

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, 16 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 $\text{fin } n$, of natural numbers smaller than n is finite; 2) the predicate $\text{nfact } n \text{ nf}$, 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 $\llbracket x \backslash e \rrbracket$, and square brackets denote a list of terms separated by comma. As an example we show the encoding of the Coq term $\llbracket \forall y: t, \text{nfact } y \text{ 3} \rrbracket$:

```
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) 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*, *if* $\mathcal{T} \subseteq \mathcal{L}_\lambda$ *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 .

XXX permuting *hrun* does not change the final result if check does not fail eagerly

XXX if we want to apply heuristics, we can apply them in *decomp* to avoid committing to a non MGU too early

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).

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

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

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

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

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. 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 :

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

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 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:

```
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 $\text{fapp}[\text{fuva}\ N, x]$, where N is a memory address and x is a bound variable.

In \mathcal{H}_0 the representation of $P\ x$ is instead $\text{uva}\ N\ [x]$. We say that the unification variable $\text{uva}\ N\ L$ is in \mathcal{L}_λ 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.² The compiler ?? needs to support terms outside \mathcal{L}_λ for practical reasons, so we don’t assume all out terms are in \mathcal{L}_λ 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:

```
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 Notations

We use math mode for \mathcal{H}_0 .

```
λx.λy.Fxy   lam x\ lam y\ uva F [x, y]
f a          app[con "f", con "a"]
λx.Fx a     lam x\ app[uva F [x], con "a"]
λx.Fx x     lam x\ app[uva F [x], x]
```

4.2 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: ρ s and σ 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 $(\text{app} [\text{app} [\text{con} "f", \text{con} "a"], \text{con} "b"])$ into $(\text{app} [\text{con} "f", \text{con} "a", \text{con} "b"])$.

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:

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


```

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

```

472 type fder fsubst -> fm -> fm -> o.
473 fder S (fapp A) (fapp B) :- map (fder S) A B.
474 fder S (flam F) (flam G) :-
475   pi x\ fder S x x => fder S (F x) (G x).
476 fder _ (fcon C) (fcon C).
477 fder S (fuva N) R :- set? N S T, fder S T R.
478 fder S (fuva N) (fuva N) :- unset? N S.
479
480 type fderef fsubst -> fm -> fm -> o. (ρs)
481 fderef S T R :- fder S T T', napp T' R.

```

```

483 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}_o is similar, we only show the last two rules that differ in a substantial way:

```

490 type deref subst -> tm -> tm -> o. (σt)
491 deref S (app A) (app B) :- map (deref S) A B.
492 deref S (lam F) (lam G) :-
493   pi x\ deref S x x => deref S (F x) (G x).
494 deref _ (con C) (con C).
495 deref S (uva N A) R :- set? N S F, move F A T, deref S T R.
496 deref S (uva N A) (uva N B) :- unset? N S, map (deref S) A B.
497 type move assignment -> list tm -> tm -> o.
498 move (abs Bo) [H|L] R :- move (Bo H) L R.
499 move (val A) [] A :- !.
500 move (val (uva N A)) L (uva N X) :- 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

```

510 type (=o) ftm -> ftm -> o. (=o)
511 fapp A =o fapp B :- map (=o) A B.
512 flam F =o flam G :- pi x\ x =o x => F x =o G x.
513 fcon C =o fcon C.
514 fuva N =o fuva N.
515 flam F =o T :- (ηl)
516   pi x\ beta T [x] (R x), x =o x => F x =o R x.
517 T =o flam F :- (ηr)
518   pi x\ beta T [x] (R x), x =o x => R x =o F x.
519 fapp [flam X | L] =o T :- beta (flam X) L R, R =o T. (βl)
520 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 \approx_λ 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 ??.

```

548 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}_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.

types

```
kind arity type.
type arity nat -> arity.
```

```
kind fvariable type.
type fv address -> fvariable.
```

```
kind hvariable type.
type hv address -> arity -> hvariable.
```

```
kind mapping type.
type mapping fvariable -> hvariable -> mapping.
typeabbrev map (list mapping).
```

```
typeabbrev scope (list tm).
```

core

```
type comp fm -> tm -> map -> map -> links -> links ->
  subst -> subst -> o.
comp (fcon C) (con C)      M1 M1 L1 L1 S1 S1.
comp (flam F) (lam F1)     M1 M2 L1 L2 S1 S2 :-
  comp-lam F F1 M1 M2 L1 L2 S1 S2.
comp (fuva A) (uva B [])   M1 M2 L L S1 S1 :-
  alloc-mapping M1 M2 (fv A) (hv B (arity z)) S S1.
comp (fapp [fuva A|Scope]) (uva B Scope1) M1 M2 L1 L1 S1 S2 :-
  pattern-fragment Scope, !,
```

```
  fold6 comp Scope Scope1 M1 M1 L1 L1 S1 S1,
  len Scope Arity,
  alloc-mapping M1 M2 (fv A) (hv B (arity Arity)) S1 S2.
comp (fapp A) (app A1)     M1 M2 L1 L2 S1 S2 :-
  fold6 comp A A1 M1 M2 L1 L2 S1 S2.
```

aux

```
type comp-lam (fm -> fm) -> (tm -> tm) ->
  map -> map -> links -> links -> subst -> subst -> o.
comp-lam F F1 M1 M2 L L2 S S1 :-
  pi x y\ (pi M L S\ comp x y M M L L S S) =>
  comp (F x) (F1 y) M1 M2 L (L1 y) S S1,
  close-links L1 L2.
```

```
type close-links (tm -> links) -> links -> o.
close-links (\[]) [].
```

```
close-links (v\ [L|XS v]) [L|YS] :- !, close-links XS YS.
close-links (v\ (L v)|XS v) [ho.abs L|YS] :- !,
  close-links XS YS.
```

