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], λProlog [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 $\simeq_o$ using the ML Elpi [3], a dialect of λProlog. Elpi's equational theory comprises $\eta\beta$ equivalence and comes equipped with a higher order unification procedure $\simeq_\lambda$ restricted to the pattern fragment [9]. We want $\simeq_o$ to be as powerful as $\simeq_\lambda$ 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" $\simeq_\lambda$ 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 $\simeq_\lambda$, effectively implementing $\simeq_o$ on top of $\simeq_\lambda$ 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, 16 pages. https://doi.org/ZZZZZZZZZZZZZ

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], λProlog [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 λProlog [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"]
```

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/ZZZZZZZZZZZZZ

We now illustrate the encoding of the three instances above as higher-order logic-programming rules: capital letters denote rule parameters; `:-` separates the rule's head from the premises; `pi w\ p` introduces a fresh nominal constant `w` for the premise `p`.

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

Unfortunately this translation of rule $(r3)$ uses the predicate `P` as a first order term: for the meta language its type is `tm`. If we try to backchain the rule $(r3)$ on the encoding of the goal (g) given below

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

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

```
app[con"nfact", y, con"3"] = app[P, y]                (p)
```

In this paper we study a more sophisticated encoding of Coq terms allowing us to rephrase the problematic rule $(r3)$ as follows:

```
decision (all A x\ Pm x) :- link Pm P A, finite A,   (r3a)
  pi x\ decision (app[P, x]).
```

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

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

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

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

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

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

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

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

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

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

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

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

`x\ f x`	$\simeq_\lambda$	`f`
`lam A x\ app[con"f", x]`	$\simeq_o$	`con"f"`
`lam A x\ app[con"f", x]`	$\not\simeq_\lambda$	`con"f"`
`P x`	$\simeq_\lambda$	`x`
`app[P, x]`	$\simeq_o$	`x`
`app[P, x]`	$\not\simeq_\lambda$	`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 $\mathcal{F}_o$ representation of DTT terms and a $\mathcal{H}_o$ one. We call $=_o$ the equality over ground terms in $\mathcal{F}_o$, $=_\lambda$ the equality over ground terms in $\mathcal{H}_o$, $\simeq_o$ the unification procedure we want to implement and $\simeq_\lambda$ the one provided by the meta language. TODO extend $=_o$ and $=_\lambda$ with reflexivity on uvars.

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

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

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

We illustrate a compilation $\langle s \rangle \mapsto (t, m, l)$ that maps a term s in $\mathcal{F}_o$ to a term t in $\mathcal{H}_o$, a variable mapping m and list of links l. The variable map connects unification variables in $\mathcal{H}_o$ with variables in $\mathcal{F}_o$ 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 $\mathcal{F}_o$as a list *steps* p of length $\mathcal{N}$. Each made of a unification problem between terms $\mathcal{S}_{p_l}$ and $\mathcal{S}_{p_r}$ taken from the set of all terms $\mathcal{S}$. The composition of these steps starting from the empty substitution ρ_0 produces the final substitution $\rho_{\mathcal{N}}$. [1] The initial here ρ_0 is the empty substitution

$$\text{fstep}(\mathcal{S}, p, \rho) \mapsto \rho'' \stackrel{def}{=\!=} \rho\mathcal{S}_{p_l} \simeq_o \rho\mathcal{S}_{p_r} \mapsto \rho' \wedge \rho'' = \rho \cup \rho'$$

$$\text{frun}(\mathcal{S}, \mathcal{N}) \mapsto \rho_{\mathcal{N}} \stackrel{def}{=\!=} \bigwedge_{p=1}^{\mathcal{N}} \text{fstep}(\mathcal{S}, p, \rho_{p-1}) \mapsto \rho_p$$

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

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

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

Proposition 2.1 (Simulation). $\forall \mathcal{S}, \forall \mathcal{N}$

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

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

Proposition 2.2 (Simulation fidelity). *In the context of* hrun, *if* $\mathcal{T} \subseteq \mathcal{L}_\lambda$ *we have that* $\forall p \in 1 \ldots \mathcal{N}$

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

In particular this property guarantees that a *failure* in the $\mathcal{F}_o$ run is matched by a failure in $\mathcal{H}_o$ *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 $\mathcal{F}_o$ by looking at its execution trace in $\mathcal{H}_o$.

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

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

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

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

Proposition 2.3 (Properties of $\simeq_o$).

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

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

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

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

Property 5 states that $\simeq_o$, hence our compilation scheme, is resilient to unification problems outside $\mathcal{L}_\lambda$ 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 $\mathcal{L}_\lambda$:

