

# HO unification from object language to meta language

Davide Fissore

davide.fissore@inria.fr

Université Côte d'Azur, Inria

France

Enrico Tassi

enrico.tassi@inria.fr

Université Côte d'Azur, Inria

France

## ABSTRACT

Specifying and implementing a logic from scratch requires significant effort. Logical Frameworks and Higher Order Logic Programming Languages provide dedicated, high-level Meta Languages (ML) to facilitate this task in two key ways: 1) variable binding and substitution are simplified when ML binders represent object logic ones; 2) proof construction, and even proof search, is greatly simplified by leveraging the unification procedure provided by the ML. Notable examples of ML are Elf [13], Twelf [14],  $\lambda$ Prolog [10] and Isabelle [20] which have been utilized to implement various formal systems such as First Order Logic [5], Set Theory [12], Higher Order Logic [11], and even the Calculus of Constuctions [4].

The object logic we are interested in is Coq's [18] Dependent Type Theory (DTT), for which we aim to implement a unification procedure  $\approx_o$  using the ML Elpi [3], a dialect of  $\lambda$ Prolog. Elpi's equational theory comprises  $\eta\beta$  equivalence and comes equipped with a higher order unification procedure  $\approx_\lambda$  restricted to the pattern fragment [9]. We want  $\approx_o$  to be as powerful as  $\approx_\lambda$  but on the object logic DTT. Elpi also comes with an encoding for DTT that works well for meta-programming [17, 16, 7, 6]. Unfortunately this encoding, which we refer to as  $\mathcal{F}_o$ , "underuses"  $\approx_\lambda$  by restricting it to first-order unification problems only. To address this issue, we propose a better-behaved encoding,  $\mathcal{H}_o$ , demonstrate how to map unification problems in  $\mathcal{F}_o$  to related problems in  $\mathcal{H}_o$ , and illustrate how to map back the unifiers found by  $\approx_\lambda$ , effectively implementing  $\approx_o$  on top of  $\approx_\lambda$  for the encoding  $\mathcal{F}_o$ .

We apply this technique to the implementation of a type-class [19] solver for Coq [18]. Type-class solvers are proof search procedures based on unification that back-chain designated lemmas, providing essential automation to widely used Coq libraries such as Stdpp/Iris [8] and TLC [1]. These two libraries constitute our test bed.

## KEYWORDS

Logic Programming, Meta-Programming, Higher-Order Unification, Proof Automation

### ACM Reference Format:

Davide Fissore and Enrico Tassi. XXXX 2024. HO unification from object language to meta language. In *YYY*. ACM, New York, NY, USA, 18 pages. <https://doi.org/ZZZZZZZZZZZZ>

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

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM  
<https://doi.org/ZZZZZZZZZZZZ>

## 1 INTRODUCTION

Specifying and implementing a logic from scratch requires significant effort. Logical Frameworks and Higher Order Logic Programming Languages provide dedicated, high-level Meta Languages (ML) to facilitate this task in two key ways: 1) variable binding and substitution are simplified when ML binders represent object logic ones; 2) proof construction, and even proof search, is greatly simplified by leveraging the unification procedure provided by the ML. Notable examples of ML are Elf [13], Twelf [14],  $\lambda$ Prolog [10] and Isabelle [20] which have been utilized to implement various formal systems such as First Order Logic [5], Set Theory [12], Higher Order Logic [11], and even the Calculus of Constuctions [4].

The object logic we are interested in is Coq's [18] Dependent Type Theory (DTT), and we want to code a type-class [19] solver for Coq [18] using the Coq-Elpi [17] meta programming framework. Type-class solvers are unification based proof search procedures that combine a set of designated lemmas in order to providing essential automation to widely used Coq libraries.

As the running example we take the Decide type class, from the Stdpp [8] library. The class identifies predicates equipped with a decision procedure. The following three designated lemmas (called Instances in the type-class jargon) state that: 1) the type `fin n`, of natural numbers smaller than `n` is finite; 2) the predicate `nfact n nf`, linking a natural number `n` to its prime factors `nf`, is decidable; 3) the universal closure of a predicate has a decision procedure if the predicate has and if its domain is finite.

```
Instance fin_fin n : Finite (fin n).          (* r1 *)
Instance nfact_dec n nf : Decision (nfact n nf). (* r2 *)
Instance forall_dec A P : Finite A →          (* r3 *)
  ∀x:A, Decision (P x) → Decision (∀x:A, P x).
```

Under this context of instances a type-class solver is able to prove the following statement automatically by back-chaining.

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

The encoding of DTT provided by Elpi, that we will discuss at length later in section ?? and ??, is an Higher Order Abstract Syntax (HOAS) datatype `tm` featuring (among others) the following constructors:

```
type lam  tm -> (tm -> tm) -> tm.    % lambda abstraction
type app  list tm -> tm.              % n-ary application
type all  tm -> (tm -> tm) -> tm.    % forall quantifier
type con  string -> tm.               % constants
```

Following standard  $\lambda$ Prolog [10] the concrete syntax to abstract, at the meta level, an expression `e` over a variable `x` is `«x\ e»`, and square brackets denote a list of terms separated by comma. As an example we show the encoding of the Coq term `«∀y:t, nfact y 3»`:

```
all (con"t") y\ app[con"nfact", y, con"3"]
```

We now illustrate the encoding of the three instances above as higher-order logic-programming rules: capital letters denote rule parameters; `:-` separates the rule's head from the premises; `pi w\` 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) :- decomp Pm P A, finite A, (r3a)
  pi x\ decision (app[P, x]).
```

Since `Pm` is an higher-order unification variable of type `tm`  $\rightarrow$  `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 `«decomp 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)  $\beta$ -redex in the second premise (the predicate under the `pi w\`):

```
decision (app[lam A (a\ app[con"nfact", a, con"3"]), w])
```

In turn this redex prevents the rule (r2) to backchain properly since the following unification problem has no solution:

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

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

*Contributions.* In this paper we discuss alternative encodings of Coq in Elpi (Section ??), 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. The code discussed in the paper can be accessed at the URL: <https://github.com/FissoreD/paper-ho>.

## 2 PROBLEM STATEMENT

The equational theory of Coq's Dependent Type Theory is very rich. In addition to the usual  $\eta\beta$ -equivalence for functions, terms (hence types) are compared up to proposition unfolding and fix-point unrolling. Still, for efficiency and predictability reasons, most form of automatic proof search employ a unification procedure that captures a simpler one, just  $\eta\beta$ , and that solves higher-order problems restricted to the pattern fragment  $\mathcal{L}_\lambda$  [9]. We call this unification procedure  $\approx_o$ .

The equational theory of the meta language Elpi that we want to use to implement a form of proof automation is strikingly similar, since it it comprises  $\eta\beta$  (for the meta language functions), and the unification procedure  $\approx_\lambda$  solves higher-order problems in  $\mathcal{L}_\lambda$ .

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

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

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

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

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

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

We write  $t_1 \approx_\lambda t_2 \mapsto \sigma$  when  $t_1$  and  $t_2$  unify with substitution  $\sigma$ ; we write  $\sigma t$  for the application of the substitution to  $t$ , and  $\sigma X = \{\sigma t \mid t \in X\}$  when  $X$  is a set; we write  $\sigma \subseteq \sigma'$  when  $\sigma$  is more general than  $\sigma'$ . We assume that the unification of our meta language is correct:

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

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

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

unification variable that stands in their place in the compiled term. These links are checked for or progress XXX improve....

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

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

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

$$\begin{aligned} \text{hstep}(\mathcal{T}, p, \sigma, \mathbb{L}) &\mapsto (\sigma'', \mathbb{L}') \stackrel{\text{def}}{=} \\ &\sigma \mathcal{T}_{p_l} \approx_\lambda \sigma \mathcal{T}_{p_r} \mapsto \sigma' \wedge \text{progress}(\mathbb{L}, \sigma \cup \sigma') \mapsto (\mathbb{L}', \sigma'') \\ \text{hrun}(\mathcal{S}, N) &\mapsto \rho_N \stackrel{\text{def}}{=} \\ &\mathcal{T} \times \mathbb{M} \times \mathbb{L}_0 = \{(t_j, m_j, l_j) | s_j \in \mathcal{S}, \langle s_j \rangle \mapsto (t_j, m_j, l_j)\} \\ &\bigwedge_{p=1}^N \text{hstep}(\mathcal{T}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p) \\ &\langle \sigma_N, \mathbb{M}, \mathbb{L}_N \rangle^{-1} \mapsto \rho_N \end{aligned}$$

Here *hstep* is made of two sub-steps: a call to  $\approx_\lambda$  (on the compiled terms) and a call to *progress* 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*  $\text{hrun}$ , *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  $s_1 \approx_o s_2$  by specializing the code of *hrun* to  $\mathcal{S} = \{s_1, s_2\}$  as follows:

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

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

PROPOSITION 2.3 (PROPERTIES OF  $\approx_o$ ).

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

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

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

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

Property 2.1 states that  $\approx_o$ , hence our compilation scheme, is resilient to unification problems outside  $\mathcal{L}_\lambda$  solved by a third party. We believe this property is of practical interest since we want the user to be able to add heuristics via hand written rules to the ones obtained by our compilation scheme. A Typical example is the following problem (*q*) that is outside  $\mathcal{L}_\lambda$ :

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

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

## 2.1 The intuition in a nutshell

A term  $s$  is compiled in a term  $t$  where every “problematic” sub term  $p$  is replaced by a fresh unification variable  $h$  and an accessory link that represent a suspended unification problem  $h \approx_\lambda p$ . As a result  $\approx_\lambda$  is “well behaved” on  $t$ , that is it does not contradict  $=_o$  as it would otherwise do on “problematic” terms. We now define “problematic” and “well behaved” more formally.

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

An example of term  $t$  in  $\diamond \eta$  is  $\lambda x. \lambda y. F y x$  since the substitution  $\rho = \{F \mapsto \lambda a. \lambda b. f b a\}$  makes  $\rho t = \lambda x. \lambda y. f x y$  that is the eta long form of  $f$ . This term is problematic since its rigid part, the  $\lambda$ -abstractions, cannot justify a unification failure against, say, a constant.

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

An example of  $t$  in  $\diamond \beta$  is  $F a$  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 (Subterms  $\mathcal{P}(t)$ ). The set of sub terms of  $t$  is the largest set  $\mathcal{P}(\sqcup)$  that can be obtained by the following rules.

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

We write  $\mathcal{P}(X) = \bigcup_{t \in X} \mathcal{P}(t)$  when  $X$  is a set of terms.

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  $\text{hstep}$  never “commits” an unneeded  $\lambda$ -abstraction in  $\sigma$  (a  $\lambda$  that could be erased by an  $\eta$ -contraction), nor puts in  $\sigma$  a flexible application outside  $\mathcal{L}_\lambda$  (an application node that could be erased by a  $\beta$ -reduction).

Note that proposition 2.8 does not hold for  $\approx_o$  since decompilation can introduce (actually restore) terms in  $\diamond\eta$  or  $\diamond\beta$  that were move out of the way (put in  $\mathbb{L}$ ) during 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  $\text{tm}$ ). A fully shallow encoding unfortunately does not fit our use case, although it would make the running example work:

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

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

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

The second reason is the encoding for Coq is used for meta programming the system, hence it must accommodate the manipulation of terms that are now know 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: $\mathcal{F}_o$ AND $\mathcal{H}_o$

In order to reason about unification we provide a description of the  $\mathcal{F}_o$  and  $\mathcal{H}_o$  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  $\text{all}$  quantifier of DTT we used in the example in Section 1 together with the type notation of terms carried by the  $\text{lam}$  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 addr -> fm.     type uva addr -> list tm -> tm.
```