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

```
type decomp1 links -> subst -> fsubst -> o.
decomp1 S [] [].
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 [] S0 [] L0,
  comp Y0 Y1 S0 S1 L0 L1,
  (X1 ≈λ Y1) [] 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.
```

```

697 type prolog-fo list fm -> list (pair nat nat) -> subst -> o.
698 prolog-fo Terms Problems S :-
699   map (pick Terms) Problems FoProblems,
700   fold4 ( $\approx_o$ ) FoProblems [] S.
701
702 type step-ho (pair tm tm) -> links -> links -> subst -> subst -> o.
703 step-ho (pr X Y) L0 L1 S0 S2 :-
704   (X1  $\approx_\lambda$  Y1) S0 S1,
705   solve-links L0 L1 S1 S2.
706
707 type prolog-ho list fm -> list (pair nat nat) -> subst -> o.
708 prolog-ho Terms Problems S :-
709   fold4 comp Terms HoTerms [] L0 [] HS0,
710   map (pick HoTerms) Problems HoProblems,
711   fold4 step-ho HoProblems L0 L HS0 HS,
712   decompile L HS S.
713
714 the proprty is that if a step for Fo succeeds then the Ho one does,
715 and if Fo fails then the Ho fails ()
716
717 5.2 Example
718 OK
719
720 Terms [ flam x\ fapp[fcon"g",fapp[fuva z, x]]
721         , flam x\ fapp[fcon"g", fcon"a"] ]
722 Problems = [ pr z (s z) ] %  $\lambda x.g(Fx) = \lambda x.ga$ 
723 lam x\ app[con"g",uva z [x]]  $\approx_o$  lam x\ app[con"g", con"a"]
724 link z z (s z)
725 HS = [some (abs x\con"a")]
726 S = [some (flam x\fcon a)]
727 KO
728
729 Terms [ flam x\ fapp[fcon"g",fapp[fuva z, x]]
730         , flam x\ fapp[fcon"g", fcon"a"] ]
731 Problems = [ pr 0 1 %  $A = \lambda x.x$ 
732               , pr 2 3 ] %  $Aa = a$ 
733 lam x\ app[con"g",uva z [x]]  $\approx_o$  lam x\ app[con"g", con"a"]
734 link z z (s z)
735 HS = [some (abs x\con"a")]
736 S = [some (flam x\fcon a)]
737 lam x\ app[f, app[X, x]] = Y,
738 lam x\ x[] = X.

```

the proprty 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

```

720 Terms [ flam x\ fapp[fcon"g",fapp[fuva z, x]]
721         , flam x\ fapp[fcon"g", fcon"a"] ]
722 Problems = [ pr z (s z) ] %  $\lambda x.g(Fx) = \lambda x.ga$ 
723 lam x\ app[con"g",uva z [x]]  $\approx_o$  lam x\ app[con"g", con"a"]
724 link z z (s z)
725 HS = [some (abs x\con"a")]
726 S = [some (flam x\fcon a)]
727 KO
728
729 Terms [ flam x\ fapp[fcon"g",fapp[fuva z, x]]
730         , flam x\ fapp[fcon"g", fcon"a"] ]
731 Problems = [ pr 0 1 %  $A = \lambda x.x$ 
732               , pr 2 3 ] %  $Aa = a$ 
733 lam x\ app[con"g",uva z [x]]  $\approx_o$  lam x\ app[con"g", con"a"]
734 link z z (s z)
735 HS = [some (abs x\con"a")]
736 S = [some (flam x\fcon a)]
737 lam x\ app[f, app[X, x]] = Y,
738 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]] \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:

```
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"]]  $\approx_o$ 
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"]]  $\approx_\lambda$ 
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:**

```
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)

integer
or
nat?

6.1 Problems with η

```

comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
  (pi x\ maybe-eta x (F x) [x]), !,
  alloc S1 A S2,
  comp-lam F F1 M1 M2 L1 L2 S2 S3,
  get-scope (lam F1) Scope,
  L3 = [eval-link-eta (uva A Scope) (lam F1) | L2].

and aux

%% x occurs rigidly in t iff  $\forall \sigma, \forall t', t' =_{\sigma} \sigma t \Rightarrow x \in \mathcal{P}(t')$ 
%%
type occurs-rigidly fm -> fm -> o.
occurs-rigidly N N.
occurs-rigidly _ (fapp [fuva _|_] ) :- !, fail.
occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.
occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).

/* maybe-eta N T L succeeds iff T could be an eta expansions for N.
%% is  $\exists \sigma, \sigma(\lambda n.t) = \lambda n.t'n$  and n
%% does not occur rigidly in t'
type maybe-eta fm -> fm -> list fm -> o.
maybe-eta N (fapp[fuva _|Args] ) _ :- !,
  exists (x\ maybe-eta-of [ ] N x) Args, !.
maybe-eta N (flam B) L :- !, pi x\ maybe-eta N (B x) [x | L].
maybe-eta _ (fapp [fcon _|Args] ) L :-
  split-last-n {len L} Args First Last,
  forall1 (x\ forall1 (y\ not (occurs-rigidly x y)) First) L,
  forall12 (maybe-eta-of [ ]) {rev L} Last.

%% is  $\exists \sigma, \sigma t =_{\sigma} n$ 
type maybe-eta-of list fm -> fm -> fm -> o.
maybe-eta-of _ N N :- !.
maybe-eta-of L N (fapp[fuva _|Args] ) :- !,
  forall1 (x\ exists (maybe-eta-of [ ] x) Args) [N|L].
maybe-eta-of L N (flam B) :- !,
  pi x\ maybe-eta-of [x | L] N (B x).
maybe-eta-of L N (fapp [N|Args] ) :-
  last-n {len L} Args R,
  forall12 (maybe-eta-of [ ]) R {rev L}.

```

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

$X = \text{lam } x \backslash \text{lam } y \backslash Y \ y \ x, X = \text{lam } x \backslash f$

TODO: The snd unif pb, we have to unif $\text{lam } x \backslash \text{lam } y \backslash Y \ x \ y$ with $\text{lam } x \backslash f$

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

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

```

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 $t = X t_1 \dots t_n$, where X is flexible and the list $[t_1 \dots t_n]$ is 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 object 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 does not generate terms outside \mathcal{W} (Property 2.8).

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 link- β .

```
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- η , we will call h and t respectively the left hand side (lhs) and the right hand side (rhs) of the link- β .

In order to build a link- β , we need to adapt the compiler so that it can recognize these “problematic” subterms. The following code snippet illustrate such behavior, we suppose the rule to be added just after ??.