```
app [F, con"a"] = app[con"f", con"a", con"a"]      (q)
F = lam x\ app[con"f",x,x]                          (h)
```

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 $h \simeq_\lambda p$. As a result $\simeq_\lambda$ 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 ($\Diamond\eta$). $\Diamond\eta = \{t \mid \exists \rho, \rho t \text{ is an eta expansion}\}$

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

Definition 2.5 ($\Diamond\beta$). $\Diamond\beta = \{Xt_1 \ldots t_n \mid Xt_1 \ldots t_n \notin \mathcal{L}_\lambda\}$.

An example of t in $\Diamond\beta$ is Fa for a constant a. Note however tha 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.

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

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 $X \subseteq \mathcal{H}_o$,

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

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

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

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

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

[1] 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

3 ALTERNATIVE ENCODINGS AND RELATED WORK

Paper [2] introduces semi-shallow.

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

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

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

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

The second reason is the encoding for Coq is used for meta programming the system, hence it must accommodate the manipulation of terms that are now 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 `all` quantifier of DTT we used in the example in Section 1 together with the type notation of terms carried by the `lam` constructor.

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

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

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 `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 `uva N [x]`, since unification variables come equipped with an explicit scope. We say that the unification variable occurrence `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. [2]

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`.

is new used?

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

```
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 map (list mapping).
```

Invariant 1 (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 arity of a variable in $\mathcal{H}_o$ (a `hvariable` is stored in the mapping. In particular `m-alloc` bla bla explain. Multiple mappings for the same `fvariable` are handled in section 6.1.

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

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

4.1 Notations

When we write $\mathcal{H}_o$ terms outside code blocks we prefer to use the usual and more compact mathematical notation. However we need to distinguish between the "application" of a unification variable to its scope and the application of a term (possibly a unification variable) to a list of arguments. We write the scope in subscript and we use juxtaposition for application. Here a few examples:

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

When detailing example we prefer to write links in mathematical notation, as equations between terms in 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"]))
```

4.2 Equational theory and Unification

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

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

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

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

```
type fder subst -> fm -> fm -> o.
fder _ (fcon C) (fcon C).
fder S (fapp A) (fapp B) :- forall2 (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 subst -> fm -> fm -> o.                 (ρs)
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) :- forall2 napp L L1.
```

TODO explain napp. We use the cut operator to keep the code compact. It is possible to rewrite the rule for application traversal so that it is mutually exclusive with the first one, but that requires a rather verbose analysis of the head of A.

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

```
type deref subst -> tm -> tm -> o.                       (σt)
deref _ (con C) (con C).
deref S (app A) (app B) :- forall2 (deref S) A B.
deref S (lam F) (lam G) :-
  pi x\ deref S x x => deref S (F x) (G x).
deref S (uva N L) R :- set? N S A, move A L T, deref S T R.
deref S (uva X A) (uva X B) :- unset? X S, forall2 (deref S) A B.

type move assignment -> list tm -> tm -> o.
move (abs Bo) [H|L] R :- move (Bo H) L R.
move (val A)  [] A :- !.
```

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

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

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

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

```
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 :-                                            (η_l)
  pi x\ beta T [x] (T' x), x =_o x => F x =_o T' x.
T =_o flam F :-                                            (η_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.  (β_l)
T =_o fapp [flam X | L] :- beta (flam X) L R, T =_o R.  (β_r)

type (=_λ) tm -> tm -> o.
app A =_λ fapp B :- map (=_λ) A B.
```

```
lam F =λ flam G :- pi x\ x =λ x => F x =λ G x.
con C =λ fcon C.
uva N A =λ fuva N B :- map (=λ) A B.
```

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

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

```
type (≃λ) tm -> tm -> subst -> subst -> o.
```

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

This predicate has four arguments, the two terms t_1 and t_2 to unify, the old substitution map ρ_1, and the new substitution map ρ_2, with the invariant $\rho_1 \subseteq \rho_2$.

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

In this section we describe a basic compilation scheme that we refine later, in the following sections. This scheme is sufficient to implement an $\simeq_o$ that respects β-conversion for terms in $\mathcal{L}_\lambda$. The extension to $\eta\beta$-conversion is described in Section 6.1 and the support for terms outside $\mathcal{L}_\lambda$ in Section 6.2.

5.1 Compilation

The objective of the compilation is to recognize the higher-order variables available in $\mathcal{H}_o$ when expressed in a first order way in $\mathcal{F}_o$. The compiler also generates a map to bring back the substitution from $\mathcal{H}_o$ to $\mathcal{F}_o$.

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

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 6.1 and 6.2.

```
type comp fm -> tm -> map -> map -> links -> links ->
  subst -> subst -> o.
comp (fcon C) (con C)        M1 M1 L1 L1 S1 S1.
comp (flam F) (lam F1)       M1 M2 L1 L2 S1 S2 :-
  comp-lam F F1 M1 M2 L1 L2 S1 S2.
comp (fuva A) (uva B [])     M1 M2 L L S S1 :-
  m-alloc (fv A) (hv B (arity z)) M1 M2 S S1.
comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L1 L1 S1 S2 :-
  pattern-fragment Ag, !,
    fold6 comp Ag Ag1 M1 M1 L1 L1 S1 S1,
    len Ag Arity,
    m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
comp (fapp A) (app A1)       M1 M2 L1 L2 S1 S2 :-
  fold6 comp A A1 M1 M2 L1 L2 S1 S2.
```

This preliminary version of comp simply recognizes $\mathcal{F}_o$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, see the comp-lam hyp rule.

