ) tm -> tm -> subst -> subst -> o.

```

On the other hand, unification in the ML needs to be defined. In ??, we give an implementation of \approx_λ 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 ρ_1 , and the new substitution map ρ_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 ρ_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 ρ_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}_o TO \mathcal{H}_o

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

same
or
⊇
or
⊆

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

```

kind link type.
type link nat -> nat -> nat -> subst. % link Fo Ho Arity
typeabbrev links list link.
type comp fm -> tm -> links -> links -> subst -> subst -> o.
comp (fcon X) (con X) L L S S.
comp (flam F) (lam G) K L R S :- pi x y\
  (pi A S\ comp x y L L S S) => comp (F x) (G y) K L R S.
comp (fuva M) (uva N [I]) K [link M N z|K] R S :- new R N S.
comp (fapp[fuva M|A]) (uva N B) K L R S :- distinct A, !,
  fold4 comp A B K K R R,
  new R N S, len A Arity,
  L = [link N M Arity | K].
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.

```

type solve-links links -> links -> subst -> subst -> o.
solve-links L L S S.

```

Then decomp

```

type decompile links -> subst -> fsubst -> o.
decompile L S O :-
  map (\r\l = none) S O1, % allocate empty fsubst
  (pi N X\ knil N X :- mem L (link X N _) ; N = X) =>
  decomp L S L O1 O.
type knil nat -> nat -> o.

type decomp1 links -> subst -> fsubst -> o.
decomp1 S [I] [I].
decomp1 S [link _ N _|L] O P :- unset? N S X,
  decomp1 S L O P.
decomp1 S [link M N _|L] O P :- set? N S X,
  decomp-assignment S X T, assign M O (some T) O1,
  decomp1 S L O1 P.

type decomp-assignment subst -> assignment -> fm -> o.
decomp-assignment S (abs F) (flam G) :-
  pi x y\ decomp-tm S x y => decomp-assignment S (F x) (G y).
decomp-assignment S (val T) T1 :- decomp S T T1.

type decomp subst -> tm -> fm.
decomp _ (con C) (fcon C).
decomp S (app A) (app B) :- map (decomp S) A B.
decomp S (lam F) (flam G) :-
  pi x y\ decomp S x y => decomp S (F x) (G y).
decomp S (uva N A) R :- set? N S F,
  move F A T, decomp S T R.
decomp S (uva N A) R :- unset? N S,
  map (decomp S) A B, knil N M, napp (fapp[fuva M|B]) R.

```

Now unif

```

type (≈o) fm -> fm -> subst -> subst -> o.
(X ≈o Y) S S1 :-
  fderef S X X0, fderef S Y Y0,
  comp X0 X1 [I] S0 [I] L0,
  comp Y0 Y1 S0 S1 L0 L1,
  (X1 ≈λ Y1) [I] HS0,
  solve-links L1 L2 HS0 HS1,
  decompile L2 HS1 S1.

```

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 [I] 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 [I] L0 [I] 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)]

```

TODO
link
TODO
nuove
subst
TODO:
code
unif

```

697 lam x\ app[f, app[X, x]] = Y,
698 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):

```

705 lam x\ app[con"g", app[uv 0, x]]  $\approx_o$  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:

```

721 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

```

730 kind link type.
731 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):

```

740 lam x\ app[c"decision", app[c"nfact", x, c"3"]]  $\approx_o$ 
741 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:

```

747 lam x\ app[c"decision", app[c"nfact", x, c"3"]]  $\approx_\lambda$ 
748 lam x\ app [c"decision", uv 1 [x]]

```

The main difference is the replacement of the subterm $app[uv\ 0, x]$ of the OL with the subterm $uv\ 0\ [x]$. Variable indexes are chosen by the ML, that is, the index 0 for that unification variable of the OL term has not the sam meaning of the index 0 in the ML. There exists

two different substitution mapping, one for the OL and one for the ML and the indexes of variable point to the respective substitution.

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

```

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

```

6 USE OF MULTIVARS

Se il termine iniziale è della forma

```

763 app[con"xxx", (lam x\ lam y\ Y y x), (lam x\ f)]
764 =
765 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 η

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

```

774 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
- NOTA!! A dx abbiamo sempre un termine della forma lam
- ↪ x.VAR x!!!
- Altrimenti il link sarebbe stato risolto!!
- dopo l'unificazione rimane un link [variabile] -> [etaX]
- nella progress-eta, se a sx abbiamo una costante o
- ↪ un'app, allora eta-espandiamo
- di uno per poter unificare con il termine di dx.