```

comp (fapp [fuva A|Ag]) (uva C Scope) M1 M3 L1 L3 S1 S4 :- !,
  pattern-fragment-prefix Ag Pf Extra,
  fold6 comp Pf Scope1 M1 M1 L1 L1 S1 S1,
  fold6 comp Extra Extra1 M1 M2 L1 L2 S1 S2,
  len Pf Arity,
  alloc-mapping M2 M3 (fv A) (hv B (arity Arity)) S2 S3,
  Beta = app [uva B Scope1 | Extra1],
  get-scope Beta Scope,
  alloc S3 C S4,
  L3 = [eval-link-beta (uva C Scope) Beta | L2].

```

A term is $\diamond \beta$ if it has the shape $\text{fapp}[fuva A|Ag]$ and distinct Ag does not hold. In that case, Ag is split in two sublist Pf and $Extra$ such that former is the longest prefix of Ag such that distinct Pf holds. $Extra$ is the list such that append $Pf \ Extra \ Ag$. Next important step is to compile recursively the terms of these lists and allocate a memory adress B from the substitution in order to map the \mathcal{F}_0 variable $fuva \ A$ to the \mathcal{H}_0 variable $uva \ B$. The link- β to return in the end is given by the term $Beta = \text{app}[uva \ B \ Scope1 \ | \ Extra1]$ constituting the rhs , and a fresh variable C having in scope all the free variables occurring in $Beta$ (this is lhs). We point out that the rhs is intentionally built as an uva where $Extra1$ are not in scope, since by invariant, we want all the variables appearing in \mathcal{H}_0 to be in \mathcal{L}_λ .

One created, there exist two main situations waking up a suspended link- β . The former is strictly connected to the definition of β -redex and occurs when the head of rhs is materialized by the oracle (see eq. (5)). In this case rhs is safely β -reduced to a new term t' and the result can be unified with lhs . In this scenario the link- β has accomplished its goal and can be removed from \mathbb{L} .

The second circumstance making the link- β to progress is the instantiation of the variables in the `Extra1` making the corresponding arguments to reduce to names. In this case, we want to take the list `Scope1` and append to it the largest prefix of `Extra1` in a new variable `Scope2` such that `Scope2` remains in \mathcal{L}_λ ; we call `Extra2` the suffix of `Extra1` such that the concatenation of `Scope1` and `Extra1` is the same as the concatenation of `Scope2` and `Extra2`.

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

$$f = \text{flam } x \backslash \text{fapp}[F, (X \ x), a] \quad \% f = \lambda x. F(Xx)a$$

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

The links generated from this unification problem are:

$$X \mapsto X1^1; F \mapsto X2^0 (\text{Themap}) \quad (6)$$

$$\vdash X0 =_\eta \lambda x.X3_x \quad (7)$$

$$x \vdash X3_x =_\beta X2 \ X1_x \ a \quad (8)$$

where the first link is a link- η between the variable $X0$, representing the right side of the unification problem (it is a $\diamond\eta$) and $X3$; and a link- β between the variable $X3$ and the subterm $\lambda x.X1_x \ a$ (it is a $\diamond\beta$). The substitution tells that $x \vdash X1_x = x$.

We can now represent the hrn 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 $x \vdash X3 =_\beta X2x \ a$. The rhs of the link has now a variable which is partially in the PF, we can therefore remove the original link- β and replace it with the following couple on links:

$$\begin{aligned} \vdash X1 &=_{\eta} x \backslash \text{`X4 } x' \\ x \vdash X3 \ x &=_{\beta} x \backslash \text{`X4 } x' \ a \end{aligned}$$

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 link- β 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: $f \ 1 \ 2 = x \ 2$, by setting X to $f \ 1$

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*. LFMTP '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 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 Benzmüller. “Theorem Proving in Dependently-Typed Higher-Order Logic”. In: *Automated Deduction — CADE 29*. Ed. by Brigitte Pientka and Cesare Tinelli. Cham: Springer Nature Switzerland, 2023, pp. 438–455. ISBN: 978-3-031-38499-8.
- [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, 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 $(\text{infix}) \text{ a b c d}$.

Explain builtin name (can be implemented by loading name after each pi)

11 THE MEMORY

kind address type.

type addr nat \rightarrow address.

typeabbrev (mem A) (list (option A)).

type get nat \rightarrow mem A \rightarrow A \rightarrow o.

get z (some Y :: _) Y.

get (s N) (_ :: L) X :- get N L X.

type alloc-aux nat \rightarrow mem A \rightarrow mem A \rightarrow o.

alloc-aux z [] [none] :- !.

alloc-aux z L L.

alloc-aux (s N) [] [none | M] :- alloc-aux N [] M.

alloc-aux (s N) [X | L] [X | M] :- alloc-aux N L M.

type alloc address \rightarrow mem A \rightarrow mem A \rightarrow o.

alloc (addr A as Ad) Mem1 Mem2 :- unset? Ad Mem1,

alloc-aux A Mem1 Mem2.

type new-aux mem A \rightarrow nat \rightarrow mem A \rightarrow o.

new-aux [] z [none].

new-aux [A | As] (s N) [A | Bs] :- new-aux As N Bs.

type new mem A \rightarrow address \rightarrow mem A \rightarrow o.

new Mem1 (addr Ad) Mem2 :- new-aux Mem1 Ad Mem2.

type set? address \rightarrow mem A \rightarrow A \rightarrow o.

set? (addr A) Mem Val :- get A Mem Val.

type unset? address \rightarrow mem A \rightarrow o.

unset? Addr Mem :- not (set? Addr Mem _).

type assign-aux nat \rightarrow mem A \rightarrow A \rightarrow mem A \rightarrow o.

assign-aux z (none :: L) Y (some Y :: L).

assign-aux (s N) (X :: L) Y (X :: L1) :- assign-aux N L Y L1.

type assign address \rightarrow mem A \rightarrow A \rightarrow mem A \rightarrow o.

assign (addr A) Mem1 Val Mem2 :- assign-aux A Mem1 Val Mem2.

12 THE OBJECT LANGUAGE

kind fm type.

type fapp list fm \rightarrow fm.

type flam (fm \rightarrow fm) \rightarrow fm.

type fcon string \rightarrow fm.

type fuva address \rightarrow fm.

typeabbrev subst mem fm.

type fder subst \rightarrow fm \rightarrow fm \rightarrow o.

fder S (fuva N) T1 :- set? N S T, fder S T T1.

%fder S (fapp [fuva N|L]) R :- set? N S T, !, beta T L R', fder S R' R.

fder S (fapp L1) (fapp L2) :- forall2 (fder S) L1 L2.

fder S (flam F1) (flam F2) :-

pi x\ fder S x x => fder S (F1 x) (F2 x).

fder _ (fcon X) (fcon X).

fder _ (fuva N) (fuva N).

%fder _ N N :- name N.

type napp fm \rightarrow fm \rightarrow o.

napp (fcon C) (fcon C).

napp (fuva A) (fuva A).

napp (flam F) (flam F1) :- pi x\ napp x x => napp (F x) (F1 x).

napp (fapp [fapp L1 | L2]) T :- !,

append L1 L2 L3, napp (fapp L3) T.

napp (fapp L) (fapp L1) :- forall2 napp L L1.

type fderef subst \rightarrow fm \rightarrow fm \rightarrow o.

fderef S T T2 :- fder S T T1, napp T1 T2.

type (=o) fm \rightarrow fm \rightarrow o.

fapp L1 =o fapp L2 :- forall2 (=o) L1 L2.

flam F1 =o flam F2 :- pi x\ x =o x => F1 x =o F2 x.

fcon X =o fcon X.

fuva N =o fuva N.

flam F =o T :- pi x\ beta T [x] (T' x), x =o x => F x =o T' x.

T =o flam F :- pi x\ beta T [x] (T' x), x =o x => T' x =o F x.

fapp [flam X | TL] =o T :- beta (flam X) TL T', T' =o T.

T =o fapp [flam X | TL] :- beta (flam X) TL T', T =o T'.

type extend-subst fm \rightarrow subst \rightarrow subst \rightarrow o.

extend-subst (fuva N) S S' :- mem.alloc N S S'.

extend-subst (flam F) S S' :-

pi x\ (pi S\ extend-subst x S S) => extend-subst (F x) S S'.

extend-subst (fcon _) S S.

extend-subst (fapp L) S S1 :- fold extend-subst L S S1.

type beta fm \rightarrow list fm \rightarrow fm \rightarrow o.

beta A [] A.

beta (flam Bo) [H | L] R :- beta (Bo H) L R.

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]).

beta N L (fapp [N | L]) :- name N.

type mk-app fm \rightarrow list fm \rightarrow fm \rightarrow o.

mk-app T L S :- beta T L S.

type eta-contract fm \rightarrow fm \rightarrow o.

eta-contract (fcon X) (fcon X).

eta-contract (fapp L) (fapp L1) :- forall2 eta-contract L L1.

eta-contract (flam F) T :- eta-contract-aux [] (flam F) T.

eta-contract (flam F) (flam F1) :-

pi x\ eta-contract x x => eta-contract (F x) (F1 x).

eta-contract (fuva X) (fuva X).

eta-contract X X :- name X.