The auxiliary function `close-links`

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

type close-links (tm -> links) -> links -> o.
close-links (_\[]) [].
close-links (v\[L|XS v]) [L|YS] :- !, close-links XS YS.
close-links (v\[(L v)|XS v]) [abs L|YS] :- !,
  close-links XS YS.
```

since we want links to bubble up we use the abs constructor of the inctx data type to bind back the variable just crossed, and we do so only if the variable v occurs in L.

5.2 Execution

5.3 Decompilation

5.4 Example

OK

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

KO

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

TODO: Goal: $s_1 \simeq_o s_2$ is compiled into $t_1 \simeq_\lambda t_2$

TODO: What is done: uvars `fo_uv` of OL are replaced into uvars `ho_uv` of the ML

TODO: Each `fo_uv` is linked to an `ho_uv` of the OL

TODO: Example needing the compiler v0 (tra l'altro lo scope è ignorato): **`lam x\ app[con"g",app[uv 0, x]]` $\simeq_o$ `lam x\ app[con"g", c"a"]`**

TODO: Links used to instantiate vars of elpi

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

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

The compilation step is meant to recover the higher-order variables of the OL, expressed in a first order way, by replacing them with higher-order variables in the ML. In particular, every time a variable of the OL is encountered in the original term, it is replaced with a meta variable, and if the OL variable is applied to a list of distinct names L, then this list becomes the scope of the variable. For all the other constructors of `tm`, the same term constructor is returned and its arguments are recursively compiled. The predicate in charge for term compilation is:

```
  type comp tm -> tm -> links -> links -> subst -> subst -> o.
```

where, we take the term of the OL, produce the term of the ML, take a list of link and produce a list of new links, take a substitution and return a new substitution.

In particular, due to programming constraints, we need to drag the old subst and return a new one extended, if needed, with the new declared meta-variables.

The following code

```
kind link type.
type link nat -> nat -> nat -> subst.
```

defines a link, which is a relation between to variables indexes, the first being the index of a OL variable and the second being the index of a ML variable. The third integer is the number of term in the scope of the two variables, or equivalently, in a typed language, their arity.

integer or nat?

As an example, let's study the following unification problem (a slightly modified version from section 1):

```
lam x\ app[c"decision", app[c"nfact", x, c"3"]] ≃_o
  lam x\ app [c"decision", app[uv 0, x]]
```

we have the main unification problem where the nested `app` nodes have lists of different lengths making the unification to fail. The compilation of these terms produces a new unification problem with the following shape:

```
lam x\ app[c"decision", app[c"nfact", x, c"3"]] ≃_λ
  lam x\ app [c"decision", uv 1 [x]]
```

The main difference is the replacement of the subterm `app[uv 0, x]` of the OL with the subterm `uv 0 [x]`. Variable indexes are chosen by the ML, that is, the index `0` for that unification variable of the OL term has not the sam meaning of the index `0` in the ML. There exists two different substitution mapping, one for the OL and one for the ML and the indexes of variable point to the respective substitution.

decomp che mappa abs verso lam **TODO: An other example: `lam x\ app[f, app[X, x]] = Y, (lam x\ x) = X`.**

6 USE OF MULTIVARS

Se il termine initziale è della forma

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

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

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

6.1 Problems with η

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

and aux

```
%% x occurs rigidly in t iff ∀σ,∀t',t' =_o σt ⇒ x ∈ 𝒫(t')
%%
type occurs-rigidly fm -> fm -> o.
occurs-rigidly N N.
occurs-rigidly _ (fapp [fuva _|_]) :- !, fail.
occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.
occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).

/* maybe-eta N T L succeeds iff T could be an eta expasions for N, that
%%   is ∃σ,σ(λn.t) = λn.t'n and n
%%   does not occur rigidly in t'
type maybe-eta fm -> fm -> list fm -> o.
maybe-eta N (fapp[fuva _|Args]) _ :- !,
  exists (x\ maybe-eta-of [] N x) Args, !.
maybe-eta N (flam B) L :- !, pi x\ maybe-eta N (B x) [x | L].
maybe-eta _ (fapp [fcon _|Args]) L :-
  split-last-n {len L} Args First Last,
  forall1 (x\ forall1 (y\ not (occurs-rigidly x y)) First) L,
  forall2 (maybe-eta-of []) {rev L} Last.

%%   is ∃σ,σt =_o n
type maybe-eta-of list fm -> fm -> fm -> o.
maybe-eta-of _ N N :- !.
maybe-eta-of L N (fapp[fuva _|Args]) :- !,
  forall1 (x\ exists (maybe-eta-of [] x) Args) [N|L].
maybe-eta-of L N (flam B) :- !,
  pi x\ maybe-eta-of [x | L] N (B x).
maybe-eta-of L N (fapp [N|Args]) :-
  last-n {len L} Args R,
  forall2 (maybe-eta-of []) R {rev L}.
```

TODO: The following goal necessita v1 (lo scope è usato): `X = lam x\ lam y\ Y y x, X = lam x\ f`

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

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

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