6.2 Problems with β

β -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}_λ . 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 ρ such that ρt falls again in the \mathcal{L}_λ .

On the other hand, the \approx_λ is not designed to understand how the β -redexes work in the onject language. Therefore, even if we know that F is assigned to $\lambda x.x$, \approx_λ is not able to unify Fa with a . On the other hand, the problem $Fa = G$ is solvable by \approx_λ , 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 β -redex is replaced with a hole h and a new dedicated link, called $\text{link-}\beta$.

```
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 $\text{link-}\eta$, we will call h and t respectively the left lhs and the right rhs hand side of the $\text{link-}\beta$.

At compile time, a subterm is $\diamond\beta$, if it has the shape $\text{fapp}[f\text{uva } N \mid L]$ 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 $\text{append } PF \text{ } NPF \text{ } L$. The lhs is set to a new variable named M with PF in scope whereas the rhs is given by the term $\text{app}[f\text{uva } N' \text{ } PF \mid NPF]$ where the \mathcal{H}_0 variable identified by N' is mapped to the \mathcal{F}_0 variable named N .

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

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

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

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

The links generated from this unification problem are:

```
X ↦ X1; F ↦ X2 % The mappings
⊢ X0 =η= x\ `X3 x'
x ⊢ X3 x =β= X2 `X1 x' a
```

where the first link is a $\text{link-}\eta$ between the variable $X0$, representing the right side of the unification problem (it is a $\diamond\eta$) and $X3$; and a $\text{link-}\beta$ between the variable $X3$ and the subterm $c0 \setminus X2 \text{ } 'X1 \text{ } c0' \text{ } a$ (it is a $\diamond\beta$). The substitution tells that $x \mapsto X1 \text{ } x = x$.

We can now represent the hrun execution from this configuration which will, at first, dereference all the links, and then try to solve them. The only link being modified is the second one, which is set to $X3 \text{ } x =\beta= X2 \text{ } x \text{ } a$. The rhs of the link has now a variable which is partially in the PF , we can therefore remove the original $\text{link-}\beta$ and replace it with the following couple on links:

```
⊢ X1 =η= x\ `X4 x'
x ⊢ X3 x =β= x\ `X4 x' a
```

By these links we say that $X1$ is now η -linked to a fresh variable $X4$ with arity one. This new variable is used in the new $\text{link-}\beta$ where the name x is in its scope. This allows

6.3 Tricky examples

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

% @ok1 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: $f \text{ } 1 \text{ } 2 = x \text{ } 2$, by setting X to $f \text{ } 1$

TODO: We can re-use part of the algo for β 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, λ 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.
- [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. Pfenning. “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 Sch  rmann. “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 Benzm  ller. “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 λ 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, C  sar Mu  oz, and Sof  ene Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7.

APPENDIX

Note that (a infix b) c d de-sugars to (infix) a b c d.

11 THE OBJECT LANGUAGE

```

type (=o) ftm -> ftm -> o.                               (=o)
fapp A =o fapp B :- map (=o) A B.
flam F =o flam G :- pi x\ x =o x => F x =o G x.
fcon C =o fcon C.
fuva N =o fuva N.
flam F =o T :-                                           (ηl)
  pi x\ beta T [x] (R x), x =o x => F x =o R x.
T =o flam F :-                                           (ηr)
  pi x\ beta T [x] (R x), x =o x => R x =o F x.
fapp [flam X | L] =o T :- beta (flam X) L R, R =o T. (βl)
T =o fapp [flam X | L] :- beta (flam X) L R, T =o R. (βr)

type beta fm -> list fm -> fm -> o.
beta A [] A.
beta (flam F) [H | L] R :- subst F H B,
  beta B L R. % since F could be x\app[x|_] and H be lam _
beta (fapp A) L (fapp X) :- append A L X.
beta (fuva N) L (fapp [fuva N | L]).
beta (fcon H) L (fapp [fcon H | L]).

type subst (fm -> fm) -> fm -> fm -> o.
subst F H B :- napp (F H) B. % since (F H) may generate (app[app _ | _])

type napp fm -> fm -> o.
napp (fcon C) (fcon C).
napp (flam F) (flam G) :- pi x\ napp x x => napp (F x) (G x).
napp (fapp[fapp L|M]) R :- !, append L M N, napp (fapp N) R.
napp (fapp[X]) R :- !, napp X R.
napp (fapp A) (fapp B) :- map napp A B.
napp (fuva N) (fuva N).

```

12 THE META LANGUAGE

