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 [15], Twelf [16], λProlog [11] and Isabelle [23] which have been utilized to implement various formal systems such as First Order Logic [5], Set Theory [14], Higher Order Logic [13], and even the Calculus of Constructions [4].

The object logic we are interested in is Coq's [21] 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 [10]. 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 [20, 19, 8, 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 [22] solver for Coq [21]. 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 [9] and TLC [1]. These two libraries constitute our test bed.

KEYWORDS

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

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 [15], Twelf [16], λProlog [11] and Isabelle [23] which have been utilized to implement various formal systems such as First Order Logic [5], Set Theory [14], Higher Order Logic [13], and even the Calculus of Constructions [4].

The object logic we are interested in is Coq's [21] Dependent Type Theory (DTT), and we want to code a type-class [22] solver for Coq [21] using the Coq-Elpi [20] 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 [9] 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 [11] 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) :- decomp 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 «decomp Pm A P» that is in charge of bringing the assignment for `Pm` back to the domain `tm` of Coq terms:

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

This simple example is sufficient to show that the encoding we seek is not trivial and does not only concern the head of rules, but the entire sequence of unification problems that constitute the execution of a logic program. In fact the solution for `P` above generates a (Coq) 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 $\mathcal{L}_\lambda$ [10].

Contributions. In this paper we discuss alternative encodings of Coq in Elpi (Section **??**), 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..

. 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 $\mathcal{L}_\lambda$ [10]. We call this unification procedure $\simeq_o$.

The equational theory of the meta language Elpi 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                     ≃λ   f
lam A x\ app[con"f", x]    ≃o   con"f"
lam A x\ app[con"f", x]    ≄λ   con"f"
P x                        ≃λ   x
app[P, x]                  ≃o   x
app[P, x]                  ≄λ   x
```

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

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

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

To state precisely the problem we solve we need a $\mathcal{F}_o$ 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.

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

> E:XXX improve....

We represent a logic program *run* in $\mathcal{F}_o$ as a list *steps* p of length $\mathcal{N}$. Each step is a unification problem between terms $\mathbb{P}_{p_l}$ and $\mathbb{P}_{p_r}$ taken from the set of all terms $\mathbb{P}$. 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

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

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

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.

$$\text{hstep}(\mathbb{T}, p, \sigma, \mathbb{L}) \mapsto (\sigma'', \mathbb{L}') \stackrel{def}{=}$$
$$\sigma\mathbb{T}_{p_l} \simeq_\lambda \sigma\mathbb{T}_{p_r} \mapsto \sigma' \wedge \text{progress}(\mathbb{L}, \sigma \cup \sigma') \mapsto (\mathbb{L}', \sigma'')$$

$$\text{hrun}(\mathbb{P}, \mathcal{N}) \mapsto \rho_{\mathcal{N}} \stackrel{def}{=}$$
$$\mathbb{T} \times \mathbb{M} \times \mathbb{L}_0 = \{(t_j, m_j, l_j) | s_j \in \mathbb{P}, \langle s_j \rangle \mapsto (t_j, m_j, l_j)\}$$
$$\bigwedge_{p=1}^{\mathcal{N}} \text{hstep}(\mathbb{T}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p)$$
$$\langle \sigma_{\mathcal{N}}, \mathbb{M}, \mathbb{L}_{\mathcal{N}} \rangle^{-1} \mapsto \rho_{\mathcal{N}}$$

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

PROPOSITION 2.1 (SIMULATION). $\forall\mathbb{P}, \forall\mathcal{N}$,

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

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

PROPOSITION 2.2 (SIMULATION FIDELITY). *In the context of* hrun, *if* $\mathbb{T} \subseteq \mathcal{L}_\lambda$ *we have that* $\forall p \in 1 \dots \mathcal{N}$,

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

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 does not fail eagerly

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

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

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

---

[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

PROPOSITION 2.3 (PROPERTIES OF $\simeq_o$).

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

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

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

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

> E:fix

Property 2.1 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 ($\overline{\mathcal{L}_\lambda}$). $\overline{\mathcal{L}_\lambda} = \{Xt_1 \dots t_n \mid Xt_1 \dots t_n \notin \mathcal{L}_\lambda\}$.

An example of t in $\overline{\mathcal{L}_\lambda}$ 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 back in $\mathcal{L}_\lambda$.

Definition 2.6 (Subterms $\mathcal{P}(t)$). The set of sub terms of t is the largest set *subtermt* that can be obtained by the following rules.

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

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

$$\mathcal{W}(X) \Leftrightarrow \forall t \in \mathcal{P}(X), t \notin (\overline{\mathcal{L}_\lambda} \cup \Diamond\eta)$$

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

$$\mathcal{W}(\sigma\mathbb{T}) \wedge \sigma\mathbb{T}_{p_l} \simeq_\lambda \sigma\mathbb{T}_{p_r} \mapsto \sigma' \Rightarrow \mathcal{W}(\sigma'\mathbb{T})$$
$$\mathcal{W}(\sigma\mathbb{T}) \wedge \text{progress}(\mathbb{L}, \sigma) \mapsto (\_, \sigma') \Rightarrow \mathcal{W}(\sigma'\mathbb{T})$$

A less formal way to state 2.8 is that hstep and progress never "commit" an unneeded λ-abstraction in σ (a λ that could be erased by an η-contraction), nor put 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 $\simeq_o$ as a whole since decompilation can introduce (actually restore) terms in $\diamond\eta$ or $\overline{\mathcal{L}_\lambda}$ that were move out of the way (put in $\mathbb{L}$) during compilation.

3 ALTERNATIVE ENCODINGS AND RELATED WORK

Paper [2] introduces semi-shallow.

Our encoding of DTT may look "semi shallow" since we use the meta-language lambda abstraction but not its application (for the terms of type 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 [17] 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: The $\mathcal{F}_o$ and $\mathcal{H}_o$ languages

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

the fapp constructor. For example the term P x is represented as 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.
```