```

1277 type eta-contract-aux list fm -> fm -> fm -> o.
1278 eta-contract-aux L (flam F) T :-
1279   pi x\ eta-contract-aux [x|L] (F x) T. % also checks H Prefix does not ACC Args.
1280 eta-contract-aux L (fapp [H|Args]) T :-
1281   rev L LRev, append Prefix LRev Args,
1282   if (Prefix = []) (T = H) (T = fapp [H|Prefix]).
1283

```

13 THE META LANGUAGE

```

1284 typeabbrev subst list (option assignment).
1285
1286 kind inctx type -> type.
1287 type abs (tm -> inctx A) -> inctx A.
1288 type val A -> inctx A.
1289
1290 typeabbrev assignment (inctx tm).
1291
1292 kind tm type.
1293 type app list tm -> tm.
1294 type lam (tm -> tm) -> tm.
1295 type con string -> tm.
1296 type uva address -> list tm -> tm.
1297
1298 type (≈λ) tm -> tm -> subst -> subst -> o.
1299 (con C ≈λ con C) S S.
1300 (app L1 ≈λ app L2) S S1 :- fold2 (≈λ) L1 L2 S S1.
1301 (lam F1 ≈λ lam F2) S S1 :-
1302   pi x\ (pi S\ (x ≈λ x) S S) => (F1 x ≈λ F2 x) S S1.
1303 (uva N Args ≈λ T) S S1 :-
1304   set? N S F,!, move F Args T1, (T1 ≈λ T) S S1.
1305 (T ≈λ uva N Args) S S1 :-
1306   set? N S F,!, move F Args T1, (T ≈λ T1) S S1.
1307 (uva M A1 ≈λ uva N A2) S1 S2 :- !,
1308   pattern-fragment A1, pattern-fragment A2,
1309   prune! M A1 N A2 S1 S2.
1310 (uva N Args ≈λ T) S S1 :- not_occ N S T, pattern-fragment Args,
1311   bind T Args T1, assign N S T1 S1.
1312 (T ≈λ uva N Args) S S1 :- not_occ N S T, pattern-fragment Args,
1313   bind T Args T1, assign N S T1 S1.
1314
1315 type prune! address -> list tm -> address ->
1316   list tm -> subst -> subst -> o.
1317
1318 /* no pruning needed */
1319 prune! N A N A S S :- !.
1320 prune! M A N A S1 S2 :- !, bind (uva M A) A Ass,
1321   assign N S1 Ass S2.
1322 /* prune different arguments */
1323 prune! N A1 N A2 S1 S3 :- !,
1324   new S1 W S2, prune-same-variable W A1 A2 [] Ass,
1325   assign N S2 Ass S3.
1326 /* prune to the intersection of scopes */
1327 prune! N A1 M A2 S1 S4 :- !,
1328   new S1 W S2, prune-diff-variables W A1 A2 Ass1 Ass2,
1329   assign N S2 Ass1 S3,
1330   assign M S3 Ass2 S4.
1331
1332 type prune-same-variable address -> list tm -> list tm ->

```

```

1333   list tm -> assignment -> o.
1334 prune-same-variable N [] [] ACC (val (uva N Args)) :-
1335   rev ACC Args.
1336 prune-same-variable N [X|XS] [X|YS] ACC (abs F) :-
1337   pi x\ prune-same-variable N XS YS [x|ACC] (F x).
1338 prune-same-variable N [_|XS] [_|YS] ACC (abs F) :-
1339   pi x\ prune-same-variable N XS YS ACC (F x).
1340
1341 type permute list nat -> list tm -> list tm -> o.
1342 permute [] _ [].
1343 permute [P|PS] Args [T|TS] :-
1344   nth P Args T,
1345   permute PS Args TS.
1346
1347 type build-perm-assign address -> list tm -> list bool ->
1348   list nat -> assignment -> o.
1349 build-perm-assign N ArgsR [] Perm (val (uva N PermutedArgs)) :-
1350   rev ArgsR Args, permute Perm Args PermutedArgs.
1351 build-perm-assign N Acc [tt|L] Perm (abs T) :-
1352   pi x\ build-perm-assign N [x|Acc] L Perm (T x).
1353 build-perm-assign N Acc [ff|L] Perm (abs T) :-
1354   pi x\ build-perm-assign N Acc L Perm (T x).
1355
1356 type keep list A -> A -> bool -> o.
1357 keep L A tt :- mem L A, !.
1358 keep _ _ ff.
1359
1360 type prune-diff-variables address -> list tm -> list tm ->
1361   assignment -> assignment -> o.
1362 prune-diff-variables N Args1 Args2 Ass1 Ass2 :-
1363   forall2 (keep Args2) Args1 Bits1,
1364   forall2 (keep Args1) Args2 Bits2,
1365   filter Args1 (mem Args2) ToKeep1,
1366   filter Args2 (mem Args1) ToKeep2,
1367   forall2 (index ToKeep1) ToKeep1 IdPerm,
1368   forall2 (index ToKeep1) ToKeep2 Perm21,
1369   build-perm-assign N [] Bits1 IdPerm Ass1,
1370   build-perm-assign N [] Bits2 Perm21 Ass2.
1371
1372 type move assignment -> list tm -> tm -> o.
1373 move (abs Bo) [H|L] R :- move (Bo H) L R.
1374 move (val A) [] A :- !.
1375
1376 type beta tm -> list tm -> tm -> o.
1377 beta A [] A.
1378 beta (lam Bo) [H | L] R :- beta (Bo H) L R.
1379 beta (app A) L (app X) :- append A L X.
1380 beta (con H) L (app [con H | L]).
1381 beta X L (app[X|L]) :- name X.
1382
1383 /* occur check for N before crossing a functor */
1384 type not_occ address -> subst -> tm -> o.
1385 not_occ N S (uva M Args) :- set? M S F,
1386   move F Args T, not_occ N S T.
1387 not_occ N S (uva M Args) :- unset? M S, not (M = N),
1388   forall1 (not_occ_aux N S) Args.
1389 not_occ _ _ (con _).

```