Figure 1: The  $\mathcal{F}_o$  and  $\mathcal{H}_o$  languages

Unification variables (fuva term constructor) in  $\mathcal{F}_o$  have no explicit scope: the arguments of an higher order variable are given via

the fapp constructor. For example the term  $P \ x$  is represented as  $\text{fapp}[fuva \ N, \ x]$ , where  $N$  is a memory address and  $x$  is a bound variable.

In  $\mathcal{H}_o$  the representation of  $P \ x$  is instead  $\text{uva} \ N \ [x]$ , since unification variables come equipped with an explicit scope. We say that the unification variable occurrence  $\text{uva} \ N \ L$  is in  $\mathcal{L}_\lambda$  if and only if  $L$  is made of distinct names. The predicate to test this condition is called pattern-fragment:

```
type pattern-fragment list A -> o.
```

The name builtin predicate tests if a term is a bound variable.<sup>2</sup>

In both languages unification variables are identified by a natural number representing a memory address. The memory and its associated operations are described below:

```
kind addr type.
type addr nat -> addr.
typeabbrev (mem A) (list (option A)).

type set? addr -> mem A -> A -> o.
type unset? addr -> mem A -> o.
type assign addr -> mem A -> A -> mem A -> o.
type new mem A -> addr -> mem 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, while new finds the first unused address and sets it to none.

Since in  $\mathcal{H}_o$  unification variables have a scope, their solution needs to be abstracted over it to enable the instantiation of a single solution to different scopes. This is obtained via the  $\text{inctx}$  container, and in particular via its  $\text{abs}$  binding constructor. On the contrary a solution to a  $\mathcal{F}_o$  variable is a plain term.

```
typeabbrev fsubst (mem fm).

kind inctx type -> type.
type abs (tm -> inctx A) -> inctx A.
type val A -> inctx A.
typeabbrev assignment (inctx tm).
typeabbrev subst (mem assignment).
```

We call  $\text{fsubst}$  the memory of  $\mathcal{F}_o$ , while we call  $\text{subst}$  the one of  $\mathcal{H}_o$ . Both have the invariant that they are not cyclic, TODO: explain.

INVARIANT 1. Any term in the substitution of  $\mathcal{H}_o$  is in normal form. TODO: move this to the right place TODO: definire normal form, maybe in definition 2.7

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

kind fvariable type.
type fv addr -> fvariable.

kind hvariable type.
type hv addr -> arity -> hvariable.

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

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



INVARIANT 2 (UNIFICATION VARIABLE ARITY). *Each variable  $A$  in  $\mathcal{H}_o$  has a (unique) arity  $N$  and each occurrence  $(uva\ A\ L)$  is such that  $(len\ L\ N)$  holds*

The compiler establishes a mapping between variables of the two languages. In order to preserve invariant 2 we store the arity of each hvariable in the mapping and we reuse an existing mapping only if the arity matches.

TODO: add ref to section 7

```
type m-alloc fvariable -> hvariable -> mmap -> mmap ->
  subst -> subst -> o. (m-alloc)
m-alloc Fv Hv M M S S :- mem M (mapping Fv Hv), !.
m-alloc Fv Hv M [mapping Fv Hv|M] S S1 :- Hv = hv N _,
  alloc S N S1.
```

When a single fvariable occurs multiple times with different numbers of arguments the compiler generates multiple mappings for it, on a first approximation, and then makes the mapping bijective by introducing link- $\eta$ ; this detail is discussed in section 6.

As we mentioned in section 2.1 the compiler replaces terms in  $\Diamond\eta$  and  $\Diamond\beta$  with fresh variables linked to the problematic terms. Each class of problematic terms has a dedicated link.

```
kind baselink type.
type link-eta tm -> tm -> baselink.
type link-beta tm -> tm -> baselink.
typeabbrev link (inctx baselink).
typeabbrev links (list link).
```

The right hand side of a link, the problematic term, can occur under binders. To accommodate this situation the compiler wraps baselink using the inctx container.

INVARIANT 3 (LINK LEFT HAND SIDE). *The left hand side of a suspended link is a variable.*

New links are suspended by construction. If the left hand side variable is assigned during a step, then the link is considered for progress and possibly eliminated. This is discussed in section 6 and ??.

## 4.1 Notational conventions

When we write  $\mathcal{H}_o$  terms outside code blocks we follow the usual  $\lambda$ -calculus notation, reserving  $f, g, a, b$  for constants,  $x, y, z$  for bound variables and  $X, Y, Z, F, G, H$  for unification variables. However we need to distinguish between the “application” of a unification variable to its scope and the application of a term to a list of arguments. We write the scope of unification variables in subscript while we use juxtaposition for regular application. Here a few examples:

```
f a      app[con "f", con "a"]
 $\lambda x.F_x a$  lam x\ app[uva F [x], con "a"]
 $\lambda x.\lambda y.F_{xy}$  lam x\ lam y\ uva F [x, y]
 $\lambda x.F_x x$  lam x\ app[uva F [x], x]
```

When detailing examples we write links as equations between terms under a context. The equality sign is subscripted with kind of baselink. For example  $x \vdash A =_\beta F_x a$  corresponds to:

```
abs x\ val (link-beta (uva A []) (app[uva F [x], con "a"]))
```

When it is clear from the context we shall use the same syntax for  $\mathcal{F}_o$  terms (although we never subscript unification variables).

## 4.2 Equational theory and Unification

In order to express properties ?? we need to equip  $\mathcal{F}_o$  and  $\mathcal{H}_o$  with term equality, substitution application and unification.

*Term equality:*  $=_o$  vs.  $=_\lambda$ . We extend the equational theory over ground terms to the full languages by adding the reflexivity of unification variables (a variable is equal to itself).

The first four rules are common to both equalities and correspond to  $\alpha$ -equivalence. In addition to that  $=_o$  has rules for  $\eta$  and  $\beta$ -equivalence.

```
type (=o) fm -> fm -> o. (=o)
fcon X =o fcon X.
fapp A =o fapp B :- forall2 (=o) A B.
flam F =o flam G :- pi x\ x =o x => F x =o G x.
fuva N =o fuva N.
flam F =o T :- (eta)
  pi x\ beta T [x] (T' x), x =o x => F x =o T' x.
T =o flam F :- (eta_r)
  pi x\ beta T [x] (T' x), x =o x => T' x =o F x.
fapp [flam X | L] =o T :- beta (flam X) L R, R =o T. (beta_l)
T =o fapp [flam X | L] :- beta (flam X) L R, T =o R. (beta_r)

type (=lambda) tm -> tm -> o.
con C =lambda fcon C.
app A =lambda fapp B :- forall2 (=lambda) A B.
lam F =lambda flam G :- pi x\ x =lambda x => F x =lambda G x.
uva N A =lambda fuva N B :- forall2 (=lambda) A B.
```

The main point in showing these equality tests is to remark how weaker  $=_\lambda$  is, and to identify the four rules that need special treatment in the implementation of  $=_\lambda$ .

For reference,  $(beta\ T\ A\ R)$  reduces away lam nodes in head position in T whenever the list A provides a corresponding argument.

```
type beta fm -> list fm -> fm -> 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.
```

The name predicate holds only on nominal constants (i.e. bound variables). Elpi provides it as a builtin, but one could implement it by systematically loading the hypothetical rule name x every time a nominal constant is postulated via `pi x\`.

*Substitution application:*  $\rho s$  and  $\sigma t$ . Applying the substitution corresponds to dereferencing a term with respect to the memory. To ease the comparison we split  $\mathcal{F}_o$  dereferencing into a fder step and a napp one. The former step replaces references to memory cells that are set with their values, ans has a corresponding operation in  $\mathcal{H}_o$ , namely deref. On the contrary napp, in charge of “flattening” fapp nodes, has no corresponding operation in  $\mathcal{H}_o$ . The reasons for this asymmetry is that an fapp node with a flexible head is always mapped to a uva (as per sections ??), preventing nested applications to materialize.

```
type fder fsubst -> fm -> fm -> o.
fder _ (fcon C) (fcon C).
```

```

581 fder S (fapp A) (fapp B) :- map (fder S) A B.
582 fder S (flam F) (flam G) :-
583   pi x\ fder S x x => fder S (F x) (G x).
584 fder S (fuva N) R :- set? N S T, fder S T R.
585 fder S (fuva N) (fuva N) :- unset? N S.
586
587 type fderef fsubst -> fm -> fm -> o. (ps)
588 fderef S T T2 :- fder S T T1, napp T1 T2.
589
590 type napp fm -> fm -> o.
591 napp (fcon C) (fcon C).
592 napp (fuva A) (fuva A).
593 napp (flam F) (flam F1) :-
594   pi x\ napp x x => napp (F x) (F1 x).
595 napp (fapp [fapp L1 |L2]) T :- !,
596   append L1 L2 L3, napp (fapp L3) T.
597 napp (fapp L) (fapp L1) :- map napp L L1.

```

Note that the cut operator is inessential, it could be removed at the cost of a verbose test on the head of `L` in the last rule (`L` head can be `fcon`, `flam` or a name).

Applying the substitution in  $\mathcal{H}_0$  is very similar, with the caveat that assignments have to be moved to the current scope, i.e. renaming the abs-bound variables with the names in the scope of the unification variable occurrence.