Natural numbers represent the memory addresses that identify unification variables in both languages. 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.

> E: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. This is obtained via the inctx container, and in particular via its abs binding constructor. On the contrary a solution to a $\mathcal{F}_o$ variable is a plain term.

```
typeabbrev fsubst (mem fm).

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

We call 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.

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

kind fvariable type.
type fv addr -> fvariable.

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

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

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 compiler establishes a mapping between variables of the two languages. In order to preserve invariant 1 we store the arity of each hvariable in the mapping and we reuse an existing mapping only if the arity matches.

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

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

As we mentioned in section 2.1 the compiler replaces terms in $\diamond\eta$ and $\overline{\mathcal{L}_\lambda}$ with fresh variables linked to the problematic terms. Each class of problematic terms has a dedicated link.

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

The right hand side of a link, the problematic term, can occur under binders. To accommodate this situation the compiler wraps baselink using the inctx container (see, $\cdot \vdash \cdot$).

INVARIANT 2 (LINK LEFT HAND SIDE). *The left hand side of a suspended link is a variable.*

New links are suspended by construction. If the left hand side variable is assigned during a step, then the link is considered for progress and possibly eliminated. This is discussed in section 6 and section 8.

4.1 Notational conventions

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

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

When variables x and y can occur in term t we shall write t_{xy} to stress this fact.

We write $\sigma = \{ A_{xy} \mapsto y \}$ for the assignment abs x\abs y\y and $\sigma = \{ A \mapsto \lambda x.\lambda y.y \}$ for lam x\lam y\y .

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

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

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

4.2 Equational theory and Unification

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

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

The first four rules are common to both equalities and just define the usual congruence over terms, and since we use an HOAS encoding they also capture α-equivalence. In addition to that $=_o$ has rules for η and β-equivalence.

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

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

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

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

```
type beta fm -> list fm -> fm -> o.
beta A [] A.
beta (flam Bo) [H | L] R :- napp (Bo H) F, beta F 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 napp fm -> fm -> o.
napp (fcon C) (fcon C).
napp (fuva A) (fuva A).
napp (flam F) (flam G) :- pi x\ napp (F x) (G x).
napp (fapp [fapp L1 |L2]) T :- !,
  append L1 L2 L3, napp (fapp L3) T.
napp (fapp L) (fapp L1) :- map napp L L1.
napp N N :- name N.
```

The name predicate holds only on nominal constants (i.e. bound variables).[2] The choice of using n-ary application, rather than binary, is to make it easy to access the application's head. The price we pay is that substituting an application in the head of an application should be amended by "flattening" fapp nodes, that is the job of

---

[2]Elpi provides it as a builtin, but one could implement it by systematically loading the hypothetical rule name x every time a nominal constant is postulated via **pi** x\

napp. [3] Finally note that the cut operator is inessential, it could be removed at the cost of a verbose test on the head of `L` in the second rule about `fapp`: `L`'s head can be `fcon`, `flam` or a name.

Substitution application: ρs and σt. Applying the substitution corresponds to dereferencing a term with respect to the memory. To ease the comparison we split $\mathcal{F}_o$ dereferencing into a `fder` step and a `napp` one. The former step replaces references to memory cells that are set with their values, and has a corresponding operation in $\mathcal{H}_o$, namely `deref`. On the contrary `napp` has no corresponding operation in $\mathcal{H}_o$. The reasons for this asymmetry is that an `fapp` node with a flexible head is always mapped to a `uva` (as per section 5.1 and section 8), preventing nested applications to materialize.

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

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

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

