HO unification from object language to meta language

Enrico Tassi
enrico.tassi@inria.fr
Université Côte d'Azur, Inria
France

Davide Fissore
davide.fissore@inria.fr
Université Côte d'Azur, Inria
France

ABSTRACT

Specifying and implementing a logic from scratch requires significant effort. Logical Frameworks and Higher Order Logic Programming Languages provide dedicated, high-level Meta Languages (ML) to facilitate this task in two key ways: 1) variable binding and substitution are simplified when ML binders represent object logic ones; 2) proof construction, and even proof search, is greatly simplified by leveraging the unification procedure provided by the ML. Notable examples of ML are Elf [13], Twelf [14], 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:
Enrico Tassi and Davide Fissore. XXXX 2024. HO unification from object language to meta language. In *YYY*. ACM, New York, NY, USA, 15 pages. https://doi.org/ZZZZZZZZZZZZ

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/ZZZZZZZZZZZZ

1 INTRODUCTION

Specifying and implementing a logic from scratch requires significant effort. Logical Frameworks and Higher Order Logic Programming Languages provide dedicated, high-level Meta Languages (ML) to facilitate this task in two key ways: 1) variable binding and substitution are simplified when ML binders represent object logic ones; 2) proof construction, and even proof search, is greatly simplified by leveraging the unification procedure provided by the ML. Notable examples of ML are Elf [13], Twelf [14], 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"]
```

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.

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 fixpoint 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:

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

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

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

$$\text{fstep}(\mathcal{S}, p, \rho) \mapsto \rho'' \stackrel{def}{=\!=} \rho\mathcal{S}_{p_l} \simeq_o \rho\mathcal{S}_{p_r} \mapsto \rho' \wedge \rho'' = \rho \cup \rho'$$
$$\text{frun}(\mathcal{S}, \mathcal{N}) \mapsto \rho_{\mathcal{N}} \stackrel{def}{=\!=} \textstyle\bigwedge_{p=1}^{\mathcal{N}} \text{fstep}(\mathcal{S}, p, \rho_{p-1}) \mapsto \rho_p$$

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.

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

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, *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 $\mathcal{H}_o$.

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

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

Proposition 2.3 (Properties of $\simeq_o$).

$$s_i \in \mathcal{L}_\lambda \Rightarrow s_1 \simeq_o s_2 \mapsto \rho \Rightarrow \rho s_1 =_o \rho s_2 \tag{3}$$
$$s_i \in \mathcal{L}_\lambda \Rightarrow \rho s_1 =_o \rho s_2 \Rightarrow \exists \rho', s_1 \simeq_o s_2 \mapsto \rho' \wedge \rho' \subseteq \rho \tag{4}$$
$$\rho s_1 =_o \rho s_2 \Rightarrow \rho' \subseteq \rho \Rightarrow \rho' s_i \in \mathcal{L}_\lambda \Rightarrow \rho' s_1 \simeq_o \rho' s_2 \tag{5}$$

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.

[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

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

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

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

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

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

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

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

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 $\mathbb{L}$) by compilation.

3 ALTERNATIVE ENCODINGS AND RELATED WORK

Paper [2] introduces semi-shallow.

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

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

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

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

The second reason is the encoding for Coq is used for meta programming the system, hence it must accommodate the manipulation of terms that are now 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 nat -> fm.             type uva nat -> list tm -> tm.
```

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

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

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

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

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

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

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

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

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

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

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

We call `fsubst` the memory of $\mathcal{F}_o$, 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

4.1 Notations

we use math mode for ho.

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

```
type (=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.
```

Figure 2: Equal predicate ML

```
type fderef fsubst -> fm -> fm -> o.                    (ρs)
fderef S T R :- fder S T T', napp T' R.

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

TODO explain napp. We use the cut operator to keep the code compact. It is possible to rewrite the rule for application traversal 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 S (app A) (app B) :- map (deref S) A B.
deref S (lam F) (lam G) :-
  pi x\ deref S x x => deref S (F x) (G x).
deref _ (con C) (con C).
deref S (uva N A) R :- set? N S F, move F A T, deref S T R.
deref S (uva N A) (uva N B) :- unset? N S, map (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 :- !.
move (val (uva N A)) L     (uva N X) :- append A L X.
```