```

605 type deref subst -> tm -> tm -> o. (st)
606 deref _ (con C) (con C).
607 deref S (app A) (app B) :- map (deref S) A B.
608 deref S (lam F) (lam G) :-
609   pi x\ deref S x x => deref S (F x) (G x).
610 deref S (uva N L) R :- set? N S A,
611   move A L T, deref S T R.
612 deref S (uva N A) (uva N B) :- unset? N S,
613   map (deref S) A B.

```

Note that move strongly relies on invariant 2: the length of the arguments of all occurrences of a unification variable and the number of abstractions in its assignment have to match. In turn this grants that move never fails.

```

619 type move assignment -> list tm -> tm -> o.
620 move (abs Bo) [H|L] R :- move (Bo H) L R.
621 move (val A) [] A.
622

```

*Term unification:*  $\approx_o$  vs.  $\approx_\lambda$ . In this paper we assume to have an implementation of  $\approx_\lambda$  that satisfies properties 1 and 2. Although we provide an implementation in the appendix (that we used for testing purposes) we only describe its signature here. Elpi is expected to provide this brick, as well as any other implementation of  $\lambda$ Prolog.

```

628 type ( $\approx_\lambda$ ) tm -> tm -> subst -> subst -> o.
629

```

The only detail worth discussing is the fact that the procedure updates a substitution, rather than just crafting one as presented in section 2. The reason is that the algorithm folds over a term, updating a substitution while it traverses it.

## 5 BASIC SIMULATION OF $\mathcal{F}_0$ IN $\mathcal{H}_0$

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

implement an  $\approx_o$  that respects  $\beta$ -conversion for terms in  $\mathcal{L}_\lambda$ . The extension to  $\eta\beta$ -conversion is described in Section 6 and the support for terms outside  $\mathcal{L}_\lambda$  in Section 8.

### 5.1 Compilation

The main task of the compiler is to recognize  $\mathcal{F}_0$  variables standing for functions and map them to higher order variables in  $\mathcal{H}_0$ . In order to bring back the substitution from  $\mathcal{H}_0$  to  $\mathcal{F}_0$  the compiler builds a “memory map” connecting the the kind of variables using routine (*malloc*).

The signature of the comp predicate below allows for the generation of links (suspended unification problems) that play no role in this section but play a major role in Sections 6 and 8. With respect to 2 the signature also allows for updates to the substitution. The code below only allocates space for the variables, i.e. sets their memory address to none, a details not worth mentioning in the previous discussion.

```

637 type comp fm -> tm -> mmap -> mmap -> links -> links ->
638   subst -> subst -> o.
639 comp (fcon C) (con C) M M L L S S.
640 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :- (c $\lambda$ )
641   comp-lam F F1 M1 M2 L1 L2 S1 S2.
642 comp (fuva A) (uva B []) M1 M2 L L S1 S2 :-
643   m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
644 comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
645   pattern-fragment Ag, !,
646   fold6 comp Ag Ag1 M1 M1 L L S1 S1,
647   len Ag Arity,
648   m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
649 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :-
650   fold6 comp A A1 M1 M2 L1 L2 S1 S2.

```

This preliminary version of comp recognizes  $\mathcal{F}_0$  variables applied to a (possibly empty) duplicate free list of names (i.e. pattern-fragment detects variables in  $\mathcal{L}_\lambda$ ). Note tha compiling `Ag` cannot create new mappings nor links, since `Ag` is made of bound variables and the hypothetical rule loaded by `comp-lam` (see below) grants this property.

```

657 type comp-lam (fm -> fm) -> (tm -> tm) ->
658   mmap -> mmap -> links -> links -> subst -> subst -> o.
659 comp-lam F G M1 M2 L1 L3 S1 S2 :-
660   pi x y\ (pi M L S\ comp x y M M L L S S) =>
661   comp (F x) (G y) M1 M2 L1 (L2 y) S1 S2,
662   close-links L2 L3.

```

In the code above the syntax `pi x y\ .` is syntactic sugar for iterated pi abstraction, as in `pi x\ pi y\ .`

The auxiliary function `close-links` tests if the bound variable `v` really occurs in the link. If it is the case the link is wrapped into an additional abs node binding `v`. In this way links generated deep inside the compiled terms can be moved outside their original context of binders.

```

692 type close-links (tm -> links) -> links -> o.
693 close-links (_[]) [].
694 close-links (v\ [X |L v]) [X|R] :- !, close-links L R.
695 close-links (v\ [X v|L v]) [abs X|R] :- close-links L R.

```

explain  
better

Note that we could remove the second rule, whose purpose is to make links more readable by pruning unneeded abstractions (unused context entries).

## 5.2 Execution

A step in  $\mathcal{H}_o$  consists in unifying two terms and reconsidering all links for progress. If any of the two tasks fail we say that the entire step fails, and it is at this granularity that we can relate steps in the two languages.

```
type hstep tm -> tm -> links -> links -> subst -> subst -> o.
hstep T1 T2 L1 L2 S1 S3 :-
  (T1  $\approx_\lambda$  T2) S1 S2,
  progress L1 L2 S2 S3.
```

Note that the infix notation  $((A \approx_\lambda B) C D)$  is syntactic sugar for  $((\approx_\lambda) A B C D)$ .

Reconsidering links is a fixpoint, since the progress of a link can update the substitution and in turn enable another link to progress.

```
type progress links -> links -> subst -> subst -> o.
progress L L2 S1 S3 :-
  progress1 L L1 S1 S2, !,
  occur-check-links L1,
  if (L = L1, S1 = S2)
    (L2 = L1, S3 = S1)
    (progress L1 L2 S2 S3).
```

In the base compilation scheme `progress1` is the identity on both the links and the substitution, so the fixpoint trivially terminates. Sections 6 and 8 add rules to `progress1` and justify why the don't hinder termination.

Since compilation moves problematic terms out of the compiled terms,  $\approx_\lambda$  can only perform a partial occur check. For example the unification problem  $X \approx_\lambda f Y$  cannot generate a cyclic substitution alone, but should be disallowed if a  $\mathbb{L}$  contains a link like  $\vdash Y =_\eta \lambda z.X_z$ : We don't know yet if  $Y$  will feature a lambda in head position, but we surely know it contains  $X$ , hence  $f Y$  and that triggers occur check. The procedure `occur-check-links` is in charge of ensuring that each link does not represent a (suspended) unification problem doomed to fail because of occur check. This check is needed in order to guarantee proposition 2.2 (simulation fidelity).

## 5.3 Substitution decompilation

Decompiling the substitution requires to first force the progress of links and then allocating new unassigned variables in the substitution for  $\mathcal{F}_o$  and finally decompiling all assignments. Note that invariant 3 and the occur check allows us to update the subst.

```
type decompile mmap -> links -> subst ->
  fsubst -> fsubst -> o.
decompile M1 L S F1 F3 :-
  commit-links L S S1,
  complete-mapping S1 S1 M1 M2 F1 F2,
  decomp M2 M2 S1 F2 F3.
```

TODO: What is `commit-links` and `complete-mapping`?, maybe `complete-mapping` can be hidden in the code rendering?

Decompiling an assignment requires to turn abstractions into lambdas. For aesthetic purposes we also eta-contract the result (not needed since  $\mathcal{F}_o$  equality can do that)

```
type decomp mmap -> mmap -> subst -> fsubst -> fsubst -> o.
decomp _ [] _ F F.
decomp M [mapping (fv V) (hv H _)]MS S F1 F3 :- set? H S A,
  deref-assmt S A A1,
  abs->lam A1 T, decomp M T T1,
  eta-contract T1 T2,
  assign V F1 T2 F2,
  decomp M MS S F2 F3.
decomp M [mapping _ (hv H _)]MS S F1 F2 :- unset? H S,
  decomp M MS S F1 F2.
```

Finally decompiling a term is trivial, now that we have an extended mapping containing all unassigned variables  $\approx_\lambda$  may have introduced.

```
type decomp mmap -> tm -> fm -> o.
decomp _ (con C) (fcon C).
decomp M (app A) (fapp B) :- map (decomp M) A B.
decomp M (lam F) (flam G) :-
  pi x y\ (pi M\ decomp M x y => decomp M (F x) (G y)).
decomp M (uva Hv Ag) R :-
  mem M (mapping (fv Fv) (hv Hv _)),
  map (decomp M) Ag Bg,
  beta (fuva Fv) Bg R.
```

Note that we use beta to build `fapp` nodes when needed (if `Ag` is empty no `fapp` node should appear).

INVARIANT 4. *TODO: dire che il mapping è bijective*

## 5.4 Definition of $\approx_o$ and its properties