```
La deduplicate eta:
- viene chiamata che della forma [variable] -> [eta1] e
↪  [variable] -> [eta2]
  (a destra non c'è mai un termine con testa rigida)
- i due termini a dx vengono unificati con la unif e uno
↪  dei due link viene buttato
  NOTA!! A dx abbiamo sempre un termine della forma lam
  ↪  x.VAR x!!!
  Altrimenti il link sarebbe stato risolto!!
- dopo l'unificazione rimane un link [variabile] -> [etaX]
- nella progress-eta, se a sx abbiamo una constante o
↪  un'app, allora eta-espandiamo
  di uno per poter unificare con il termine di dx.
```

6.2 Problems with β

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

On the other hand, the $\simeq_\lambda$ is not designed to understand how the β-redexes work in the object language. Therefore, even if we know that F is assigned to $\lambda x.x$, $\simeq_\lambda$ is not able to unify Fa with a. On the other hand, the problem $Fa = G$ is solvable by $\simeq_\lambda$, 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 outide $\mathcal{W}$ (Property 2.8).

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

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

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

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

```
comp (fapp [fuva A|Ag]) (uva C Scope) M1 M3 L1 L3 S1 S4 :- !,
  pattern-fragment-prefix Ag Pf Extra,
  fold6 comp Pf    Scope1 M1 M1 L1 L1 S1 S1,
  fold6 comp Extra Extra1 M1 M2 L1 L2 S1 S2,
  len Pf Arity,
  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 = [@val-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 $\mathcal{F}_o$variable fuva A to the $\mathcal{H}_o$ variable uva B. The link-β 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 $\mathcal{H}_o$ to be in $\mathcal{L}_\lambda$.

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

explain why

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

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

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

under the subsitution $\rho = \{X \mapsto \lambda x.x\}$.
The links generated from this unification problem are:

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

$$\vdash X0 =_\eta \lambda x.X3_x \tag{7}$$

$$x \vdash X3_x =_\beta X2\ X1_x\ a \tag{8}$$

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

A bit too fast, we first compile, then unify, then the oracle, then the manip...

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 $x \vdash X3 =_\beta X2xa$. The rhs of the link has now a variable which is partially in the PF, we can therefore remove the original link-β and replace it with the following couple on links:

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

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

6.3 Tricky examples

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

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

7 FIRST ORDER APPROXIMATION

TODO: Coq can solve this: `f 1 2 = X 2`, by setting X to f 1
TODO: We can re-use part of the algo for β given before

8 UNIF ENCODING IN REAL LIFE

TODO: Il ML presentato qui è esattamente elpi
TODO: Il OL presentato qui è esattamente coq
TODO: Come implementatiamo tutto ciò nel solver

9 RESULTS: STDPP AND TLC

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

10 CONCLUSION

REFERENCES

[1] Arthur Charguéraud. "The Optimal Fixed Point Combinator". In: *Interactive Theorem Proving*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 195–210. ISBN: 978-3-642-14052-5.

[2] Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. "Implementing HOL in an Higher Order Logic Programming Language". In: *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. LFMTP '16. Porto, Portugal: Association for Computing Machinery, 2016. ISBN: 9781450347778. DOI: 10.1145/2966268.2966272. URL: https://doi.org/10.1145/2966268.2966272.

[3] Cvetan Dunchev et al. "ELPI: Fast, Embeddable, λProlog 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%5C_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 Léchenet, and Enrico Tassi. "Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin's data types with Coq-Elpi". In: *CPP '23: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. Boston MA USA, France: ACM, Jan. 2023, pp. 167–181. DOI: 10.1145/3573105.3575683. URL: https://inria.hal.science/hal-03800154.

[8] RALF JUNG et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151.

[9] Dale Miller. "Unification under a mixed prefix". In: *Journal of Symbolic Computation* 14.4 (1992), pp. 321–358. DOI: 10.1016/0747-7171(92)90011-R.

[10] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. DOI: 10.1017/CBO9781139021326.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7.

[12] Lawrence C. Paulson. "Set theory for verification. I: from foundations to functions". In: *J. Autom. Reason.* 11.3 (Dec. 1993), pp. 353–389. ISSN: 0168-7433. DOI: 10.1007/BF00881873. URL: https://doi.org/10.1007/BF00881873.

[13] F. Pfening. "Elf: a language for logic definition and verified metaprogramming". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, California, USA: IEEE Press, 1989, pp. 313–322. ISBN: 0818619546.

[14] Frank Pfenning and Carsten Schürmann. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems". In: *Automated Deduction — CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2.

[15] Colin Rothgang, Florian Rabe, and Christoph Benzmüller. "Theorem Proving in Dependently-Typed Higher-Order Logic". In: *Automated Deduction – CADE 29*. Ed. by Brigitte Pientka and Cesare Tinelli. Cham: Springer Nature Switzerland, 2023, pp. 438–455. ISBN: 978-3-031-38499-8.

[16] Enrico Tassi. "Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq". In: *ITP 2019 - 10th International Conference on Interactive Theorem Proving*. Portland, United States, Sept. 2019. DOI: 10.4230/LIPIcs.CVIT.2016.23. URL: https://inria.hal.science/hal-01897468.

[17] Enrico Tassi. "Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)". In: *The Fourth International Workshop on Coq for Programming Languages*. Los Angeles (CA), United States, Jan. 2018. URL: https://inria.hal.science/hal-01637063.

[18] The Coq Development Team. *The Coq Reference Manual – Release 8.18.0*. https://coq.inria.fr/doc/V8.18.0/refman. 2023.