```

1393 not_occ N S (app L) :- not_occ_aux N S (app L).
1394 /* Note: lam is a functor for the meta language! */
1395 not_occ N S (lam L) :- pi x\ not_occ_aux N S (L x).
1396 not_occ _ _ X :- name X.
1397 /* finding N is ok */
1398 not_occ N _ (uva N _).
1399
1400 /* occur check for X after crossing a functor */
1401 type not_occ_aux address -> subst -> tm -> o.
1402 not_occ_aux N S (uva M _) :- unset? M S, not (N = M).
1403 not_occ_aux N S (uva M Args) :- set? M S F,
1404   move F Args T, not_occ_aux N S T.
1405 not_occ_aux N S (app L) :- forall1 (not_occ_aux N S) L.
1406 not_occ_aux N S (lam F) :- pi x\ not_occ_aux N S (F x).
1407 not_occ_aux _ _ (con _).
1408 not_occ_aux _ _ X :- name X.
1409 /* finding N is ko, hence no rule */
1410
1411 /* copy T T' vails if T contains a free variable, i.e. it
1412   performs scope checking for bind */
1413 type copy tm -> tm -> o.
1414 copy (con C) (con C).
1415 copy (app L) (app L') :- forall2 copy L L'.
1416 copy (lam T) (lam T') :- pi x\ copy x x => copy (T x) (T' x).
1417 copy (uva A L) (uva A L') :- forall2 copy L L'.
1418
1419 type bind tm -> list tm -> assignment -> o.
1420 bind T [] (val T') :- copy T T'.
1421 bind T [X | TL] (abs T') :- pi x\ copy X x => bind T TL (T' x).
1422
1423 type deref subst -> tm -> tm -> o.
1424 deref H (uva N L) X :- set? N H T,
1425   move T L X', deref H X' X.
1426 deref H (app L) (app L1) :- forall2 (deref H) L L1.
1427 deref _ (con X) (con X).
1428 deref H (uva X L) (uva X L1) :- unset? X H,
1429   forall2 (deref H) L L1.
1430 deref H (lam F) (lam G) :- pi x\ deref H (F x) (G x).
1431 deref _ N N :- name N.
1432
1433 type deref-assmt subst -> assignment -> assignment -> o.
1434 deref-assmt S (abs T) (abs R) :- pi x\ deref-assmt S (T x) (R x).
1435 deref-assmt S (val T) (val R) :- deref S T R.
1436

```

14 THE COMPILER

```

1439 kind arity type.
1440 type arity nat -> arity.
1441
1442 kind fvariable type.
1443 type fv address -> fvariable.
1444
1445 kind hvariable type.
1446 type hv address -> arity -> hvariable.
1447
1448 kind mapping type.
1449 type mapping fvariable -> hvariable -> mapping.
1450

```

```

1451 typeabbrev map (list mapping).
1452
1453 typeabbrev scope (list tm).
1454
1455 kind linkctx type.
1456 type link-eta tm -> tm -> linkctx.
1457 type link-beta tm -> tm -> linkctx.
1458
1459 macro @val-link-eta T1 T2 :- ho.val (link-eta T1 T2).
1460 macro @val-link-beta T1 T2 :- ho.val (link-beta T1 T2).
1461
1462 typeabbrev link (ho.inctx linkctx).
1463
1464 typeabbrev links (list link).
1465
1466 %% x occurs rigidly in t iff  $\forall \sigma, \forall t', t' =_o \sigma t \Rightarrow x \in \mathcal{P}(t')$ 
1467 %%
1468 type occurs-rigidly fm -> fm -> o.
1469 occurs-rigidly N N.
1470 occurs-rigidly _ (fapp [fuva _|_]) :- !, fail.
1471 occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.
1472 occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).
1473
1474 /* maybe-eta N T L succeeds iff T could be an eta expansions for N, that
1475   is  $\exists \sigma, \sigma(\lambda n.t) = \lambda n.t'n$  and n
1476   does not occur rigidly in t'
1477 type maybe-eta fm -> fm -> list fm -> o.
1478 maybe-eta N (fapp [fuva _|Args]) _ :- !,
1479   exists (x\ maybe-eta-of [] N x) Args, !.
1480 maybe-eta N (flam B) L :- !, pi x\ maybe-eta N (B x) [x | L].
1481 maybe-eta _ (fapp [fcon _|Args]) L :-
1482   split-last-n {len L} Args First Last,
1483   forall1 (x\ forall1 (y\ not (occurs-rigidly x y)) First) L,
1484   forall2 (maybe-eta-of []) {rev L} Last.
1485
1486 %% is  $\exists \sigma, \sigma t =_o n$ 
1487 type maybe-eta-of list fm -> fm -> fm -> o.
1488 maybe-eta-of _ N N :- !.
1489 maybe-eta-of L N (fapp [fuva _|Args]) :- !,
1490   forall1 (x\ exists (maybe-eta-of [] x) Args) [N|L].
1491 maybe-eta-of L N (flam B) :- !,
1492   pi x\ maybe-eta-of [x | L] N (B x).
1493 maybe-eta-of L N (fapp [N|Args]) :-
1494   last-n {len L} Args R,
1495   forall2 (maybe-eta-of []) R {rev L}.
1496
1497 type locally-bound tm -> o.
1498 type get-scope-aux tm -> list tm -> o.
1499 get-scope-aux (con _) [].
1500 get-scope-aux (uva _ L) L1 :-
1501   forall2 get-scope-aux L R,
1502   flatten R L1.
1503 get-scope-aux (lam B) L1 :-
1504   pi x\ locally-bound x => get-scope-aux (B x) L1.
1505 get-scope-aux (app L) L1 :-
1506   forall2 get-scope-aux L R,
1507

```