```
type ( $\approx_o$ ) fm -> fm -> fsubst -> o.
(A  $\approx_o$  B) F :-
  comp A A' [] M1 [] [] S1,
  comp B B' M1 M2 [] [] S1 S2,
  hstep A' B' [] [] S2 S3,
  decomp M2 M2 S3 [] F.
```

The code given so far applies to terms in  $\beta\eta$ -normal form where unification variables in  $\mathcal{F}_o$  can occur non linearly but always with the same number of arguments, and where their arguments are distinct names (as per  $\mathcal{L}_\lambda$ ).

LEMMA 5.1 (COMPILATION ROUND TRIP). *If  $\text{comp } S T [] M [] [] \_ [] \_$  then  $\text{decomp } M T S$*

PROOF SKETCH. trivial, since the terms are beta normal beta just builds an app.  $\square$

LEMMA 5.2. *Properties (correct) and (complete) hold for the implementation of  $\approx_o$  above*

PROOF SKETCH. In this setting  $=_\lambda$  is as strong as  $=_o$  on ground terms. What we have to show is that whenever two different  $\mathcal{F}_o$  terms can be made equal by a substitution  $\rho$  (plus the  $\beta_l$  and  $\beta_r$  if needed) we can find this  $\rho$  by finding a  $\sigma$  via  $\approx_\lambda$  on the corresponding  $\mathcal{H}_o$  terms and by decompiling it. If we look at the  $\mathcal{F}_o$  terms, there are two interesting cases:

- $\text{fuva } X \approx_o s$ . In this case after  $\text{comp}$  we have  $Y \approx_\lambda t$  that succeeds with  $\sigma = \{Y \mapsto t\}$  and  $\sigma$  is decompiled to  $\rho = \{Y \mapsto s\}$ .
- $\text{fapp}[\text{fuva } X \mid L] \approx_o s$ . In this case we have  $Y_{\vec{x}} \approx_\lambda t$  that succeeds with  $\sigma = \{\vec{y} \vdash Y \mapsto t[\vec{x}/\vec{y}]\}$  that in turn is decompiled to  $\rho = \{Y \mapsto \lambda \vec{y}.s[\vec{x}/\vec{y}]\}$ . Thanks to  $\beta_l (\lambda \vec{y}.s[\vec{x}/\vec{y}]) \vec{x} \approx_o s$ .

Since the mapping is a bijection occur check in  $\mathcal{H}_o$  corresponds to occur check in  $\mathcal{F}_o$ .  $\square$

LEMMA 5.3. *Properties simulation (2.1) and fidelity (2.2) hold*

PROOF SKETCH. Since  $\text{progress1}$  is trivial  $\text{fstep}$  and  $\text{hstep}$  are the same, that is in this context where input terms are  $\beta\eta$ -normal and we disregard  $\eta$ -equivalence  $\approx_\lambda$  is equivalent to  $\approx_o$ .  $\square$

## 5.5 Limitations of by this basic scheme

$$\lambda xy.F \ y \ x = \lambda xy.x \quad (6)$$

$$\lambda x.f \ (F \ x) \ x = f \ (\lambda y.y) \quad (7)$$

Note that here  $F$  is used with different arities, moreover in the second problem the left hand side happens to be an eta expansion (of  $f(\lambda y.y)$ ) only after we discover that  $F = \lambda x\lambda y.y$  (i.e. that  $F$  discards the  $x$  argument). Both problems are addressed in the next section.

## 6 HANDLING OF $\Diamond\eta$

$\eta$ -reduction is an equivalence relation over terms where a term of the form  $\lambda x.t \ x$  can be converted to  $t$  any time  $x$  does not occur as a free variable in  $t$ . We call  $\lambda x.t \ x$  the  $\eta$ -expanded version of  $t$ . The implementation of the  $\text{comp}$  relation given in section 5 compiles the  $\mathcal{F}_o$  terms  $t_1 = \text{flam } x \mid \text{fapp} [\text{fuva } A, \ x]$  and  $t_2 = \text{fcon } "f"$  into the  $\mathcal{H}_o$  terms  $t'_1 = \text{lam } x \mid \text{uva } A' \ x$  and  $t'_2 = \text{fcon } "f"$  with mapping  $A \mapsto A'^1$ . However, if the oracle sets  $A'$  to the constant  $"f"$ , the unification of  $t'_1$  and  $t'_2$  in the meta language will fail even though  $t_1 \approx_o t_2$ . The reason of this failure is attributed to the fact that  $t'_1 = \text{lam } x \mid \text{app}[\text{con } "f", \ x]$  cannot be unified with  $t'_2 = \text{con } "f"$  since the two terms have different rigid heads. We solve this unification problem by adapting the  $\text{comp}$  relation such that it recognizes  $\Diamond\eta$  subterms  $s$  and replaces them with fresh  $\mathcal{H}_o$  variables  $v$ . This link between the variable  $v$  and the subterm  $t$  is stored in what we call  $\text{link-}\eta$  which is an object with the following type

**type**  $\text{link-}\eta \ \text{tm} \rightarrow \text{tm} \rightarrow \text{baselink}$

where, as sketched in section 4, the term on the left hand side (lhs) is linked with its left counterpart (rhs).

$\text{link-}\eta$  are added in the link store ( $\mathbb{L}$ ) and activated when special conditions are satisfied on lhs or rhs. These link activations are managed by extending the  $\text{progress1}$  predicate (see section 5.2). We claim that  $\text{link-}\eta$  progression does not contradict invariant 3 and we add the following invariant:

INVARIANT 5 ( $\text{link-}\eta$  rhs). *The rhs of a  $\text{link-}\eta$  having the shape  $\lambda x.F_x$  where  $F_x$  is a term not starting with the  $\text{lam}$  constructor.*

In the next three subsections we explain how we detect  $\Diamond\eta$  terms, how we compile them and how  $\text{link-}\eta$  are activated during the execution of the program and provide justification for why invariants 3 and 5 remain true.

## 6.1 Detection of $\Diamond\eta$

Compiling term with  $\Diamond\eta$  terms forces us to determine if,  $\lambda x.T_x$ , for any term  $T$  having  $x$  in scope, can be a  $\eta$ -expansion, i.e. under a given substitution  $\sigma$ , we have  $\sigma(\lambda x.T_x) = t$ . This  $\Diamond\eta$  detection is not a trivial operation as it may seems.

$$\lambda x.f \ A_x \quad (8)$$

$$\lambda x.f \ x \ A_x \quad (9)$$

$$\lambda x.\lambda y.f \ A_x \ B_{yx} \quad (10)$$

In the examples above, the first expression is a  $\Diamond\eta$  since  $A_x$  can reduce to  $x$ , the second one is not a  $\Diamond\eta$  since for any substitution for  $A_x$ ,  $x$  is not free in  $f \ x$ . The third equation is a bit more complicated since, we have a spine of lambdas, this means that the whole term is a  $\Diamond\eta$ , if the inner  $\lambda$ -term is an  $\eta$ -expansion of a term  $t$ , and  $t$  can be reduced to a term on the form  $t'x$  where  $x$  is not free in  $t'$ . Indeed, eq. 10 is a  $\Diamond\eta$  under the substitution  $\sigma = \{A \mapsto \lambda x.x, B \mapsto \lambda x.\lambda y.x\}$ .

TODO: clarify this As a remark, note that  $\lambda x.f \ A_x \ x$  is a  $\Diamond\eta$ , since, despite,  $x$  occurs in  $f \ A_x$ , it is still possible that this subterm does not use  $x$ , for example if  $A$  is a function on the form  $\lambda x.a$ , where  $a$  is a defined constant. In this case, the  $\Diamond\eta$  should consider that the bound variable  $x$  does not “rigidly” occur in the given subterm.

We can now define more formally the two auxiliary relation we need for  $\Diamond\eta$  detection:

**Definition 6.1 (reduce-to).** A term  $t$  reduce to a name  $x$ , if  $\exists \sigma, \sigma t = x$ . In particular, for any term  $t$ ,  $\lambda x_1 \dots x_n.t_{x_1 \dots x_n}$  reduces to a bound variable  $x$  if one of the three following cases is satisfied: 1)  $n = 0$  and  $t = x$ ; 2)  $t$  is the application of  $x$  to a list of terms  $l$  and each  $l_i$  reduces to  $x_i$  (since  $\lambda x_1 \dots x_n.x_{x_1 \dots x_n} = x$ ); 3)  $t$  is a unification variable in  $\mathcal{L}_\lambda$  with scope  $s$ , and for any  $v \in \{x, x_1 \dots x_n\}$ , there exists a  $s_i \in s$ , such that  $s_i$  reduces to  $v$

**Definition 6.2 (occurs-rigidly).** A name  $x$  occurs rigidly in a term  $t$ , if  $\forall \sigma, x \in \mathcal{P}(\sigma t)$

In other words  $x$  occurs-rigidly in  $t$  if it occurs in  $t$  outside of the scope of unification variables since their instantiations are allowed to discard their scope.

Finally, we can derive the implementation for  $\Diamond\eta$  detection:

**Definition 6.3 ( $\Diamond\eta$  detection).** A term  $\lambda x_1 \dots x_n.t$  where each  $x_i$  occurs in  $t$  is a  $\Diamond\eta$  if either: 1)  $t$  is a constant applied to arguments  $l_1 \dots l_m$  such that  $m \geq n$  and every  $l_{m-n+i}$  reduces-to  $x_i$  and no  $x_i$  occurs-rigidly in  $l_{1 \dots m-n-1}$ ; or 2)  $t$  is a unification variable with scope  $s$  and for each  $x_i$  there exists a  $s_j \in s$  such that  $s_j$  reduces-to  $x_i$ .

As a final remark, the  $\Diamond\eta$  detection defined just before is an over-approximation, in the sense that there exists some terms  $t$  considered as  $\Diamond\eta$ , such that for all substitution  $\sigma$ ,  $\sigma t$  is not an  $\eta$ -expansion. A small example is  $\lambda x.f \ A_x \ A_x$ . On the other hand, we also point out that the  $\Diamond\eta$  detection spot out potential  $\eta$ -expansion for terms that are not in  $\mathcal{L}_\lambda$ . For example,  $\lambda x.F \ G_x$  is considered as  $\Diamond\eta$ , since we have the application of term whose argument can reduce to  $x$ .

The implementation we propose for the  $\Diamond\eta$  relation is given below.



```

929 type occurs-rigidly fm -> fm -> o.
930 occurs-rigidly N N.
931 occurs-rigidly _ (fapp [fuva _|_]) :- !, fail.
932 occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.
933 occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).
934
935 type reducible-to list fm -> fm -> fm -> o.
936 reducible-to _ N N :- !.
937 reducible-to L N (fapp [fuva _|Args]) :- !,
938   forall1 (x\ exists (reducible-to [ ] x) Args) [N|L].
939 reducible-to L N (flam B) :- !,
940   pi x\ reducible-to [x | L] N (B x).
941 reducible-to L N (fapp [N|Args]) :-
942   last-n {len L} Args R,
943   forall2 (reducible-to [ ] R {rev L}).
944
945 type maybe-eta fm -> list fm -> o. (◇η)
946 maybe-eta (fapp [fuva _|Args]) L :- !,
947   forall1 (x\ exists (reducible-to [ ] x) Args) L, !.
948 maybe-eta (flam B) L :- !, pi x\ maybe-eta (B x) [x | L].
949 maybe-eta (fapp [fcon _|Args]) L :-
950   split-last-n {len L} Args First Last,
951   none (x\ exists (y\ occurs-rigidly x y) First) L,
952   forall2 (reducible-to [ ] {rev L} Last).

```

## 6.2 Compilation

Thanks to the maybe-eta predicate, we can detect “ $\eta$ -problematic” terms and, consequently replace them with fresh  $\mathcal{H}_o$  unification variables at compilation time. The code below illustrate how this relation is used to for term compilation.