[19] P. Wadler and S. Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing

Machinery, 1989, pp. 60–76. ISBN: 0897912942. DOI: 10.1145/75277.75283. URL: https://doi.org/10.1145/75277.75283.

[20] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. "The Isabelle Framework". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7.

APPENDIX

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)

11 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.
```

12 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).

typeabbrev subst fsubst.

type fder subst -> fm -> fm -> o.
fder _ (fcon C) (fcon C).
fder S (fapp A) (fapp B) :- forall2 (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 subst -> fm -> fm -> o.                  (ρs)
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) :- forall2 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 :-                                      (η_l)
  pi x\ beta T [x] (T' x), x =_o x => F x =_o T' x.
T =_o flam F :-                                      (η_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. (β_l)
T =_o fapp [flam X | L] :- beta (flam X) L R, T =_o R. (β_r)

type extend-subst fm -> subst -> subst -> 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 (fapp [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) :- forall2 eta-contract L L1.
eta-contract (flam F) T :- eta-contract-aux [] (flam F) T.
eta-contract (flam F) (flam F1) :-
```

```
  pi x\ eta-contract x x => eta-contract (F x) (F1 x).
eta-contract (fuva X) (fuva X).
eta-contract X X :- name X.

type eta-contract-aux list fm -> fm -> fm -> o.
eta-contract-aux L (flam F) T :-
  pi x\ eta-contract-aux [x|L] (F x) T. % also checks H Prefix does not see x
eta-contract-aux L (fapp [H|Args]) T :-
  rev L LRev, append Prefix LRev Args,
  if (Prefix = []) (T = H) (T = fapp [H|Prefix]).
```

13 THE META LANGUAGE

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

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

type (≃λ) tm -> tm -> subst -> subst -> o.
(con C ≃λ con C) S S.
(app L1 ≃λ app L2) S S1 :- fold2 (≃λ) L1 L2 S S1.
(lam F1 ≃λ lam F2) S S1 :-
  pi x\ (pi S\ (x ≃λ x) S S) => (F1 x ≃λ F2 x) S S1.
(uva N Args ≃λ T) S S1 :-
  set? N S F,!, move F Args T1, (T1 ≃λ T) S S1.
(T ≃λ uva N Args) S S1 :-
  set? N S F,!, move F Args T1, (T ≃λ T1) S S1.
(uva M A1 ≃λ uva N A2) S1 S2 :- !,
  pattern-fragment A1, pattern-fragment A2,
  prune! M A1 N A2 S1 S2.
(uva N Args ≃λ T) S S1 :- not_occ N S T, pattern-fragment Args,
  bind T Args T1, assign N S T1 S1.
(T ≃λ uva N Args) S S1 :- not_occ N S T, pattern-fragment Args,
  bind T Args T1, assign N S T1 S1.

type prune! addr -> list tm -> addr ->
            list tm -> subst -> subst -> o.
/* no pruning needed */
prune! N A  N A  S S :- !.
prune! M A  N A  S1 S2 :- !, bind (uva M A) A Ass,
  assign N S1 Ass S2.
/* prune different arguments */
prune! N A1 N A2 S1 S3 :- !,
  new S1 W S2, prune-same-variable W A1 A2 [] Ass,
  assign N S2 Ass S3.
/* prune to the intersection of scopes */
prune! N A1 M A2 S1 S4 :- !,
  new S1 W S2, prune-diff-variables W A1 A2 Ass1 Ass2,
  assign N S2 Ass1 S3,
  assign M S3 Ass2 S4.

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

type permute list nat -> list tm -> list tm -> o.
permute [] _ [].
permute [P|PS] Args [T|TS] :-
  nth P Args T,
  permute PS Args TS.

type build-perm-assign addr -> list tm -> list bool ->
                       list nat -> assignment -> o.
build-perm-assign N ArgsR [] Perm (val (uva N PermutedArgs)) :-
  rev ArgsR Args, permute Perm Args PermutedArgs.
build-perm-assign N Acc [tt|L] Perm (abs T) :-
  pi x\ build-perm-assign N [x|Acc] L Perm (T x).
build-perm-assign N Acc [ff|L] Perm (abs T) :-
  pi x\ build-perm-assign N Acc L Perm (T x).

type keep list A -> A -> bool -> o.
keep L A tt :- mem L A, !.
keep _ _ ff.

type prune-diff-variables addr -> list tm -> list tm ->
                          assignment -> assignment -> o.
prune-diff-variables N Args1 Args2 Ass1 Ass2 :-
  forall2 (keep Args2) Args1 Bits1,
  forall2 (keep Args1) Args2 Bits2,
  filter Args1 (mem Args2) ToKeep1,
  filter Args2 (mem Args1) ToKeep2,
  forall2 (index ToKeep1) ToKeep1 IdPerm,
  forall2 (index ToKeep1) ToKeep2 Perm21,
  build-perm-assign N [] Bits1 IdPerm Ass1,
  build-perm-assign N [] Bits2 Perm21 Ass2.

type beta tm -> list tm -> tm -> o.
beta A [] A.
beta (lam Bo) [H | L] R :- beta (Bo H) L R.
beta (app A) L (app X) :- append A L X.
beta (con H) L (app [con H | L]).
beta X L (app[X|L]) :- name X.