```

typeabbrev subst list (option assignment).

kind inctx type -> type.
type abs (tm -> inctx A) -> inctx A.
type val A -> inctx A.

typeabbrev assignment (inctx tm).

kind tm type.
type app list tm -> tm.
type lam (tm -> tm) -> tm.
type con string -> tm.
type uva address -> list tm -> tm.

type (==l) tm -> tm -> subst -> subst -> o.
% congruence
((app L1) ==l (app L2)) S S1 :- fold2 (==l) L1 L2 S S1.
((lam F1) ==l (lam F2)) S S1 :-
  pi x\ copy x x => ((F1 x) ==l (F2 x)) S S1.

```

```

((con X) ==l (con X)) S S.
% set variables
((uva N Args) ==l T) S S1 :-
  mem.set? N S F, !, move F Args T1, (T ==l T) S S1.
(T ==l (uva N Args)) S S1 :-
  mem.set? N S F, !, move F Args T1, (T ==l T1) S S1.
% flex-flex
((uva M A1) ==l (uva N A2)) S1 S2 :- !,
  pattern-fragment A1, pattern-fragment A2,
  prune! M A1 N A2 S1 S2.
((uva N Args) ==l T) S S1 :- not_occ N S T, pattern-fragment Args,
  bind T Args T1, mem.assign N S T1 S1.
% variable assignment
(T ==l (uva N Args)) S S1 :- not_occ N S T, pattern-fragment Args,
  bind T Args T1, mem.assign N S T1 S1.
(N ==l N) S S :- name N.

% Note: We suppose the scopes to always be in PF
type prune! address -> list ho.tm -> address ->
  list ho.tm -> subst -> subst -> o.
prune! N A N A S S :- !.
prune! M A N A S1 S2 :- !, bind (uva M A) A Ass,
  mem.assign N S1 Ass S2.
prune! N A1 N A2 S1 S3 :- !,
  std.assert!(len A1 {len A2}) "Not typechecking", !,
  mem.new S1 W S2, prune-same-variable W A1 A2 [] Ass,
  mem.assign N S2 Ass S3.
prune! N A1 M A2 S1 S4 :- !,
  mem.new S1 W S2, prune-diff-variables W A1 A2 Ass1 Ass2,
  mem.assign N S2 Ass1 S3,
  mem.assign M S3 Ass2 S4.

type prune-same-variable address -> list tm -> list tm ->
  list tm -> assignment -> o.
prune-same-variable N [] [] ACC (val (uva N Args)) :-
  std.rev ACC Args.
prune-same-variable N [X|XS] [X|YS] ACC (abs F) :-
  pi x\ prune-same-variable N XS YS [x|ACC] (F x).
prune-same-variable N [_|XS] [_|YS] ACC (abs F) :-
  pi x\ prune-same-variable N XS YS ACC (F x).

type prune-build-ass1 address -> list tm ->
  list bool -> assignment -> o.
prune-build-ass1 N Acc [] (val (uva N Args)) :-
  std.rev Acc Args.
prune-build-ass1 N Acc [tt|L] (abs T) :-
  pi x\ prune-build-ass1 N [x|Acc] L (T x).
prune-build-ass1 N Acc [ff|L] (abs T) :-
  pi x\ prune-build-ass1 N Acc L (T x).

type build-order list nat -> list tm -> list tm -> o.
build-order L T R :-
  len L Len, list-init Len z
  (p\r\ sigma Index Elt\ index L p Index, nth Index T r) R.

type prune-build-ass2 address -> list tm -> list bool ->
  list nat -> assignment -> o.

```