```

960 comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
961   maybe-eta (flam F) [ ], !,
962   alloc S1 A S2,
963   comp-lam F F1 M1 M2 L1 L2 S2 S3,
964   get-scope (lam F1) Scope,
965   L3 = [val (link-eta (uva A Scope) (lam F1)) | L2].

```

This rule, to be inserted just before rule  $(c_\lambda)$  from the code in section 5, verifies if the  $\mathcal{F}_o$  term  $t$  received in entry is a  $\Diamond\eta$ . Let  $\lambda t'$  be the compiled version of  $t$ , then the fresh variable  $A$  returned as the new  $\mathcal{H}_o$  term as in scope all the free names in  $t'$ . The critical part of this compilation is the creation of the link- $\eta$ , which links the variable  $A$  with  $t$ . This link creation enforce invariant 3 and invariant 5, since clearly the lhs is a variable and the rhs is a term starting with the  $\text{lam}$  constructor. We can also note that for any name  $x$ ,  $tx = t'x$  never start with  $\text{lam}$ , since by contradiction, if 1)  $tx$  is an  $\eta$ -expansion, then the recursive call to  $\text{comp}$  would have replaced  $tx$  with a fresh variable, which contradict the hypothesis and if 2)  $t'$  is not an  $\eta$ -expansion, then neither  $\lambda t'$  is.

## 6.3 Progress

link- $\eta$  are meant to delay the unification of “problematic” terms. In the following, we call  $\mathbb{L}$  the list of suspended links.

In order to activate a link- $\eta$   $l$ , we need to implement new rules for the progress1 predicate. After passing under all the abs constructors of  $l$ , there are two cases making a link- $\eta$  to progress, 1)

lhs is instantiated to a rigid term 2) rhs can be  $\eta$ -reduced to a term with rigid head. If lhs is instantiated to a rigid term  $t$ , by invariant 1, we know that  $t$  does not contain any  $\Diamond\eta$ . Let  $t'$  the right hand side, if  $t$  is a constant or a function application, then,  $t'$ , which by construction has  $\text{lam}$  as head, should be an  $\eta$ -expansion. We are therefore allowed to unify  $\lambda x.t x$  (the  $\eta$ -expanded version of  $t$ ) with  $t'$ . Finally, if lhs starts has  $\text{lam}$  as constructor, then  $t$  is not an  $\eta$ -expansion and therefore,  $t$  can be unified with  $t'$ .

The second way to activate a link- $\eta$  is when the rhs can  $\eta$ -reduced to a term  $t$  with rigid head, i.e.  $t'$  is not a  $\Diamond\eta$ . This means that we can unify lhs with  $t'$ .

Once a link- $\eta$  is activated, it can be removed from  $\mathbb{L}$ , otherwise, the link is kept for a further iteration of progress. Note that this link progression enforce invariants 1, 3 and 5: we never commit a term in the  $\mathcal{H}_o$  substitution, since we make unification only when we know that the terms are no more  $\Diamond\eta$ , and when lhs is no more a variable or rhs is no more a  $\Diamond\eta$ , the link is removed from  $\mathbb{L}$ .

TODO: example for case 1:  $\lambda x.\lambda y.F y x = f$  TODO: example for case 2:  $\lambda x.\lambda y.F y x = G, G = \lambda x.\lambda y.a$

A second way to progress link- $\eta$ , that we call link- $\eta$  deduplication, is when  $\mathbb{L}$  contains two link- $\eta$   $l_1$  and  $l_2$  with a lhs having the same variable address. Let the lhs of  $l_1$  be  $\text{uva } U R$  and the lhs of  $l_2$  be  $\text{uva } V S$ , then, by invariant 2,  $R$  and  $S$  have same scope. Let  $t$  be the term obtained by replacing all each name  $S_i$  in the rhs of  $l_1$  with  $R_i$ ,  $t$  is unified with the rhs of  $l_2$  and one of the two links between  $l_1$  and  $l_2$  is removed from  $\mathbb{L}$ .

TODO: example for this:  $\lambda x.\lambda y.F y x = X, \lambda x.\lambda y.F y x = Y$

We can note that the insertion of these rules for progress1 do not prevent the termination of progress, since: if a link is activated it is removed from  $\mathbb{L}$  and the recursive call to progress will have a smaller list of links to recurse on. Moreover, link activation only runs terminating instructions (such as unification). If a link is deduplicated, the termination of progress is still guaranteed since again we reduce  $\mathbb{L}$  and the instructions run by link deduplications are all terminating. Finally, if a link is neither activated nor deduplicated, i.e. it remains suspended, then the condition of the if-statement of progress succeeds making it to terminate.

TODO: we can have  $\lambda x.F_x$  in the substitution if we know that  $F$  does not reduce to  $Tx$  where  $x$  is not free in  $T$ .

## 7 ENFORCING INVARIANT ??

In section 5.5, we have given two unification problems to be run one after the other. The compilation of each term would generate the following result:

**Unif problems:**  $X = \lambda x.\lambda y.x, Z = f \cdot (\lambda x.x)$

**Mappings:**  $F \mapsto G^1, F \mapsto H^2$

**Links:**  $x \vdash Y_x =_\eta \lambda y.H_y x,$

$\vdash X =_\eta \lambda x.Y_x,$

$\vdash Z =_\eta \lambda x.f G_x x$

We see that the maybe-eta as detected  $\lambda xy.F y x$  and  $\lambda x.f (F x) x$  and replaced them with respectively the  $\mathcal{H}_o$  vars  $X$  and  $Z$ .  $X$  is linked with  $\lambda x.Y_x$ ,  $Y$  has arity 1 and is  $\eta$ -linked with  $\lambda y.H y x$  and  $Z$  is linked to the term  $\lambda x.f G_x x$ . However, we can remark that this compilation breaks invariant 4, since

Deduplicate mapping code etc...

## 8 HANDLING OF $\diamond\beta$

$\beta$ -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  $\mathcal{L}_\lambda$ . This unification problem is not solvable without loss of generality, since there is not a most general unifier. If we take back the example given in section 2.1, the unification  $Fa = a$  admits two solutions for  $F$ :  $\rho_1 = \{F \mapsto \lambda x.x\}$  and  $\rho_2 = \{F \mapsto \lambda_.a\}$ . Despite this, it is possible to work with  $\diamond\beta$  if an oracle provides a substitution  $\rho$  such that  $\rho t$  falls again in the  $\mathcal{L}_\lambda$ .

On the other hand, the  $\approx_\lambda$  is not designed to understand how the  $\beta$ -redexes work in the object language. Therefore, even if we know that  $F$  is assigned to  $\lambda x.x$ ,  $\approx_\lambda$  is not able to unify  $Fa$  with  $a$ . On the other hand, the problem  $Fa = G$  is solvable by  $\approx_\lambda$ , but the final result is that  $G$  is assigned to  $(\lambda x.x)a$  which breaks the invariant saying that the substitution of the meta language 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  $\beta$ -redex is replaced with a hole  $h$  and a new dedicated link, called link- $\beta$ .

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

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

### 8.1 Compilation

*Detection of  $\diamond\beta$ .* TODO: ...

*Compilation with link- $\beta$ .* In order to build a link- $\beta$ , 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,
  m-alloc (fv A) (hv B (arity Arity)) M2 M3 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  $fapp[fuva A|Ag]$  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- $\beta$  to return in the end is given by the term  $Beta = app[uva B Scope1 | Extra1]$  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  $\mathcal{L}_\lambda$ .

### 8.2 Progress

Once created, there exist two main situations waking up a suspended link- $\beta$ . The former is strictly connected to the definition of  $\beta$ -redex and occurs when the head of  $rhs$  is materialized by the oracle (see proposition 2.1). In this case  $rhs$  is safely  $\beta$ -reduced to a new term  $t'$  and the result can be unified with  $lhs$ . In this scenario the link- $\beta$  has accomplished its goal and can be removed from  $\mathbb{L}$ .

The second circumstance making the link- $\beta$  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  $\mathcal{L}_\lambda$ ; we call  $Extra2$  the suffix of  $Extra1$  such that the concatenation of  $Scope1$  and  $Extra1$  is the same as the concatenation of  $Scope2$  and  $Extra2$ . Finally, two cases should be considered: 1)  $Extra2$  is the empty list,  $lhs$  and  $rhs$  can be unified: we have two terms in  $\mathcal{L}_\lambda$ ; otherwise 2) the link- $\beta$  in question is replaced with a refined version where the  $rhs$  is  $app[uva C Scope2 | Extra2]$  and a new link- $\eta$  is added between the  $lhs$  and the new-added variable  $C$ .

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

```
f = flam x\ fapp[F, fapp[A, x]].
```

The compilation of these terms produces the new unification problem:  $f = X0$

We obtain the mappings  $F \mapsto \mathbf{F}^0$ ,  $A \mapsto \mathbf{A}^1$  and the links:

$$c0 \vdash X3_{c0} =_\beta X2 X1_{c0} \quad (11)$$

$$\vdash X0 =_\eta \lambda c0.X3_{c0} \quad (12)$$

where the first link is a link- $\eta$  between the variable  $X0$ , representing the right side of the unification problem (it is a  $\diamond\eta$ ) and  $X3$ ; and a link- $\beta$  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 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  $x \vdash X3 =_\beta X2 x a$ . The  $rhs$  of the link has now a variable which is partially in the PF, we can therefore remove the original link- $\beta$  and replace it with the following couple on links:

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

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

### 8.3 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,
% ].
```

## 9 FIRST ORDER APPROXIMATION

**TODO:** Coq can solve this:  $f \ 1 \ 2 = x \ 2$ , by setting  $x$  to  $f \ 1$   
**TODO:** We can re-use part of the algo for  $\beta$  given before