```
type deref subst -> tm -> tm -> o.             (σt)
deref _ (con C) (con C).
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 S (uva N L) R :- set? N S A,
  move A L T, deref S T R.
deref S (uva N A) (uva N B) :- unset? N S,
  map (deref S) A B.
```

Note that move strongly relies on invariant 1: the length of the arguments of all occurrences of a unification variable and the number of abstractions in its assignment have to match. In turn this grants that move never fails.

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

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

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

The only detail worth discussing is the fact that the procedure updates a substitution, rather than just crafting one as presented

in section 2. The reason is that the algorithm folds over a term, updating a substitution while it traverses it.

> E:explain better

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 and the support for terms outside $\mathcal{L}_\lambda$ in Section 8.

5.1 Compilation

> E:manca beta normal in entrata

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

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

```
type comp fm -> tm -> mmap -> mmap -> links -> links ->
  subst -> subst -> o.
comp (fcon C) (con C) M M L L S S.
comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :-        (cλ)
  comp-lam F F1 M1 M2 L1 L2 S1 S2.
comp (fuva A) (uva B []) M1 M2 L L S1 S2 :-
  m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
  pattern-fragment Ag, !,
    fold6 comp Ag Ag1 M1 M1 L L 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` recognizes $\mathcal{F}_o$ variables applied to a (possibly empty) duplicate free list of names. Note that compiling `Ag` cannot create new mappings nor links, since `Ag` is made of bound variables and the hypothetical rule loaded by `comp-lam` (see below) grants this property.

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

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

The auxiliary function `close-links` tests if the bound variable `v` really occurs in the link. If it is the case the link is wrapped

---

[3]Note that `napp` is an artefact of formalization of $\mathcal{F}_o$ we do in this presentation and, as we explain later, no equivalent of `napp` is needed in $\mathcal{H}_o$.

into an additional abs node binding v. In this way links generated deep inside the compiled terms can be moved outside their original context of binders.

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

Note that we could remove the first rule, whose solve purpose is to make links more readable by pruning unused context entries.

5.2 Execution

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

```
type hstep tm -> tm -> links -> links -> subst -> subst -> o.
hstep T1 T2 L1 L2 S1 S3 :-
   (T1 ≃λ T2) S1 S2,
   progress L1 L2 S2 S3.
```

Note tha the infix notation ((A ≃λ B) C D) is syntactic sugar for ((≃λ) A B C D).

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

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

In the base compilation scheme progress1 is the identity on both the links and the substitution, so the fixpoint trivially terminates. Sections 6 and 8 add rules to progress1 and justify why the don't hinder termination. For brevity we omit the code that applies the substitution S1 to all terms in $\mathbb{L}$.

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

5.3 Substitution decompilation

Decompiling the substitution requires to first force the progress of links and then allocating new unassigned variables in the substitution for $\mathcal{F}_o$ and finally decompiling all assignments. Note that invariant 2 and the occur check allows us to update the subst.

```
type decompile mmap -> links -> subst ->
   fsubst -> fsubst -> o.
```

```
decompile M1 L S F1 F3 :-
   commit-links L S S1,
   complete-mapping S1 S1 M1 M2 F1 F2,
   decomp M2 M2 S1 F2 F3.
```

TODO: What is commit-links and complete-mapping?, maybe complete-mapping can be hidden in the code rendering? Decompiling an assignment requires to turn abstractions into lambdas. For aesthetic purposes we also eta-contract the result (not needed since $\mathcal{F}_o$ equality can do that)

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

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

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

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

INVARIANT 3. *TODO: dire che il mapping è bijective*

5.4 Definition of $≃_o$ and its properties

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

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

LEMMA 5.1 (COMPILATION ROUND TRIP). *If* comp S T [] M [] _ [] _ *then* decomp M T S

PROOF SKETCH. trivial, since the terms are beta normal beta just builds an app. □

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

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

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

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

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

PROOF SKETCH. Since progress1 is trivial fstep and hstep are the same, that is in this context where input terms are $\beta\eta$-normal and we disregard η-equivalence $\simeq_\lambda$ is equivalent to $\simeq_o$. □