```

1509     flatten R L1.
1510 get-scope-aux X [X] :- name X, not (locally-bound X).
1511 get-scope-aux X [] :- name X, (locally-bound X).
1512
1513 %% TODO: scrivere undup
1514 type get-scope tm -> list tm -> o.
1515 get-scope T Scope :-
1516   get-scope-aux T ScopeDuplicata,
1517   names N, filter N (mem ScopeDuplicata) Scope.
1518 type comp-lam (fm -> fm) -> (tm -> tm) ->
1519   map -> map -> links -> links -> subst -> subst -> o.
1520 comp-lam F F1 M1 M2 L L2 S S1 :-
1521   pi x y\ (pi M L S\ comp x y M M L L S S) =>
1522     comp (F x) (F1 y) M1 M2 L (L1 y) S S1,
1523     close-links L1 L2.
1524
1525 type close-links (tm -> links) -> links -> o.
1526 close-links (_\[]) [].
1527 close-links (v\ [L|XS v]) [L|YS] :- !, close-links XS YS.
1528 close-links (v\ (L v)|XS v) [ho.abs L|YS] :- !,
1529   close-links XS YS.
1530 type comp fm -> tm -> map -> map -> links -> links ->
1531   subst -> subst -> o.
1532 comp (fcon C) (con C) M1 M1 L1 L1 S1 S1.
1533 comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
1534   (pi x\ maybe-eta x (F x) [x]), !,
1535   alloc S1 A S2,
1536   comp-lam F F1 M1 M2 L1 L2 S2 S3,
1537   get-scope (lam F1) Scope,
1538   L3 = [eval-link-eta (uva A Scope) (lam F1)| L2].
1539 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :-
1540   comp-lam F F1 M1 M2 L1 L2 S1 S2.
1541 comp (fuva A) (uva B []) M1 M2 L L S S1 :-
1542   alloc-mapping M1 M2 (fv A) (hv B (arity z)) S S1.
1543 comp (fapp [fuva A|Scope]) (uva B Scope1) M1 M2 L1 L1 S1 S2 :-
1544   pattern-fragment Scope, !,
1545   fold6 comp Scope Scope1 M1 M1 L1 L1 S1 S1,
1546   len Scope Arity,
1547   alloc-mapping M1 M2 (fv A) (hv B (arity Arity)) S1 S2.
1548 comp (fapp [fuva A|Ag]) (uva C Scope) M1 M3 L1 L3 S1 S4 :- !,
1549   pattern-fragment-prefix Ag Pf Extra,
1550   fold6 comp Pf Scope1 M1 M1 L1 L1 S1 S1,
1551   fold6 comp Extra Extra1 M1 M2 L1 L2 S1 S2,
1552   len Pf Arity,
1553   alloc-mapping M2 M3 (fv A) (hv B (arity Arity)) S2 S3,
1554   Beta = app [uva B Scope1 | Extra1],
1555   get-scope Beta Scope,
1556   alloc S3 C S4,
1557   L3 = [eval-link-beta (uva C Scope) Beta | L2].
1558 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :-
1559   fold6 comp A A1 M1 M2 L1 L2 S1 S2.
1560
1561 type alloc mem A -> address -> mem A -> o.
1562 alloc S N S1 :- mem.new S N S1.
1563
1564 type compile-terms-diagnostic
1565   triple diagnostic fm fm ->
1566
1567   triple diagnostic tm tm ->
1568   map -> map ->
1569   links -> links ->
1570   subst -> subst -> o.
1571 compile-terms-diagnostic (triple D F01 F02) (triple D H01 H02) M1 M3 L1
1572   comp F01 H01 M1 M2 L1 L2 S1 S2,
1573   comp F02 H02 M2 M3 L2 L3 S2 S3.
1574
1575 type compile-terms
1576   list (triple diagnostic fm fm) ->
1577   list (triple diagnostic tm tm) ->
1578   map -> links -> subst -> o.
1579 compile-terms T H M L S :-
1580   fold6 compile-terms-diagnostic T H [] M_ [] L_ [] S_,
1581   deduplicate-map M_ M S_ S L_ L.
1582
1583 type make-eta-link-aux nat -> address -> address ->
1584   list tm -> links -> subst -> subst -> o.
1585 make-eta-link-aux z Ad1 Ad2 Scope1 L H1 H1 :-
1586   rev Scope1 Scope, eta-expand (uva Ad2 Scope) @one T1,
1587   L = [eval-link-eta (uva Ad1 Scope) T1].
1588 make-eta-link-aux (s N) Ad1 Ad2 Scope1 L H1 H3 :-
1589   rev Scope1 Scope, alloc H1 Ad H2,
1590   eta-expand (uva Ad Scope) @one T2,
1591   (pi x\ make-eta-link-aux N Ad Ad2 [x|Scope1] (L1 x) H2 H3),
1592   close-links L1 L2,
1593   L = [eval-link-eta (uva Ad1 Scope) T2 | L2].
1594
1595 type make-eta-link nat -> nat -> address -> address ->
1596   list tm -> links -> subst -> subst -> o.
1597 make-eta-link (s N) z Ad1 Ad2 Vars L H H1 :-
1598   make-eta-link-aux N Ad2 Ad1 Vars L H H1.
1599 make-eta-link z (s N) Ad1 Ad2 Vars L H H1 :-
1600   make-eta-link-aux N Ad1 Ad2 Vars L H H1.
1601 make-eta-link (s N) (s M) Ad1 Ad2 Vars Links H H1 :-
1602   (pi x\ make-eta-link N M Ad1 Ad2 [x|Vars] (L x) H H1),
1603   close-links L Links.
1604
1605 type deduplicate-map map -> map ->
1606   subst -> subst -> links -> links -> o.
1607 deduplicate-map [] [] H H L L.
1608 deduplicate-map [(mapping (fv 0) (hv M (arity LenM)) as X1) | Map1] Map2
1609   take-list Map1 (mapping (fv 0) (hv M' (arity LenM')) _) !,
1610   std.assert! (not (LenM = LenM')) "Deduplicate map, there is a bug",
1611   print "arity-fix links:" {ppmapping X1} "~!~" {ppmapping (mapping (fv
1612     make-eta-link LenM LenM' M M' [] New H1 H2,
1613     print "new eta link" {pplinks New},
1614     append New L1 L2,
1615     deduplicate-map Map1 Map2 H2 H3 L2 L3.
1616 deduplicate-map [A|As] [A|Bs] H1 H2 L1 L2 :-
1617   deduplicate-map As Bs H1 H2 L1 L2, !.
1618 deduplicate-map [A|_] _ H _ _ _ :-
1619   halt "deduplicating mapping error" {ppmapping A} {ho.ppsubst H}.
1620
1621
1622
1623
1624

```

15 THE PROGRESS FUNCTION