## 10 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

## 11 RESULTS: STDPP AND TLC

**TODO:** How may rule are we solving?  
**TODO:** Can we do some perf test

## 12 CONCLUSION

## REFERENCES

- [1] Arthur Charguéraud. “The Optimal Fixed Point Combinator”. In: *Interactive Theorem Proving*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 195–210. ISBN: 978-3-642-14052-5.
- [2] Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. “Implementing HOL in an Higher Order Logic Programming Language”. In: *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. LFMTTP '16. Porto, Portugal: Association for Computing Machinery, 2016. ISBN: 9781450347778. DOI: 10.1145/2966268.2966272. URL: <https://doi.org/10.1145/2966268.2966272>.
- [3] Cvetan Dunchev et al. “ELPI: Fast, Embeddable,  $\lambda$ Prolog Interpreter”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis et al. Vol. 9450. 2015, pp. 460–468. DOI: 10.1007/978-3-662-48899-7\_32. URL: [http://dx.doi.org/10.1007/978-3-662-48899-7\\_32](http://dx.doi.org/10.1007/978-3-662-48899-7_32).
- [4] Amy Felty. “Encoding the Calculus of Constructions in a Higher-Order Logic”. In: ed. by M. Vardi. IEEE, June 1993, pp. 233–244. DOI: 10.1109/LICS.1993.287584.
- [5] Amy Felty and Dale Miller. “Specifying theorem provers in a higher-order logic programming language”. In: *Ninth International Conference on Automated Deduction*. Ed. by Ewing Lusk and Ross Overbeck. 310. Argonne, IL: Springer, May 1988, pp. 61–80. DOI: 10.1007/BFb0012823.
- [6] Davide Fissore and Enrico Tassi. “A new Type-Class solver for Coq in Elpi”. In: *The Coq Workshop 2023*. Bialystok, Poland, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- [7] Benjamin Grégoire, Jean-Christophe L  chenet, and Enrico Tassi. “Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin’s data types with Coq-Elpi”. In: *CPP ’23: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. Boston MA USA, France: ACM, Jan. 2023, pp. 167–181. DOI: 10.1145/3573105.3575683. URL: <https://inria.hal.science/hal-03800154>.
- [8] RALF JUNG et al. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. In: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151.
- [9] Dale Miller. “Unification under a mixed prefix”. In: *Journal of Symbolic Computation* 14.4 (1992), pp. 321–358. DOI: 10.1016/0747-7171(92)90011-R.
- [10] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. DOI: 10.1017/CBO9781139021326.
- [11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7.
- [12] Lawrence C. Paulson. “Set theory for verification. I: from foundations to functions”. In: *J. Autom. Reason.* 11.3 (Dec. 1993), pp. 353–389. ISSN: 0168-7433. DOI: 10.1007/BF00881873. URL: <https://doi.org/10.1007/BF00881873>.
- [13] F. Pfening. “Elf: a language for logic definition and verified metaprogramming”. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, California, USA: IEEE Press, 1989, pp. 313–322. ISBN: 0818619546.
- [14] Frank Pfenning and Carsten 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  $\lambda$ Prolog dialect)”. In: *The Fourth International Workshop on Coq for Programming Languages*. Los Angeles (CA), United States, Jan. 2018. URL: <https://inria.hal.science/hal-01637063>.
- [18] The Coq Development Team. *The Coq Reference Manual – Release 8.18.0*. <https://coq.inria.fr/doc/V8.18.0/refman>. 2023.
- [19] P. Wadler and S. Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. DOI: 10.1145/75277.75283. URL: <https://doi.org/10.1145/75277.75283>.
- [20] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. “The Isabelle Framework”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, C  sar Mu   oz, and

1277	Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7.	1335
1278		1336
1279		1337
1280		1338
1281		1339
1282		1340
1283		1341
1284		1342
1285		1343
1286		1344
1287		1345
1288		1346
1289		1347
1290		1348
1291		1349
1292		1350
1293		1351
1294		1352
1295		1353
1296		1354
1297		1355
1298		1356
1299		1357
1300		1358
1301		1359
1302		1360
1303		1361
1304		1362
1305		1363
1306		1364
1307		1365
1308		1366
1309		1367
1310		1368
1311		1369
1312		1370
1313		1371
1314		1372
1315		1373
1316		1374
1317		1375
1318		1376
1319		1377
1320		1378
1321		1379
1322		1380
1323		1381
1324		1382
1325		1383
1326		1384
1327		1385
1328		1386
1329		1387
1330		1388
1331		1389
1332		1390
1333		1391
1334		1392



## APPENDIX

This appendix contains the entire code described in this paper. The code can also be accessed at the URL: <https://github.com/FissoreD/paper-ho>

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

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

## 13 THE MEMORY

```

kind addr type.
type addr nat -> addr.
typeabbrev (mem A) (list (option A)).

type set? addr -> mem A -> A -> o.
set? (addr A) Mem Val :- get A Mem Val.

type unset? addr -> mem A -> o.
unset? Addr Mem :- not (set? Addr Mem _).

type assign-aux nat -> mem A -> A -> mem A -> 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 addr -> mem A -> A -> mem A -> o.
assign (addr A) Mem1 Val Mem2 :- assign-aux A Mem1 Val Mem2.

type get nat -> mem A -> A -> o.
get z (some Y :: _) Y.
get (s N) (_ :: L) X :- get N L X.

type alloc-aux nat -> mem A -> mem A -> 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 addr -> mem A -> mem A -> o.
alloc (addr A as Ad) Mem1 Mem2 :- unset? Ad Mem1,
  alloc-aux A Mem1 Mem2.

type new-aux mem A -> nat -> mem A -> o.
new-aux [] z [none].
new-aux [A | As] (s N) [A | Bs] :- new-aux As N Bs.

type new mem A -> addr -> mem A -> o.
new Mem1 (addr Ad) Mem2 :- new-aux Mem1 Ad Mem2.

```

## 14 THE OBJECT LANGUAGE

```

kind fm type.
type fapp list fm -> fm.
type flam (fm -> fm) -> fm.
type fcon string -> fm.
type fuva addr -> fm.

typeabbrev fsubst (mem fm).

```

```

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

type fderef fsubst -> fm -> fm -> o. (ps)
fderef S T T2 :- fder S T T1, napp T1 T2.

type napp fm -> fm -> 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) :- map napp L L1.

type (=o) fm -> fm -> o. (=o)
fcon X =o fcon X.
fapp A =o fapp B :- forall2 (=o) A B.
flam F =o flam G :- pi x\ x =o x => F x =o G x.
fuva N =o fuva N.
flam F =o T :- (eta)
  pi x\ beta T [x] (T' x), x =o x => F x =o T' x.
T =o flam F :- (eta_r)
  pi x\ beta T [x] (T' x), x =o x => T' x =o F x.
fapp [flam X | L] =o T :- beta (flam X) L R, R =o T. (beta_l)
T =o fapp [flam X | L] :- beta (flam X) L R, T =o R. (beta_r)

type extend-subst fm -> fsubst -> fsubst -> 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 -> list fm -> fm -> 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 (fcon [H | L]).
beta N L (fapp [N | L]) :- name N.

type mk-app fm -> list fm -> fm -> o.
mk-app T L S :- beta T L S.

type eta-contract fm -> fm -> o.
eta-contract (fcon X) (fcon X).
eta-contract (fapp L) (fapp L1) :- map 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).

```

## 15 THE META LANGUAGE

```

1509 eta-contract (fuva X) (fuva X).
1510 eta-contract X X := name X.
1511
1512 type eta-contract-aux list fm -> fm -> fm -> o.
1513 eta-contract-aux L (flam F) T :-
1514   pi x\ eta-contract-aux [x|L] (F x) T. % also checks H Prefix does not x
1515 eta-contract-aux L (fapp [H|Args]) T :-
1516   rev L LRev, append Prefix LRev Args,
1517   if (Prefix = []) (T = H) (T = fapp [H|Prefix]).
1518
1519
1520
1521 kind inctx type -> type.
1522 type abs (tm -> inctx A) -> inctx A.
1523 type val A -> inctx A.
1524 typeabbrev assignment (inctx tm).
1525 typeabbrev subst (mem assignment).
1526
1527 kind tm type.
1528 type app list tm -> tm.
1529 type lam (tm -> tm) -> tm.
1530 type con string -> tm.
1531 type uva addr -> list tm -> tm.
1532
1533 type (≈λ) tm -> tm -> subst -> subst -> o.
1534 (con C ≈λ con C) S S.
1535 (app L1 ≈λ app L2) S S1 :- fold2 (≈λ) L1 L2 S S1.
1536 (lam F1 ≈λ lam F2) S S1 :-
1537   pi x\ (pi S\ (x ≈λ x) S S) => (F1 x ≈λ F2 x) S S1.
1538 (uva N Args ≈λ T) S S1 :-
1539   set? N S F,!, move F Args T1, (T1 ≈λ T) S S1.
1540 (T ≈λ uva N Args) S S1 :-
1541   set? N S F,!, move F Args T1, (T ≈λ T1) S S1.
1542 (uva M A1 ≈λ uva N A2) S1 S2 :- !,
1543   pattern-fragment A1, pattern-fragment A2,
1544   prune! M A1 N A2 S1 S2.
1545 (uva N Args ≈λ T) S S1 :- not_occ N S T, pattern-fragment Args,
1546   bind T Args T1, assign N S T1 S1.
1547 (T ≈λ uva N Args) S S1 :- not_occ N S T, pattern-fragment Args,
1548   bind T Args T1, assign N S T1 S1.
1549
1550 type prune! addr -> list tm -> addr ->
1551   list tm -> subst -> subst -> o.
1552 /* no pruning needed */
1553 prune! N A N A S S :- !.
1554 prune! M A N A S1 S2 :- !, bind (uva M A) A Ass,
1555   assign N S1 Ass S2.
1556 /* prune different arguments */
1557 prune! N A1 N A2 S1 S3 :- !,
1558   new S1 W S2, prune-same-variable W A1 A2 [] Ass,
1559   assign N S2 Ass S3.
1560 /* prune to the intersection of scopes */
1561 prune! N A1 M A2 S1 S4 :- !,
1562   new S1 W S2, prune-diff-variables W A1 A2 Ass1 Ass2,
1563   assign N S2 Ass1 S3,
1564   assign M S3 Ass2 S4.
1565
1566
1567 type prune-same-variable addr -> list tm -> list tm ->
1568   list tm -> assignment -> o.
1569 prune-same-variable N [] [] ACC (val (uva N Args)) :-
1570   rev ACC Args.
1571 prune-same-variable N [X|XS] [X|YS] ACC (abs F) :-
1572   pi x\ prune-same-variable N XS YS [x|ACC] (F x).
1573 prune-same-variable N [_|XS] [_|YS] ACC (abs F) :-
1574   pi x\ prune-same-variable N XS YS ACC (F x).
1575
1576 type permute list nat -> list tm -> list tm -> o.
1577 permute [] _ [].
1578 permute [P|PS] Args [T|TS] :-
1579   nth P Args T,
1580   permute PS Args TS.
1581
1582 type build-perm-assign addr -> list tm -> list bool ->
1583   list nat -> assignment -> o.
1584 build-perm-assign N ArgsR [] Perm (val (uva N PermutedArgs)) :-
1585   rev ArgsR Args, permute Perm Args PermutedArgs.
1586 build-perm-assign N Acc [tt|L] Perm (abs T) :-
1587   pi x\ build-perm-assign N [x|Acc] L Perm (T x).
1588 build-perm-assign N Acc [ff|L] Perm (abs T) :-
1589   pi x\ build-perm-assign N Acc L Perm (T x).
1590
1591 type keep list A -> A -> bool -> o.
1592 keep L A tt :- mem L A, !.
1593 keep _ _ ff.
1594
1595 type prune-diff-variables addr -> list tm -> list tm ->
1596   assignment -> assignment -> o.
1597 prune-diff-variables N Args1 Args2 Ass1 Ass2 :-
1598   map (keep Args2) Args1 Bits1,
1599   map (keep Args1) Args2 Bits2,
1600   filter Args1 (mem Args2) ToKeep1,
1601   filter Args2 (mem Args1) ToKeep2,
1602   map (index ToKeep1) ToKeep1 IdPerm,
1603   map (index ToKeep1) ToKeep2 Perm21,
1604   build-perm-assign N [] Bits1 IdPerm Ass1,
1605   build-perm-assign N [] Bits2 Perm21 Ass2.
1606
1607 type beta tm -> list tm -> tm -> o.
1608 beta A [] A.
1609 beta (lam Bo) [H | L] R :- beta (Bo H) L R.
1610 beta (app A) L (app X) :- append A L X.
1611 beta (con H) L (app [con H | L]).
1612 beta X L (app[X|L]) :- name X.
1613
1614 /* occur check for N before crossing a functor */
1615 type not_occ addr -> subst -> tm -> o.
1616 not_occ N S (uva M Args) :- set? M S F,
1617   move F Args T, not_occ N S T.
1618 not_occ N S (uva M Args) :- unset? M S, not (M = N),
1619   forall1 (not_occ_aux N S) Args.
1620 not_occ _ _ (con _).
1621 not_occ N S (app L) :- not_occ_aux N S (app L).
1622 /* Note: lam is a functor for the meta language! */
1623 not_occ N S (lam L) :- pi x\ not_occ_aux N S (L x).
1624