/* occur check for N before crossing a functor */
type not_occ  addr -> subst -> tm -> o.
not_occ N S (uva M Args) :- set? M S F,
  move F Args T, not_occ N S T.
not_occ N S (uva M Args) :- unset? M S, not (M = N),
  forall1 (not_occ_aux N S) Args.
not_occ _ _ (con _).
not_occ N S (app L) :- not_occ_aux N S (app L).
/* Note: lam is a functor for the meta language! */
```

```
not_occ N S (lam L) :- pi x\ not_occ_aux N S (L x).
not_occ _ _ X :- name X.
/* finding N is ok */
not_occ N _ (uva N _).

/* occur check for X after crossing a functor */
type not_occ_aux addr -> subst -> tm -> o.
not_occ_aux N S (uva M _) :- unset? M S, not (N = M).
not_occ_aux N S (uva M Args) :- set? M S F,
  move F Args T, not_occ_aux N S T.
not_occ_aux N S (app L) :- forall1 (not_occ_aux N S) L.
not_occ_aux N S (lam F) :- pi x\ not_occ_aux N S (F x).
not_occ_aux _ _ (con _).
not_occ_aux _ _ X :- name X.
/* finding N is ko, hence no rule */

/* copy T T' vails if T contains a free variable, i.e. it
   performs scope checking for bind */
type copy tm -> tm -> o.
copy (con C)   (con C).
copy (app L)   (app L') :- forall2 copy L L'.
copy (lam T)   (lam T') :- pi x\ copy x x => copy (T x) (T' x).
copy (uva A L) (uva A L') :- forall2 copy L L'.

type bind tm -> list tm -> assignment -> o.
bind T [] (val T') :- copy T T'.
bind T [X | TL] (abs T') :- pi x\ copy X x => bind T TL (T' x).

type deref subst -> tm -> tm -> o.                    (σt)
deref _ (con C) (con C).
deref S (app A) (app B) :- forall2 (deref S) A B.
deref S (lam F) (lam G) :-
  pi x\ deref S x x => deref S (F x) (G x).
deref S (uva N L) R :- set? N S A, move A L T, deref S T R.
deref S (uva X A) (uva X B) :- unset? X S, forall2 (deref S) A B.

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


type deref-assmt subst -> assignment -> assignment -> o.
deref-assmt S (abs T) (abs R) :- pi x\ deref-assmt S (T x) (R x).
deref-assmt S (val T) (val R) :- deref S T R.
```

14 THE COMPILER

```
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 map (list mapping).

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

macro @val-link-eta T1 T2 :- ho.val (link-eta T1 T2).
macro @val-link-beta T1 T2 :- ho.val (link-beta T1 T2).



%% x occurs rigidly in t iff ∀σ,∀t′,t′ =ₒ σt ⇒ x ∈ 𝒫(t′)
%%
type occurs-rigidly fm -> fm -> o.
occurs-rigidly N N.
occurs-rigidly _ (fapp [fuva _|_]) :- !, fail.
occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.
occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).

/* maybe-eta N T L succeeds iff T could be an eta expasions for N; that
%%   is ∃σ,σ(λn.t) = λn.t′n and n
%%   does not occur rigidly in t′
type maybe-eta fm -> fm -> list fm -> o.
maybe-eta N (fapp[fuva _|Args]) _ :- !,
  exists (x\ maybe-eta-of [] N x) Args, !.
maybe-eta N (flam B) L :- !, pi x\ maybe-eta N (B x) [x | L].
maybe-eta _ (fapp [fcon _|Args]) L :-
  split-last-n {len L} Args First Last,
  forall1 (x\ forall1 (y\ not (occurs-rigidly x y)) First) L,
  forall2 (maybe-eta-of []) {rev L} Last.

%%   is ∃σ,σt =ₒ n
type maybe-eta-of list fm -> fm -> fm -> o.
maybe-eta-of _ N N :- !.
maybe-eta-of L N (fapp[fuva _|Args]) :- !,
  forall1 (x\ exists (maybe-eta-of [] x) Args) [N|L].
maybe-eta-of L N (flam B) :- !,
  pi x\ maybe-eta-of [x | L] N (B x).
maybe-eta-of L N (fapp [N|Args]) :-
  last-n {len L} Args R,
  forall2 (maybe-eta-of []) R {rev L}.


type locally-bound tm -> o.
type get-scope-aux tm -> list tm -> o.
get-scope-aux (con _) [].
get-scope-aux (uva _ L) L1 :-
  forall2 get-scope-aux L R,
  flatten R L1.
get-scope-aux (lam B) L1 :-
  pi x\ locally-bound x => get-scope-aux (B x) L1.
get-scope-aux (app L) L1 :-
```

```
  forall2 get-scope-aux L R,
  flatten R L1.
get-scope-aux X [X] :- name X, not (locally-bound X).
get-scope-aux X [] :- name X, (locally-bound X).

%% TODO: scrivere undup
type get-scope tm -> list tm -> o.
get-scope T Scope :-
  get-scope-aux T ScopeDuplicata,
  names N, filter N (mem ScopeDuplicata) Scope.
type rigid fm -> o.
rigid X :- not (X = fuva _).