```
macro @one :- s z.
```

```
type contract-rigid list ho.tm -> ho.tm -> ho.tm -> o.
```

```
contract-rigid L (ho.lam F) T :-
```

```
  pi x\ contract-rigid [x|L] (F x) T. % also checks H Prefix does not see x
```

```
contract-rigid L (ho.app [H|Args]) T :-
```

```
  rev L LRev, append Prefix LRev Args,  
  if (Prefix = []) (T = H) (T = ho.app [H|Prefix]).
```

```
type progress-eta-link ho.tm -> ho.tm -> ho.subst -> ho.subst -> links -> o.
```

```
progress-eta-link (ho.app _ as T) (ho.lam x\ _ as T1) H H1 [] :- !,  
  ({eta-expand T @one} ==1 T1) H H1.
```

```
progress-eta-link (ho.con _ as T) (ho.lam x\ _ as T1) H H1 [] :- !,  
  ({eta-expand T @one} ==1 T1) H H1.
```

```
progress-eta-link (ho.lam _ as T) T1 H H1 [] :- !,  
  (T ==1 T1) H H1.
```

```
progress-eta-link (ho.uva _ _ as X) T H H1 [] :-  
  contract-rigid [] T T1, !, (X ==1 T1) H H1.
```

```
progress-eta-link (ho.uva Ad _ as T1) T2 H H1 [eval-link-eta T1 T2] :- !,  
  if (ho.not_occ Ad H T2) true fail.
```

```
type is-in-pf ho.tm -> o.
```

```
is-in-pf (ho.app [ho.uva _ _ | _]) :- !, fail.
```

```
is-in-pf (ho.lam B) :- !, pi x\ is-in-pf (B x).
```

```
is-in-pf (ho.con _).
```

```
is-in-pf (ho.app L) :- forall1 is-in-pf L.
```

```
is-in-pf N :- name N.
```

```
is-in-pf (ho.uva _ L) :- pattern-fragment L.
```

```
type arity ho.tm -> nat -> o.
```

```
arity (ho.con _) z.
```

```
arity (ho.app L) A :- len L A.
```

```
type occur-check-err ho.tm -> ho.tm -> ho.subst -> o.
```

```
occur-check-err (ho.con _) _ _ :- !.
```

```
occur-check-err (ho.app _) _ _ :- !.
```

```
occur-check-err (ho.lam _) _ _ :- !.
```

```
occur-check-err (ho.uva Ad _) T S :-  
  not (ho.not_occ Ad S T).
```

```
type progress-beta-link-aux ho.tm -> ho.tm ->
```

```
  ho.subst -> ho.subst -> links -> o.
```

```
progress-beta-link-aux T1 T2 S1 S2 [] :- is-in-pf T2, !,  
  (T1 ==1 T2) S1 S2.
```

```
progress-beta-link-aux T1 T2 S S [eval-link-beta T1 T2] :- !.
```

```
type progress-beta-link ho.tm -> ho.tm -> ho.subst ->
```

```
  ho.subst -> links -> o.
```

```
progress-beta-link T (ho.app [ho.uva V Scope | L] as T2) S S2 [eval-link-beta T1 T2] :- !,  
  arity T Arity, len L ArgsNb, ArgsNb >n Arity, !,  
  minus ArgsNb Arity Diff, mem.new S V1 S1,  
  eta-expand (ho.uva V1 Scope) Diff T1,  
  ((ho.uva V Scope) ==1 T1) S1 S2.
```

```
progress-beta-link (ho.uva _ _ as T) (ho.app [ho.uva Ad1 Scope1 | L] as T2) S1 S2 NewLinks :- !,
```

```
  append Scope1 L1 Scope1L,
```

```
  pattern-fragment-prefix Scope1L Scope2 L2,
```

```
  not (Scope1 = Scope2), !,
```

```
  mem.new S1 Ad2 S2,
```

```
  len Scope1 Scope1Len,
```

```
  len Scope2 Scope2Len,
```

```
  make-eta-link Scope1Len Scope2Len Ad1 Ad2 [] LinkEta S2 S3,
```

```
  if (L2 = []) (NewLinks = LinkEta, T2 = ho.uva Ad2 Scope2)
```

```
  (T2 = ho.app [ho.uva Ad2 Scope2 | L2],
```

```
  NewLinks = [eval-link-beta T T2 | LinkEta]).
```

```
progress-beta-link T1 (ho.app [ho.uva _ _ | _] as T2) _ _ _ :- !,  
  not (T1 = ho.uva _ _), !, fail.
```

```
progress-beta-link (ho.uva _ _ as T) (ho.app [ho.uva _ _ | _] as T2) S1 S2 :- !,  
  occur-check-err T T2 S1, !, fail.
```

```
progress-beta-link T1 (ho.app [ho.uva _ _ | _] as T2) H H1 [eval-link-beta T1 T2] :- !,
```

```
progress-beta-link T1 (ho.app [Hd | T1]) S1 S2 B :-
```

```
  ho.beta Hd T1 T3,
```

```
  progress-beta-link-aux T1 T3 S1 S2 B.
```

```
type solve-link-abs link -> links -> ho.subst -> ho.subst -> o.
```

```
solve-link-abs (ho.abs X) R H H1 :-
```

```
  pi x\ ho.copy x x => (pi S\ ho.deref S x x) =>
```

```
    solve-link-abs (X x) (R' x) H H1,
```

```
  close-links R' R.
```

```
solve-link-abs (@eval-link-eta A B) NewLinks S S1 :- !,
```

```
  progress-eta-link A B S S1 NewLinks.
```

```
solve-link-abs (@eval-link-beta A B) NewLinks S S1 :- !,
```

```
  progress-beta-link A B S S1 NewLinks.
```

```
type take-link link -> links -> link -> links -> o.
```

```
take-link A [B|XS] B XS :- link-abs-same-lhs A B, !.
```

```
take-link A [L|XS] B [L|YS] :- take-link A XS B YS.
```

```
type link-abs-same-lhs link -> link -> o.
```

```
link-abs-same-lhs (ho.abs F) B :-
```

```
  pi x\ link-abs-same-lhs (F x) B.
```

```
link-abs-same-lhs A (ho.abs G) :-
```

```
  pi x\ link-abs-same-lhs A (G x).
```

```
link-abs-same-lhs (@eval-link-eta (ho.uva N _) _) (@eval-link-eta (ho.uva N _) _) :- !.
```