```

```

1625 not_occ _ _ X := name X.
1626 /* finding N is ok */
1627 not_occ N _ (uva N _).
1628
1629 /* occur check for X after crossing a functor */
1630 type not_occ_aux addr -> subst -> tm -> o.
1631 not_occ_aux N S (uva M _) := unset? M S, not (N = M).
1632 not_occ_aux N S (uva M Args) := set? M S F,
1633   move F Args T, not_occ_aux N S T.
1634 not_occ_aux N S (app L) := forall1 (not_occ_aux N S) L.
1635 not_occ_aux N S (lam F) := pi x\ not_occ_aux N S (F x).
1636 not_occ_aux _ _ (con _).
1637 not_occ_aux _ _ X := name X.
1638 /* finding N is ko, hence no rule */
1639
1640 /* copy T T' vails if T contains a free variable, i.e. it
1641   performs scope checking for bind */
1642 type copy tm -> tm -> o.
1643 copy (con C) (con C).
1644 copy (app L) (app L') := map copy L L'.
1645 copy (lam T) (lam T') := pi x\ copy x x => copy (T x) (T' x).
1646 copy (uva A L) (uva A L') := map copy L L'.
1647
1648 type bind tm -> list tm -> assignment -> o.
1649 bind T [] (val T') := copy T T'.
1650 bind T [X | TL] (abs T') := pi x\ copy X x => bind T TL (T' x).
1651
1652 type deref subst -> tm -> tm -> o. (σt)
1653 deref _ (con C) (con C).
1654 deref S (app A) (app B) := map (deref S) A B.
1655 deref S (lam F) (lam G) :=
1656   pi x\ deref S x x => deref S (F x) (G x).
1657 deref S (uva N L) R := set? N S A,
1658   move A L T, deref S T R.
1659 deref S (uva N A) (uva N B) := unset? N S,
1660   map (deref S) A B.
1661
1662 type move assignment -> list tm -> tm -> o.
1663 move (abs Bo) [H|L] R := move (Bo H) L R.
1664 move (val A) [] A.
1665
1666 type deref-assmt subst -> assignment -> assignment -> o.
1667 deref-assmt S (abs T) (abs R) := pi x\ deref-assmt S (T x) (R x).
1668 deref-assmt S (val T) (val R) := deref S T R.
1669
1670
1671 16 THE COMPILER
1672
1673 kind arity type.
1674 type arity nat -> arity.
1675
1676 kind fvariable type.
1677 type fv addr -> fvariable.
1678
1679 kind hvariable type.
1680 type hv addr -> arity -> hvariable.
1681
1682
1683 kind mapping type.
1684 type mapping fvariable -> hvariable -> mapping.
1685 typeabbrev mmap (list mapping).
1686
1687 typeabbrev scope (list tm).
1688 typeabbrev inctx ho.inctx.
1689 kind baselink type.
1690 type link-eta tm -> tm -> baselink.
1691 type link-beta tm -> tm -> baselink.
1692 typeabbrev link (inctx baselink).
1693 typeabbrev links (list link).
1694
1695 macro @val-link-eta T1 T2 := ho.val (link-eta T1 T2).
1696 macro @val-link-beta T1 T2 := ho.val (link-beta T1 T2).
1697
1698
1699 type occurs-rigidly fm -> fm -> o.
1700 occurs-rigidly N N.
1701 occurs-rigidly _ (fapp [fuva _|_]) := !, fail.
1702 occurs-rigidly N (fapp L) := exists (occurs-rigidly N) L.
1703 occurs-rigidly N (flam B) := pi x\ occurs-rigidly N (B x).
1704
1705 type reducible-to list fm -> fm -> fm -> o.
1706 reducible-to _ N N := !.
1707 reducible-to L N (fapp [fuva _|Args]) := !,
1708   forall1 (x\ exists (reducible-to [] x) Args) [N|L].
1709 reducible-to L N (flam B) := !,
1710   pi x\ reducible-to [x | L] N (B x).
1711 reducible-to L N (fapp [N|Args]) :=
1712   last-n {len L} Args R,
1713   forall2 (reducible-to []) R {rev L}.
1714
1715 type maybe-eta fm -> list fm -> o. (◇η)
1716 maybe-eta (fapp [fuva _|Args]) L := !,
1717   forall1 (x\ exists (reducible-to [] x) Args) L, !.
1718 maybe-eta (flam B) L := !, pi x\ maybe-eta (B x) [x | L].
1719 maybe-eta (fapp [fcon _|Args]) L :=
1720   split-last-n {len L} Args First Last,
1721   none (x\ exists (y\ occurs-rigidly x y) First) L,
1722   forall2 (reducible-to []) {rev L} Last.
1723
1724 type locally-bound tm -> o.
1725 type get-scope-aux tm -> list tm -> o.
1726 get-scope-aux (con _) [].
1727 get-scope-aux (uva _ L) L1 :=
1728   forall2 get-scope-aux L R,
1729   flatten R L1.
1730 get-scope-aux (lam B) L1 :=
1731   pi x\ locally-bound x => get-scope-aux (B x) L1.
1732 get-scope-aux (app L) L1 :=
1733   forall2 get-scope-aux L R,
1734   flatten R L1.
1735 get-scope-aux X [X] := name X, not (locally-bound X).
1736 get-scope-aux X [] := name X, (locally-bound X).
1737
1738
1739
1740

```

```

1741 type names1 list tm -> o.
1742 names1 L :-
1743   names L1,
1744   new_int N,
1745   if (1 is N mod 2) (L1 = L) (rev L1 L).
1746
1747 type get-scope tm -> list tm -> o.
1748 get-scope T Scope :-
1749   get-scope-aux T ScopeDuplicata,
1750   undup ScopeDuplicata Scope.
1751 type rigid fm -> o.
1752 rigid X :- not (X = fuva _).
1753
1754 type comp-lam (fm -> fm) -> (tm -> tm) ->
1755   mmap -> mmap -> links -> links -> subst -> subst -> o.
1756 comp-lam F G M1 M2 L1 L3 S1 S2 :-
1757   pi x y\ (pi M L S\ comp x y M M L L S S) =>
1758     comp (F x) (G y) M1 M2 L1 (L2 y) S1 S2,
1759     close-links L2 L3.
1760
1761 type close-links (tm -> links) -> links -> o.
1762 close-links (\[_][]) [].
1763 close-links (v\[X |L v]) [X|R] :- !, close-links L R.
1764 close-links (v\[X v|L v]) [abs X|R] :- close-links L R.
1765 type comp fm -> tm -> mmap -> mmap -> links -> links ->
1766   subst -> subst -> o.
1767 comp (fcon C) (con C) M M L L S S.
1768 comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
1769   maybe-eta (flam F) [], !,
1770   alloc S1 A S2,
1771   comp-lam F F1 M1 M2 L1 L2 S2 S3,
1772   get-scope (lam F1) Scope,
1773   L3 = [val (link-eta (uva A Scope) (lam F1)) | L2].
1774 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :- (cλ)
1775   comp-lam F F1 M1 M2 L1 L2 S1 S2.
1776 comp (fuva A) (uva B []) M1 M2 L L S1 S2 :-
1777   m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
1778 comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
1779   pattern-fragment Ag, !,
1780   fold6 comp Ag Ag1 M1 M1 L L S1 S1,
1781   len Ag Arity,
1782   m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
1783 comp (fapp [fuva A|Ag]) (uva C Scope) M1 M3 L1 L3 S1 S4 :- !,
1784   pattern-fragment-prefix Ag Pf Extra,
1785   fold6 comp Pf Scope1 M1 M1 L1 L1 S1 S1,
1786   fold6 comp Extra Extra1 M1 M2 L1 L2 S1 S2,
1787   len Pf Arity,
1788   m-alloc (fv A) (hv B (arity Arity)) M2 M3 S2 S3,
1789   Beta = app [uva B Scope1 | Extra1],
1790   get-scope Beta Scope,
1791   alloc S3 C S4,
1792   L3 = [eval-link-beta (uva C Scope) Beta | L2].
1793 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :-
1794   fold6 comp A A1 M1 M2 L1 L2 S1 S2.
1795
1796 type alloc mem A -> addr -> mem A -> o.
1797 alloc S N S1 :- mem.new S N S1.