5.5 Limitations of by this basic scheme

The basic compilation scheme is not about to deal wit the following problem:

$$\mathbb{P} = \{ \ \lambda xy.X \cdot y \cdot x \simeq_o \lambda xy.x \quad \lambda x.f \cdot (X \cdot x) \cdot x \simeq_o Y \ \}$$

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

6 HANDLING OF $\diamond\eta$

η-reduction is an equivalence relation where a term of the form $\lambda x.t \cdot x$ can be converted to t any time x does not occur as a free variable in t. We call t the η-contraction of $\lambda x.t \cdot x$.

Following the compilation scheme of section 5.1 the unification problem $\mathbb{P}$ is compiled as follows:

$$\mathbb{P} = \{ \ \lambda x.X \cdot x \simeq_o f \ \}$$
$$\mathbb{T} = \{ \ \lambda x.A_x \simeq_\lambda f \ \}$$
$$\mathbb{M} = \{ \ X \mapsto A^1 \ \}$$

While $\lambda x.X \cdot x \simeq_o f$ does admit the solution $\rho = \{ X \mapsto f \}$, the corresponding problem in $\mathbb{T}$ does not: lam x\ uva A [x] and con"f" start with different, rigid, term constructors hence $\simeq_\lambda$ fails.

In order to guarantee proposition 2.1 we detect lambdas that can disappear by eta contraction (section 6.1) and we modify the compiled terms by putting fresh unification variables in their place: the problematic term is moved from $\mathbb{T}$ to $\mathbb{L}$ (section 6.2). The compilation of the problem $\mathbb{P}$ above is refined to:

$$\mathbb{P} = \{ \ \lambda x.X \cdot x \simeq_o f \ \}$$
$$\mathbb{T} = \{ \ A \simeq_\lambda f \ \}$$
$$\mathbb{M} = \{ \ X \mapsto B^1 \ \}$$
$$\mathbb{L} = \{ \ \vdash A =_\eta \lambda x.B_x \ \}$$

As per invariant 2 the term on the left is a variable, and its right counterpart is the term in $\diamond\eta$, and that term has the following property:

INVARIANT 4 (link-η rhs). *The* rhs *of any* link-η *has the shape* $\lambda x.t$ *and* t *is not a lambda.*

link-η are kept in the link store $\mathbb{L}$ during execution and activated when some conditions hold on lhs or rhs. Link activation is implemented by extending the progress1 predicate (defined in section 5.2).

6.1 Detection of $\diamond\eta$

When compiling a term t we need to determine if any subterm $s \in \mathcal{P}(t)$ that is of the form $\lambda x.r$, where x occurs in r, can be a η-expansion, i.e. if there exists a substitution ρ such that $\rho(\lambda x.r) =_o s$. The detection of lambda abstractions that can "disappear" is not as trivial as it may seems, here a few examples:

$\lambda x.f \cdot (A \cdot x)$	$\in \diamond\eta$	$\rho = \{ A \mapsto \lambda x.x \}$
$\lambda x.f \cdot (A \cdot x) \cdot x$	$\in \diamond\eta$	$\rho = \{ A \mapsto \lambda x.a \}$
$\lambda x.f \cdot x \cdot (A \cdot x)$	$\notin \diamond\eta$	
$\lambda x.\lambda y.f \cdot (A \cdot x) \cdot (B \cdot y \cdot x)$	$\in \diamond\eta$	$\rho = \{ A \mapsto \lambda x.x, B \mapsto \lambda y.\lambda x.y \}$

The first two examples are easy, and show how a unification variable can expose or erase a variable and turn the resulting term in an η-expansion or not.
The third example shows that when a variable occurs outside the scope of a unification variable it cannot be erased and can hence prevent a term from being an η-expansion.
The last example shows the recursive nature of the check we need to implement. The term starts with a spine of two lambdas hence the whole term is in $\diamond\eta$ iff the inner term $\lambda y.f \cdot (A \cdot x) \cdot (B \cdot y \cdot x)$ is in $\diamond\eta$ itself. If it is, it could η-contract to $f \cdot (A \cdot x)$ making $\lambda x.f \cdot (A \cdot x)$ a potential η-expansion.

We can now define more formally how $\diamond\eta$ terms are detected together with its auxiliary functions:

E:the compiler must do beta normal!!!!

Definition 6.1 (may-contract-to). A term s *may-contract-to* a name x if there exists a substitution ρ such that $\rho s =_o x$.