```
type same-link-eta link -> link -> ho.subst -> ho.subst -> o.
```

```
same-link-eta (ho.abs F) B H H1 :- !, pi x\ same-link-eta (F x) B H H1.
```

```
same-link-eta A (ho.abs G) H H1 :- !, pi x\ same-link-eta A (G x) H H1.
```

```
same-link-eta (@eval-link-eta (ho.uva N S1) A)
```

```
  (@eval-link-eta (ho.uva N S2) B) H H1 :-
```

```
  std.map2 S1 S2 (x\y\r\ r = ho.copy x y) Perm,
```

```
  Perm => ho.copy A A',
```

```
  (A' ==1 B) H H1.
```

```
type solve-links links -> links -> ho.subst -> ho.subst -> o.
```

```
solve-links [] [] X X.
```

```

1741 solve-links [A|L1] [A|L3] S S2 :- take-link A L1 B L2, !,
1742   same-link-eta A B S S1,
1743   solve-links L2 L3 S1 S2.
1744 solve-links [L0|L1] L3 S S2 :- deref-link S L0 L,
1745   solve-link-abs L R S S1, !,
1746   solve-links L1 L2 S1 S2, append R L2 L3.

```

16 THE DECOMPIER

```

1750 type abs->lam ho.assignment -> ho.tm -> o.
1751 abs->lam (ho.abs T) (ho.lam R) :- !, pi x\ abs->lam (T x) (R x).
1752 abs->lam (ho.val A) A.
1753
1754 type commit-links-aux link -> ho.subst -> ho.subst -> o.
1755 commit-links-aux (@val-link-eta T1 T2) H1 H2 :-
1756   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
1757   (T1' ==1 T2') H1 H2.
1758 commit-links-aux (@val-link-beta T1 T2) H1 H2 :-
1759   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
1760   (T1' ==1 T2') H1 H2.
1761 commit-links-aux (ho.abs B) H H1 :-
1762   pi x\ commit-links-aux (B x) H H1.
1763
1764 type commit-links links -> links -> ho.subst -> ho.subst -> o.
1765 commit-links [] [] H H.
1766 commit-links [Abs | Links] L H H2 :-
1767   commit-links-aux Abs H H1, !, commit-links Links L H1 H2.
1768
1769 type decomp-subst map -> map -> ho.subst ->
1770   fo.subst -> fo.subst -> o.
1771 decomp-subst _ [A|_] _ _ :- fail.
1772 decomp-subst _ [] _ F F.
1773 decomp-subst Map [mapping (fv V0) (hv VM _)|T1] H F F2 :-
1774   mem.set? VM H T, !,
1775   ho.deref-assmt H T TTT,
1776   abs->lam TTT T', tm->fm Map T' T1,
1777   fo.eta-contract T1 T2, mem.assign V0 F T2 F1,
1778   decomp-subst Map T1 H F1 F2.
1779 decomp-subst Map [mapping _ (hv VM _)|T1] H F F2 :-
1780   mem.unset? VM H, decomp-subst Map T1 H F F2.
1781
1782 type tm->fm map -> ho.tm -> fo.fm -> o.
1783 tm->fm _ (ho.con C) (fo.fcon C).
1784 tm->fm L (ho.lam B1) (fo.flam B2) :-
1785   pi x y\ tm->fm _ x y => tm->fm L (B1 x) (B2 y).
1786 tm->fm L (ho.app L1) T :- forall12 (tm->fm L) L1 [Hd|T1],
1787   fo.mk-app Hd T1 T.
1788 tm->fm L (ho.uva VM TL) T :- mem L (mapping (fv V0) (hv VM _)),
1789   forall12 (tm->fm L) TL T1, fo.mk-app (fo.fuva V0) T1 T.
1790
1791 type add-new-map-aux ho.subst -> list ho.tm -> map ->
1792   map -> fo.subst -> fo.subst -> o.
1793 add-new-map-aux _ [] _ [] S S.
1794 add-new-map-aux H [T|Ts] L L2 S S2 :-
1795   add-new-map H T L L1 S S1,
1796   add-new-map-aux H Ts L1 L2 S1 S2.
1797
1798

```

```

1799 type add-new-map ho.subst -> ho.tm -> map ->
1800   map -> fo.subst -> fo.subst -> o.
1801 add-new-map _ (ho.uva N _) Map [] F1 F1 :-
1802   mem Map (mapping _ (hv N _)), !.
1803 add-new-map H (ho.uva N L) Map [Map1 | MapL] F1 F3 :-
1804   mem.new F1 M F2,
1805   len L Arity, Map1 = mapping (fv M) (hv N (arity Arity)),
1806   add-new-map H (ho.app L) [Map1 | Map] MapL F2 F3.
1807 add-new-map H (ho.lam B) Map NewMap F1 F2 :-
1808   pi x\ add-new-map H (B x) Map NewMap F1 F2.
1809 add-new-map H (ho.app L) Map NewMap F1 F3 :-
1810   add-new-map-aux H L Map NewMap F1 F3.
1811 add-new-map _ (ho.con _) _ [] F F :- !.
1812 add-new-map _ N _ [] F F :- name N.
1813
1814 type complete-mapping-under-ass ho.subst -> ho.assignment ->
1815   map -> map -> fo.subst -> fo.subst -> o.
1816 complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
1817   add-new-map H Val Map1 Map2 F1 F2.
1818 complete-mapping-under-ass H (ho.abs Abs) Map1 Map2 F1 F2 :-
1819   pi x\ complete-mapping-under-ass H (Abs x) Map1 Map2 F1 F2.
1820
1821 type complete-mapping ho.subst -> ho.subst ->
1822   map -> map -> fo.subst -> fo.subst -> o.
1823 complete-mapping _ [] L L F F.
1824 complete-mapping H [none | T1] L1 L2 F1 F2 :-
1825   complete-mapping H T1 L1 L2 F1 F2.
1826 complete-mapping H [some T0 | T1] L1 L3 F1 F3 :-
1827   ho.deref-assmt H T0 T,
1828   complete-mapping-under-ass H T L1 L2 F1 F2,
1829   append L1 L2 Lall,
1830   complete-mapping H T1 Lall L3 F2 F3.
1831
1832 type decompile map -> links -> ho.subst ->
1833   fo.subst -> fo.subst -> o.
1834 decompile Map1 L H0 F0 F02 :-
1835   commit-links L L1_ H0 H01, !,
1836   complete-mapping H01 H01 Map1 Map2 F0 F01,
1837   decomp-subst Map2 Map2 H01 F01 F02.
1838
1839

```

17 AUXILIARY FUNCTIONS

```

1840 type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A ->
1841   list A1 -> B -> B -> C -> C -> o.
1842 fold4 _ [] [] A A B B.
1843 fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0,
1844   fold4 F XS YS A0 A1 B0 B1.
1845
1846 type len list A -> nat -> o.
1847 len [] z.
1848 len [_|L] (s X) :- len L X.
1849
1850

```