```

```

type compile-terms-diagnostic
triple diagnostic fm fm ->
triple diagnostic tm tm ->
mmap -> mmap ->
links -> links ->
subst -> subst -> o.
compile-terms-diagnostic (triple D F01 F02) (triple D H01 H02) M1 M3 L1
comp F01 H01 M1 M2 L1 L2 S1 S2,
comp F02 H02 M2 M3 L2 L3 S2 S3.

type compile-terms
list (triple diagnostic fm fm) ->
list (triple diagnostic tm tm) ->
mmap -> links -> subst -> o.
compile-terms T H M L S :-
fold6 compile-terms-diagnostic T H [] M_ [] L_ [] S_,
deduplicate-map M_ M S_ S L_ L.

type make-eta-link-aux nat -> addr -> addr ->
list tm -> links -> subst -> subst -> o.
make-eta-link-aux z Ad1 Ad2 Scope1 L H1 H1 :-
rev Scope1 Scope, eta-expand (uva Ad2 Scope) T1,
L = [eval-link-eta (uva Ad1 Scope) T1].
make-eta-link-aux (s N) Ad1 Ad2 Scope1 L H1 H3 :-
rev Scope1 Scope, alloc H1 Ad H2,
eta-expand (uva Ad Scope) T2,
(pi x\ make-eta-link-aux N Ad Ad2 [x|Scope1] (L1 x) H2 H3),
close-links L1 L2,
L = [eval-link-eta (uva Ad1 Scope) T2 | L2].

type make-eta-link nat -> nat -> addr -> addr ->
list tm -> links -> subst -> subst -> o.
make-eta-link (s N) z Ad1 Ad2 Vars L H H1 :-
make-eta-link-aux N Ad2 Ad1 Vars L H H1.
make-eta-link z (s N) Ad1 Ad2 Vars L H H1 :-
make-eta-link-aux N Ad1 Ad2 Vars L H H1.
make-eta-link (s N) (s M) Ad1 Ad2 Vars Links H H1 :-
(pi x\ make-eta-link N M Ad1 Ad2 [x|Vars] (L x) H H1),
close-links L Links.

type deduplicate-map mmap -> mmap ->
subst -> subst -> links -> links -> o.
deduplicate-map [] [] H H L L.
deduplicate-map [(mapping (fv 0) (hv M (arity LenM))) as X1] | Map1 Map2
take-list Map1 (mapping (fv 0) (hv M' (arity LenM')))) _, !,
std.assert! (not (LenM = LenM')) "Deduplicate map, there is a bug",
print "arity-fix links:" {ppmapping X1} "~!~" {ppmapping (mapping (fv
make-eta-link LenM LenM' M M' [] New H1 H2,
print "new eta link" {pplinks New},
append New L1 L2,
deduplicate-map Map1 Map2 H2 H3 L2 L3.
deduplicate-map [A|As] [A|Bs] H1 H2 L1 L2 :-
deduplicate-map As Bs H1 H2 L1 L2, !.
deduplicate-map [A|_] _ H _ _ _ :-
halt "deduplicating mapping error" {ppmapping A} {ho.ppsubst H1}

```



## 17 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) S S2 [eval-link-beta T1 T2] :- !,
```

```
  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 S1) A) :- !,
```

```
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 progress1-links -> links -> ho.subst -> ho.subst -> o.
```

```
progress1 [] [] X X.
```

```

1973 progress1 [A|L1] [A|L3] S S2 :- take-link A L1 B L2, !,
1974   same-link-eta A B S S1,
1975   progress1 L2 L3 S1 S2.
1976 progress1 [L0|L1] L3 S S2 :- deref-link S L0 L,
1977   solve-link-abs L R S S1, !,
1978   progress1 L1 L2 S1 S2, append R L2 L3.

```

## 18 THE DECOMPIER

```

1982 type abs->lam ho.assignment -> ho.tm -> o.
1983 abs->lam (ho.abs T) (ho.lam R) :- !, pi x\ abs->lam (T x) (R x).
1984 abs->lam (ho.val A) A.

```

```

1986 type commit-links-aux link -> ho.subst -> ho.subst -> o.
1987 commit-links-aux (@val-link-eta T1 T2) H1 H2 :-
1988   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
1989   (T1' ==1 T2') H1 H2.
1990 commit-links-aux (@val-link-beta T1 T2) H1 H2 :-
1991   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
1992   (T1' ==1 T2') H1 H2.
1993 commit-links-aux (ho.abs B) H H1 :-
1994   pi x\ commit-links-aux (B x) H H1.

```

```

1996 type commit-links links -> links -> ho.subst -> ho.subst -> o.
1997 commit-links [] [] H H.
1998 commit-links [Abs | Links] L H H2 :-
1999   commit-links-aux Abs H H1, !, commit-links Links L H1 H2.

```

```

2001 type decomp1-subst map -> map -> ho.subst ->
2002   fo.fsubst -> fo.fsubst -> o.
2003 decomp1-subst _ [A|_] _ _ :- fail.
2004 decomp1-subst _ [] _ F F.
2005 decomp1-subst Map [mapping (fv V0) (hv VM _)|T1] H F F2 :-
2006   mem.set? VM H T, !,
2007   ho.deref-assmt H T TTT,
2008   abs->lam TTT T', tm->fm Map T' T1,
2009   fo.eta-contract T1 T2, mem.assign V0 F T2 F1,
2010   decomp1-subst Map T1 H F1 F2.
2011 decomp1-subst Map [mapping _ (hv VM _)|T1] H F F2 :-
2012   mem.unset? VM H, decomp1-subst Map T1 H F F2.

```

```

2014 type tm->fm map -> ho.tm -> fo.fm -> o.
2015 tm->fm _ (ho.con C) (fo.fcon C).
2016 tm->fm L (ho.lam B1) (fo.flam B2) :-
2017   pi x y\ tm->fm _ x y => tm->fm L (B1 x) (B2 y).
2018 tm->fm L (ho.app L1) T :- map (tm->fm L) L1 [Hd|T1],
2019   fo.mk-app Hd T1 T.
2020 tm->fm L (ho.uva VM TL) T :- mem L (mapping (fv V0) (hv VM _)),
2021   map (tm->fm L) TL T1, fo.mk-app (fo.fuva V0) T1 T.
2022
2023 type add-new-map-aux ho.subst -> list ho.tm -> map ->
2024   map -> fo.fsubst -> fo.fsubst -> o.
2025 add-new-map-aux _ [] _ [] S S.
2026 add-new-map-aux H [T|Ts] L L2 S S2 :-
2027   add-new-map H T L L1 S S1,
2028   add-new-map-aux H Ts L1 L2 S1 S2.

```

```

2031 type add-new-map ho.subst -> ho.tm -> map ->
2032   map -> fo.fsubst -> fo.fsubst -> o.
2033 add-new-map _ (ho.uva N _) Map [] F1 F1 :-
2034   mem Map (mapping _ (hv N _)), !.
2035 add-new-map H (ho.uva N L) Map [Map1 | MapL] F1 F3 :-
2036   mem.new F1 M F2,
2037   len L Arity, Map1 = mapping (fv M) (hv N (arity Arity)),
2038   add-new-map H (ho.app L) [Map1 | Map] MapL F2 F3.
2039 add-new-map H (ho.lam B) Map NewMap F1 F2 :-
2040   pi x\ add-new-map H (B x) Map NewMap F1 F2.
2041 add-new-map H (ho.app L) Map NewMap F1 F3 :-
2042   add-new-map-aux H L Map NewMap F1 F3.
2043 add-new-map _ (ho.con _) _ [] F F :- !.
2044 add-new-map _ N _ [] F F :- name N.
2045

```

```

2046 type complete-mapping-under-ass ho.subst -> ho.assignment ->
2047   map -> map -> fo.fsubst -> fo.fsubst -> o.
2048 complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
2049   add-new-map H Val Map1 Map2 F1 F2.
2050 complete-mapping-under-ass H (ho.abs Abs) Map1 Map2 F1 F2 :-
2051   pi x\ complete-mapping-under-ass H (Abs x) Map1 Map2 F1 F2.
2052
2053 type complete-mapping ho.subst -> ho.subst ->
2054   map -> map -> fo.fsubst -> fo.fsubst -> o.
2055 complete-mapping _ [] L L F F.
2056 complete-mapping H [none | T1] L1 L2 F1 F2 :-
2057   complete-mapping H T1 L1 L2 F1 F2.
2058 complete-mapping H [some T0 | T1] L1 L3 F1 F3 :-
2059   ho.deref-assmt H T0 T,
2060   complete-mapping-under-ass H T L1 L2 F1 F2,
2061   append L1 L2 Lall,
2062   complete-mapping H T1 Lall L3 F2 F3.
2063

```

```

2064 type decompile map -> links -> ho.subst ->
2065   fo.fsubst -> fo.fsubst -> o.
2066 decompile Map1 L H0 F0 F02 :-
2067   commit-links L L1_ H0 H01, !,
2068   complete-mapping H01 H01 Map1 Map2 F0 F01,
2069   decomp1-subst Map2 Map2 H01 F01 F02.
2070

```

## 19 AUXILIARY FUNCTIONS

```

2071 type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A ->
2072   list A1 -> B -> B -> C -> C -> o.
2073 fold4 _ [] [] A A B B.
2074 fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0,
2075   fold4 F XS YS A0 A1 B0 B1.
2076
2077 type len list A -> nat -> o.
2078 len [] z.
2079 len [_|L] (s X) :- len L X.
2080

```