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

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

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

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

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

The equality relation for the ML, accepts $\eta\beta$-equivalence between terms of the ML. Recall that `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 $\rho_1 \subseteq \rho_2$. The first three rules unify terms with same rigid heads, and call the unification relation on the sub-terms. If t_1 (resp. t_2) is an assigned variables, t_1 is dereferenced to t_1' (resp. t_2') and the unification is called between t_1' and t_2 (resp. t_1 and t_2'). If both terms are unification variables, we test that their arguments are in the pattern fragment, we allocate a new variable w in ρ_1 such that w is the pruning of the arguments of t_1 and t_2, we assign both t_1 and t_2 to w and return the new mapping ρ_2 containing all the new variable assignment. Finally, if only one of the two terms is an unification variable v, after having verified that v does not occur in the other term t, we bind v to t and return the new substitution mapping.

OLD

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

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

The solution we are proposing aim to overcome these unification issues by 1) compiling the terms t and u of the OL into an internal version t' and u' in the ML; 2) unifying t' and u' at the meta level instantiating meta variables; 3) decompiling the meta variable into terms of the OL; 4) assigning the variables of the OL with the decompiled version of their corresponding meta variables. We claim that t and u unify if and only if t' and u' unify and that the substitution in the object language is the same as the one returned by the ML.

same or ⊇ or ⊆

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

5 BASIC COMPILATION $\mathcal{F}_o$ TO $\mathcal{H}_o$

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

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

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

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

say when this is needed

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

Then decomp

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

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

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

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

Now unif

TODO link

TODO nuove subst

TODO: code unif

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

5.1 Prolog simulation

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

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

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

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

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

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

5.2 Example

OK

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

KO

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

```
HS = [some (abs x\con"a")]
S = [some (flam x\fcon a)]
lam x\ app[f, app[X, x]] = Y,
  lam x\ x) = X.
```

TODO: Goal: $s_1 \simeq_o s_2$ 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

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.

A subterm is in $\Diamond\beta$ if it has the shape `fapp[fuva N | L]` and `distinct L` does not hold. In that case, `L` is split in two sublist `PF` and `NPF` such that former is the longest prefix of `L` such that `distinct PF` holds. `NPF` is the list such that `append PF NPF L`. The LHS is set to a new variable named M with `PF` in scope whereas the RHS is given by the term `app[uva N' PF | NPF]` where the $\mathcal{H}_o$ variable identified by `N'` is mapped to the $\mathcal{F}_o$ variable named `N`.

signposting: a link is reconsidered in two cases. then make 2 paragraphs

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

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

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

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 `c0\ X2 'X1 c0' a` (it is a $\Diamond\beta$). The substitution tells that `x ⊢ 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 `X3 x =β= 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-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

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 address type.
type addr nat -> address.

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

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

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

type unset? address -> 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 address -> mem A -> A -> mem A -> o.
assign (addr A) Mem1 Val Mem2 :- assign-aux A Mem1 Val Mem2.
```

12 THE OBJECT LANGUAGE

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

typeabbrev subst mem fm.

type fder subst -> fm -> fm -> o.
fder S (fuva N) T1 :- set? N S T, fder S T T1.
fder S (fapp L1) (fapp L2) :- forall2 (fder S) L1 L2.
fder S (flam F1) (flam F2) :-
  pi x\ fder S x x => fder S (F1 x) (F2 x).
fder _ (fcon X) (fcon X).
fder _ (fuva N) (fuva N).

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 fderef subst -> fm -> fm -> o.
fderef S T T2 :- fder S T T1, napp T1 T2.

type (=_o) fm -> fm -> o.
fapp L1 =_o fapp L2 :- forall2 (=_o) L1 L2.
flam F1 =_o flam F2 :- pi x\ x =_o x => F1 x =_o F2 x.
fcon X =_o fcon X.
fuva N =_o fuva N.
flam F =_o T :- pi x\ beta T [x] (T' x), x =_o x => F x =_o T' x.
T =_o flam F :- pi x\ beta T [x] (T' x), x =_o x => T' x =_o F x.
fapp [flam X | TL] =_o T :- beta (flam X) TL T', T' =_o T.
T =_o fapp [flam X | TL] :- beta (flam X) TL T', T =_o T'.

type extend-subst fm -> 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

```
typeabbrev subst list (option assignment).

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

typeabbrev assignment (inctx tm).

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

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

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  address -> 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 address -> 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 scope-check o.