LEMMA 6.2. *A β-normal term $s = \lambda x_1 \ldots x_n.t$ may-contract-to x only if one of the following three conditions holds:*

(1) *$n = 0$ and $t = x$;*
(2) *t is the application of x to a list of terms l and each l_i may-contract-to x_i (e.g. $\lambda x_1 \ldots x_n.x \cdot x_1 \ldots x_n =_o x$);*
(3) *t is a unification variable with scope W, and for any $v \in \{x, x_1 \ldots x_n\}$, there exists a $w_i \in W$, such that w_i may-contract-to v (if $n = 0$ this is equivalent to $x \in W$).*

PROOF SKETCH. Since our terms are in β-normal form there is only one rule that can play a role (namely η_l), hence if the term s is not exactly x (case 1) it can only be an η-expansion of x, or a unification variable that can be assigned to x, or a combination of both. If s begins with a lambda, then the lambda can only disappear by η contraction. In that case the term t under the spine of binders

for $x_1 \ldots x_n$ can either be x itself applied to terms that can *may-contract-to* these variables (case 2), or a unification variable that can be assigned to that application (case 3). □

Note that this condition does not require the term to be in $\mathcal{L}_\lambda$.

> E:Is this relevant

Definition 6.3 (occurs-rigidly). A name x *occurs-rigidly* in a β-normal term t, if $\forall \rho, x \in \mathcal{P}(\rho t)$

In other words x occurs-rigidly in t if it occurs in t outside of the scope of unification variables since an instantiation is allowed to discard x from the scope of the unification variable. Note that η-contraction cannot make x disappear, since the variables being erased by η-contraction are locally bound inside t.

We can now derive the implementation for $\diamond \eta$ detection:

Definition 6.4 (maybe-eta). Given a β-normal term $s = \lambda x_1 \ldots x_n.t$, *maybe-eta s* holds if any of the following holds:

(1) t is a constant or a variable applied to the arguments $l_1 \ldots l_m$ such that $m \geq n$ and for every i such that $1 \leq i \leq m - n$ the term l_i *may-contract-to* x_i, and no x_i *occurs-rigidly* in $l_1 \ldots l_{m-n-1}$;

(2) t is a unification variable with scope W and for each x_i there exists a $w_j \in W$ such that w_j *may-contract-to* x_i.

LEMMA 6.5 ($\diamond \eta$ DETECTION). *If t is a β-normal term and maybe-eta t holds, then $t \in \diamond \eta$.*

PROOF SKETCH. Follows from definition 6.3 and lemma 6.2 □

Remark that the converse of lemma 6.5 does not hold: there exists a term t satisfying the criteria (1) of definition 6.4 that is not in $\diamond \eta$, i.e. there exists no substitution ρ such that ρt is an η-expansion. A simple counter example is $\lambda x.f \cdot (A \cdot x) \cdot (A \cdot x)$ since x does not *occur-rigidly* in the first argument of f, and the second argument of f *may-contract-to* x. In other words $A \cdot x$ may either use or discard x, but our analysis does not take into account that *the same term* cannot have two contrasting behaviors.

As we will see in the rest of this section this is not a problem since it does not break proposition 2.1 nor proposition 2.2.

6.2 Compilation

The following rule is inserted just before rule (c_λ) from the code in section 5.1.

```
comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
  maybe-eta (flam F) [], !,
    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].
```

The rule triggers when the input term `flam F` is in $\diamond \eta$. It compiles it to `lam F1` but puts the fresh variable `A` in its place. The variable sees all the names free in `lam F1`. The critical part of this rule is the creation of the `link-η`, which relates the variable `A` with `lam F1`. This link clearly validates invariant 2.

COROLLARY 6.6. *The* rhs *of any* link-η *has exactly one lambda abstraction, hence the rule above respects invariant 4.*

PROOF SKETCH. By contradiction, suppose that the rule above triggered and that the rhs of the link is $\lambda x.\lambda y.t_{xy}$. If *maybe-eta* $\lambda y.t_{xy}$ holds the recursive call to `comp` (made by `comp-lam`) must have put a fresh variable in its place, so this case is impossible. Otherwise, if *maybe-eta* $\lambda y.t_{xy}$ does not hold, also *maybe-eta* $\lambda x.\lambda y.t_{xy}$ does not hold, contradicting the assumption that the rule triggered. □

6.3 Progress

link-η are meant to delay the unification of "problematic" terms until we know for sure if the head lambda has to be η-contracted or not.