```

1161 prune-build-ass2 N Acc [] Pos (val (uva N Args)) :-
1162   std.rev Acc Acc', build-order Pos Acc' Args.
1163 prune-build-ass2 N Acc [tt|L] Pos (abs T) :-
1164   pi x\ prune-build-ass2 N [x|Acc] L Pos (T x).
1165 prune-build-ass2 N Acc [ff|L] Pos (abs T) :-
1166   pi x\ prune-build-ass2 N Acc L Pos (T x).
1167
1168 type keep list A -> A -> bool -> o.
1169 keep L A tt :- mem L A, !.
1170 keep _ _ ff.
1171
1172 type prune-diff-variables address -> list tm -> list tm ->
1173   assignment -> assignment -> o.
1174 prune-diff-variables N Args1 Args2 Ass1 Ass2 :-
1175   std.map Args1 (keep Args2) Bits1,
1176   prune-build-ass1 N [] Bits1 Ass1,
1177   std.map Args2 (keep Args1) Bits2,
1178   std.filter Args1 (mem Args2) ToKeep1,
1179   std.filter Args2 (mem Args1) ToKeep2,
1180   std.map ToKeep2 (index ToKeep1) Pos,
1181   prune-build-ass2 N [] Bits2 Pos Ass2.
1182
1183 type move assignment -> list tm -> tm -> o.
1184 move (abs Bo) [H|L] R :- move (Bo H) L R.
1185 move (val A) [] A :- !.
1186 move (val (uva N A)) L (uva N X) :- std.append A L X.
1187
1188 type beta tm -> list tm -> tm -> o.
1189 beta A [] A.
1190 beta (lam Bo) [H | L] R :- beta (Bo H) L R.
1191 beta (app A) L (app X) :- std.append A L X.
1192 beta (uva N A) L (uva N A') :- std.append A L A'.
1193 beta (con H) L (app [con H | L]).
1194
1195 type not_occ_aux address -> subst -> tm -> o.
1196 not_occ_aux N S (uva M _) :- mem.unset? M S, not (N = M).
1197 not_occ_aux N S (uva M Args) :- mem.set? M S F,
1198   move F Args T, not_occ_aux N S T.
1199 not_occ_aux N S (app L) :- forall1 (not_occ_aux N S) L.
1200 not_occ_aux N S (lam F) :- pi x\ not_occ_aux N S (F x).
1201 not_occ_aux _ _ (con _).
1202 not_occ_aux _ _ X :- name X.
1203
1204 type not_occ address -> subst -> tm -> o.
1205 not_occ N _ (uva N _).
1206 not_occ N S (uva M Args) :- mem.set? M S F,
1207   move F Args T, not_occ N S T.
1208 not_occ N S (uva M Args) :- mem.unset? M S,
1209   std.forall Args (not_occ_aux N S).
1210 not_occ _ _ (con _).
1211 not_occ N S (app L) :- not_occ_aux N S (app L).
1212 not_occ N S (lam L) :- pi x\ not_occ N S (L x).
1213 not_occ _ _ X :- name X.
1214
1215 type copy tm -> tm -> o.
1216 copy (app L) (app L') :- forall2 copy L L'.
1217 copy (lam T) (lam T') :- pi x\ copy x x => copy (T x) (T' x).
1218

```

```

1219 copy (uva N L) (uva N L') :- forall2 copy L L'.
1220 copy (con C) (con C).
1221 copy N N :- name N.
1222
1223 type bind tm -> list tm -> assignment -> o.
1224 bind T [] (val T') :- copy T T'.
1225 bind T [X | TL] (abs T') :- pi x\ copy X x => bind T TL (T' x).
1226
1227 type deref subst -> tm -> tm -> o.
1228 deref S X _ :- (var S; var X), halt "flex deref".
1229 deref H (uva N L) X :- mem.set? N H T,
1230   move T L X', !, deref H X' X.
1231 deref H (app L) (app L1) :- forall2 (deref H) L L1.
1232 deref _ (con X) (con X).
1233 deref H (uva X L) (uva X L1) :- mem.unset? X H,
1234   forall2 (deref H) L L1.
1235 deref H (lam F) (lam G) :- pi x\ deref H (F x) (G x).
1236 deref _ N N :- name N.
1237
1238 type deref-assmt subst -> assignment -> assignment -> o.
1239 deref-assmt S (abs T) (abs R) :- pi x\ deref-assmt S (T x) (R x).
1240 deref-assmt S (val T) (val R) :- deref S T R.
1241

```

13 THE COMPILER

14 THE PROGRESS FUNCTION

15 THE DECOMPILER

16 AUXILIARY FUNCTIONS

```

1247 type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A -> list A1 -> B
1248 fold4 _ [] [] A A B B.
1249 fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0, fold4 F XS YS A0 A1
1250
1251 type len list A -> nat -> o.
1252 len [] z.
1253 len [_|L] (s X) :- len L X.
1254

```