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

type close-links (tm -> links) -> links -> o.
close-links (_\[]) [].
close-links (v\[L|XS v]) [L|YS] :- !, close-links XS YS.
close-links (v\[(L v)|XS v]) [abs L|YS] :- !,
  close-links XS YS.
type comp fm -> tm -> map -> map -> links -> links ->
  subst -> subst -> o.
comp (fcon C) (con C)       M1 M1 L1 L1 S1 S1.
comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
  (pi x\ maybe-eta x (F x) [x]), !,
    alloc S1 A S2,
    comp-lam F F1 M1 M2 L1 L2 S2 S3,
    get-scope (lam F1) Scope,
    L3 = [@val-link-eta (uva A Scope) (lam F1)| L2].
comp (flam F) (lam F1)      M1 M2 L1 L2 S1 S2 :-
  comp-lam F F1 M1 M2 L1 L2 S1 S2.
comp (fuva A) (uva B [])    M1 M2 L L S S1 :-
  m-alloc (fv A) (hv B (arity z)) M1 M2 S S1.
comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L1 L1 S1 S2 :-
  pattern-fragment Ag, !,
    fold6 comp Ag Ag1 M1 M1 L1 L1 S1 S1,
    len Ag Arity,
    m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
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 = [@val-link-beta (uva C Scope) Beta | L2].
comp (fapp A) (app A1)      M1 M2 L1 L2 S1 S2 :-
  fold6 comp A A1 M1 M2 L1 L2 S1 S2.

type alloc mem A -> addr -> mem A -> o.
```

```
alloc S N S1 :- mem.new S N S1.

type compile-terms-diagnostic
  triple diagnostic fm fm ->
  triple diagnostic tm tm ->
  map -> map ->
  links -> links ->
  subst -> subst -> o.
compile-terms-diagnostic (triple D FO1 FO2) (triple D HO1 HO2) M1 M3 L1
  comp FO1 HO1 M1 M2 L1 L2 S1 S2,
  comp FO2 HO2 M2 M3 L2 L3 S2 S3.

type compile-terms
  list (triple diagnostic fm fm) ->
  list (triple diagnostic tm tm) ->
  map -> 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) @one T1,
  L = [@val-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) @one T2,
  (pi x\ make-eta-link-aux N Ad Ad2 [x|Scope1] (L1 x) H2 H3),
  close-links L1 L2,
  L = [@val-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 map -> map ->
    subst -> subst -> links -> links -> o.
deduplicate-map [] [] H H L L.
deduplicate-map [(mapping (fv O) (hv M (arity LenM)) as X1) | Map1] Map2
  take-list Map1 (mapping (fv O) (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 H}.
```

15 THE PROGRESS FUNCTION

```
macro @one :- s z.

type contract-rigid list ho.tm -> ho.tm -> ho.tm -> o.
contract-rigid L (ho.lam F) T :-
  pi x\ contract-rigid [x|L] (F x) T. % also checks H Prefix does not see x
contract-rigid L (ho.app [H|Args]) T :-
  rev L LRev, append Prefix LRev Args,
  if (Prefix = []) (T = H) (T = ho.app [H|Prefix]).

type progress-eta-link ho.tm -> ho.tm -> ho.subst -> ho.subst -> links -> o.
progress-eta-link (ho.app _ as T) (ho.lam x\ _ as T1) H H1 [] :- !,
  ({eta-expand T @one} ==l T1) H H1.
progress-eta-link (ho.con _ as T) (ho.lam x\ _ as T1) H H1 [] :- !,
  ({eta-expand T @one} ==l T1) H H1.
progress-eta-link (ho.lam _ as T) T1 H H1 [] :- !,
  (T ==l T1) H H1.
progress-eta-link (ho.uva _ _ as X) T H H1 [] :-
  contract-rigid [] T T1, !, (X ==l T1) H H1.
progress-eta-link (ho.uva Ad _ as T1) T2 H H [@val-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 ==l T2) S1 S2.
progress-beta-link-aux T1 T2 S S [@val-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 [@val-link-beta T 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) ==l T1) S1 S2.

progress-beta-link (ho.uva _ _ as T) (ho.app[ho.uva Ad1 Scope1 | L1] as T1) S1 S3 NewLinks :-
  append Scope1 L1 Scope1L,
  pattern-fragment-prefix Scope1L Scope2 L2,
  not (Scope1 = Scope2), !,
  mem.new S1 Ad2 S2,
  len Scope1 Scope1Len,
  len Scope2 Scope2Len,
  make-eta-link Scope1Len Scope2Len Ad1 Ad2 [] LinkEta S2 S3,
  if (L2 = []) (NewLinks = LinkEta, T2 = ho.uva Ad2 Scope2)
    (T2 = ho.app [ho.uva Ad2 Scope2 | L2],
    NewLinks = [@val-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 _
  occur-check-err T T2 S1, !, fail.

progress-beta-link T1 (ho.app[ho.uva _ _ | _] as T2) H H [@val-link-beta

progress-beta-link T1 (ho.app [Hd | Tl]) S1 S2 B :-
  ho.beta Hd Tl 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 (@val-link-eta A B) NewLinks S S1 :- !,
  progress-eta-link A B S S1 NewLinks.