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.
deref H (uva N L)  X         :- set? N H T,
  move T L X', !, deref H X' X.
deref H (app L)   (app L1)   :- forall2 (deref H) L L1.
deref _ (con X)   (con X).
deref H (uva X L) (uva X L1) :- unset? X H,
  forall2 (deref H) L L1.
deref H (lam F)   (lam G)    :- pi x\ deref H (F x) (G x).
deref _ N         N          :- name N.

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 address -> fvariable.

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

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

typeabbrev scope (list tm).

kind linkctx type.
type link-eta tm -> tm -> linkctx.
type link-beta tm -> tm -> linkctx.

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

typeabbrev link (ho.inctx linkctx).

typeabbrev links (list link).


type use-binder fm -> fm -> o.
use-binder N N.
use-binder N (fapp L) :- exists (use-binder N) L.
use-binder N (flam B) :- pi x\ use-binder N (B x).

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 (use-binder x y)) First) L,
  forall2 (maybe-eta-of []) {rev L} Last.

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).

type get-scope tm -> list tm -> o.
get-scope T Scope :- names N,
  get-scope-aux T ScopeDuplicata,
```

```
  filter N (mem ScopeDuplicata) Scope.

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

type comp-lam (fm -> fm) -> (tm -> tm) ->
  mappings -> mappings -> 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 comp fm -> tm -> mappings -> mappings -> 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 :-
  alloc-mapping M1 M2 (fv A) (hv B (arity z)) S S1.
comp (fapp [fuva A|Scope]) (uva B Scope1) M1 M2 L1 L1 S1 S2 :-
  pattern-fragment Scope, !,
    fold6 comp Scope Scope1 M1 M1 L1 L1 S1 S1,
    len Scope Arity,
    alloc-mapping M1 M2 (fv A) (hv B (arity Arity)) S1 S2.
comp (fapp [fuva A|Args]) (uva C Scope) M1 M3 L1 L3 S1 S4 :- !,
  pattern-fragment-prefix Args Pf Extra,
    fold6 comp Pf    Scope1 M1 M1 L1 L1 S1 S1,
    fold6 comp Extra Extra1 M1 M2 L1 L2 S1 S2,
    len Pf Arity,
    alloc-mapping M2 M3 (fv A) (hv B (arity Arity)) S2 S3,
    Beta = app [uva B Scope1 | Extra1],
    get-scope Beta Scope,
    alloc S3 C S4,
    L3 = [@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 -> address -> mem A -> o.
alloc S N S1 :- mem.new S N S1.

type compile-terms-diagnostic
  triple diagnostic fm fm ->
  triple diagnostic tm tm ->
  mappings -> mappings ->
  links -> links ->
  subst -> subst -> o.
compile-terms-diagnostic (triple D FO1 FO2) (triple D HO1 HO2) M1 M3 L1 L3 S1 S3 :-
  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) ->
  mappings -> links -> subst -> o.
compile-terms T H M L S :-
  fold6 compile-terms-diagnostic T H [] M_ [] L_ [] S_,
  deduplicate-mappings M_ M S_ S L_ L.

type make-eta-link-aux nat -> address -> address ->
  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 -> address -> address ->
       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-mappings mappings -> mappings ->
    subst -> subst -> links -> links -> o.
deduplicate-mappings [] [] H H L L.
deduplicate-mappings [(mapping (fv O) (hv M (arity LenM)) as X1) | Map1]
  take-list Map1 (mapping (fv O) (hv M' (arity LenM'))) _, !,
  std.assert! (not (LenM = LenM')) "Deduplicate mappings, 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-mappings Map1 Map2 H2 H3 L2 L3.
deduplicate-mappings [A|As] [A|Bs] H1 H2 L1 L2 :-
  deduplicate-mappings As Bs H1 H2 L1 L2, !.
deduplicate-mappings [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 se
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 mappings -> mappings -> 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 mappings -> 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-mappings-aux ho.subst -> list ho.tm -> mappings ->
      mappings ->  fo.subst -> fo.subst -> o.
add-new-mappings-aux _ [] _ [] S S.
add-new-mappings-aux H [T|Ts] L L2 S S2 :-
  add-new-mappings H T L L1 S S1,
  add-new-mappings-aux H Ts L1 L2 S1 S2.

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

type complete-mapping-under-ass ho.subst -> ho.assignment ->
  mappings -> mappings ->  fo.subst -> fo.subst -> o.
complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
  add-new-mappings 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 ->
  mappings -> mappings -> 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 mappings -> 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.
```