solve-link-abs (@val-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 (@val-link-eta (ho.uva N _) _) (@val-link-eta (ho.uva

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 (@val-link-eta (ho.uva N S1) A)
              (@val-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' ==l B) H H1.

type solve-links links -> links -> ho.subst -> ho.subst -> o.
solve-links [] [] X X.
```

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

16 THE DECOMPILER

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

type commit-links-aux link -> ho.subst -> ho.subst -> o.
commit-links-aux (@val-link-eta T1 T2) H1 H2 :-
  ho.deref H1 T1 T1', ho.deref H1 T2 T2',
  (T1' ==l T2') H1 H2.
commit-links-aux (@val-link-beta T1 T2) H1 H2 :-
  ho.deref H1 T1 T1', ho.deref H1 T2 T2',
  (T1' ==l T2') H1 H2.
commit-links-aux (ho.abs B) H H1 :-
  pi x\ commit-links-aux (B x) H H1.

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

type decompl-subst map -> map -> ho.subst ->
  fo.subst -> fo.subst -> o.
decompl-subst _ [A|_] _ _ _ :- fail.
decompl-subst _ [] _ F F.
decompl-subst Map [mapping (fv VO) (hv VM _)|Tl] H F F2 :-
  mem.set? VM H T, !,
  ho.deref-assmt H T TTT,
  abs->lam TTT T', tm->fm Map T' T1,
  fo.eta-contract T1 T2, mem.assign VO F T2 F1,
  decompl-subst Map Tl H F1 F2.
decompl-subst Map [mapping _ (hv VM _)|Tl] H F F2 :-
  mem.unset? VM H, decompl-subst Map Tl H F F2.

type tm->fm map -> ho.tm -> fo.fm -> o.
tm->fm _ (ho.con C)  (fo.fcon C).
tm->fm L (ho.lam B1) (fo.flam B2) :-
  pi x y\ tm->fm _ x y => tm->fm L (B1 x) (B2 y).
tm->fm L (ho.app L1) T :- forall2 (tm->fm L) L1 [Hd|Tl],
  fo.mk-app Hd Tl T.
tm->fm L (ho.uva VM TL) T :- mem L (mapping (fv VO) (hv VM _)),
  forall2 (tm->fm L) TL T1, fo.mk-app (fo.fuva VO) T1 T.

type add-new-map-aux ho.subst -> list ho.tm -> map ->
     map ->  fo.subst -> fo.subst -> o.
add-new-map-aux _ [] _ [] S S.
add-new-map-aux H [T|Ts] L L2 S S2 :-
  add-new-map H T L L1 S S1,
  add-new-map-aux H Ts L1 L2 S1 S2.
```

```
type add-new-map ho.subst -> ho.tm -> map ->
   map ->  fo.subst -> fo.subst -> o.
add-new-map _ (ho.uva N _) Map [] F1 F1 :-
  mem Map (mapping _ (hv N _)), !.
add-new-map H (ho.uva N L) Map [Map1 | MapL] F1 F3 :-
  mem.new F1 M F2,
  len L Arity, Map1 = mapping (fv M) (hv N (arity Arity)),
  add-new-map H (ho.app L) [Map1 | Map] MapL F2 F3.
add-new-map H (ho.lam B) Map NewMap F1 F2 :-
  pi x\ add-new-map H (B x) Map NewMap F1 F2.
add-new-map H (ho.app L) Map NewMap F1 F3 :-
  add-new-map-aux H L Map NewMap F1 F3.
add-new-map _ (ho.con _) _ [] F F :- !.
add-new-map _ N _ [] F F :- name N.

type complete-mapping-under-ass ho.subst -> ho.assignment ->
  map -> map ->  fo.subst -> fo.subst -> o.
complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
  add-new-map H Val Map1 Map2 F1 F2.
complete-mapping-under-ass H (ho.abs Abs) Map1 Map2 F1 F2 :-
  pi x\ complete-mapping-under-ass H (Abs x) Map1 Map2 F1 F2.

type complete-mapping ho.subst -> ho.subst ->
  map -> map -> fo.subst -> fo.subst -> o.
complete-mapping _ [] L L F F.
complete-mapping H [none | Tl] L1 L2 F1 F2 :-
  complete-mapping H Tl L1 L2 F1 F2.
complete-mapping H [some T0 | Tl] L1 L3 F1 F3 :-
  ho.deref-assmt H T0 T,
  complete-mapping-under-ass H T L1 L2 F1 F2,
  append L1 L2 LAll,
  complete-mapping H Tl LAll L3 F2 F3.

type decompile map -> links -> ho.subst ->
  fo.subst -> fo.subst -> o.
decompile Map1 L HO FO FO2 :-
  commit-links L L1_ HO HO1, !,
  complete-mapping HO1 HO1 Map1 Map2 FO FO1,
  decompl-subst Map2 Map2 HO1 FO1 FO2.
```

17 AUXILIARY FUNCTIONS

```
type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A ->
  list A1 -> B -> B -> C -> C -> o.
fold4 _ [] [] A A B B.
fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0,
  fold4 F XS YS A0 A1 B0 B1.

type len list A -> nat -> o.
len [] z.
len [_|L] (s X) :- len L X.
```