Definition 6.7 (progress-η-left). A link $\Gamma \vdash X =_\eta T$ is removed from $\mathbb{L}$ when X becomes rigid. There are two cases:

(1) if $X = a$ or $X = y$ or $X = f \cdot a_1 \ldots a_n$ we unify the η-expansion of the X with T, that is we run $\lambda x.X \cdot x \simeq_\lambda T$ (under the context Γ)

> E:where y comes from? $X = y$: y is in the ctx of X

(2) if $X = \lambda x.t$ we run $X \simeq_\lambda T$.

Definition 6.8 (progress-η-right). A link $\Gamma \vdash X =_\eta T$ is removed from $\mathbb{L}$ when either 1) *maybe-eta T* does not hold (anymore) or 2) by η-contracting T to T', T' is a term not starting with the `lam` constructor. In the first case, X is unified with T and in the second one, X is unified with T' (under the context Γ).

There is a third case in which a link is removed from $\mathbb{L}$, namely when the lhs is assigned to a variable that is the lhs of another link-η.

Definition 6.9 (progress-η-deduplicate). A link $\Gamma \vdash X_{\vec{s}} =_\eta T$ is removed from $\mathbb{L}$ when another link $\Delta \vdash X_{\vec{r}} =_\eta T'$ is in $\mathbb{L}$. By invariant 1 the length of $\vec{s}$ and $\vec{r}$ is the same hence we can move the term T' from Δ to Γ by renaming its bound variables, i.e. $T'' = T'[\vec{r}/\vec{s}]$. We then run $T \simeq_\lambda T''$ (under the context Γ).

> D:Below the proof of proposition 2.8, ho usato 3 lemmi ausiliari, forse si può compattare in una prova più piccola?

LEMMA 6.10. *Given a* link-η *l, the unification done by* progress-η-left *is between terms in* $\mathcal{W}$

PROOF SKETCH. Let σ be the substitution, such that $\mathcal{W}(\sigma)$. lhs $\in \sigma$, therefore $\mathcal{W}(\text{lhs})$. By definition 6.7, if 1) lhs is a name, a constant of an application, then, lhs is unified with the η-reduced term t obtain from rhs. By corollary 6.6, rhs has one lambda, therefore $\mathcal{W}(t)$. Otherwise, 2) lhs has `lam` as functor, rhs should not be an η-expansion ans, so, $\mathcal{W}(\text{rhs})$. In both cases, unification is performed between terms in $\mathcal{W}$. □

LEMMA 6.11. *Given a* link-η *l, the unification done by* progress-η-right *is between terms in* $\mathcal{W}$.

PROOF SKETCH. lhs is variable, and, by definition 6.8, rhs is either no more a $\diamond \eta$, i.e. rhs is not a η-expansion and, so, $\mathcal{W}(\text{rhs})$. Otherwise, rhs can reduce to a term which cannot be a η-expansion, and, so, $\mathcal{W}(\text{rhs})$. In both cases, unification is done between terms in $\mathcal{W}$. □

LEMMA 6.12. *Given a* link-η *l, the unification done by* progress-η-deduplicate *is between terms in* $\mathcal{W}$.

PROOF. Trivial, since the unification is done between unification variables, which are by definition in $\mathcal{W}$. □

LEMMA 6.13. *Proposition 2.8 holds, i.e., given a substitution σ and a* link-η *l, after the activation of l,* $\mathcal{W}(\sigma)$ *holds.*

PROOF SKETCH. By lemmas 6.10 to 6.12, every unification performed by the activation of a link-η is performed between terms in $\mathcal{W}$, therefore, the substitution remains $\mathcal{W}$. □

LEMMA 6.14. progress *terminates.*

PROOF SKETCH. Rules definitions 6.7 and 6.8 and definition 6.9 remove one link from $\mathbb{L}$, hence they cannot be applied indefinitely. Moreover each rule only relies on terminating operations such as $\simeq_\lambda$, η-contraction, η-expansion, relocation (a recursive copy of a finite term). □

D:Should we proove simulation fidelity for link-η insertion?

Example of progress-η-left. The example at the beginning of section 6, once $\sigma = \{ A \mapsto f \}$, triggers this rule since the link becomes $\vdash f =_\eta \lambda x.B_x$ and the lhs is a constant. In turn the rule runs $\lambda x.f \cdot x \simeq_\lambda \lambda x.B_x$, resulting in $\sigma = \{ A \mapsto f ; B_x \mapsto f \}$. Decompilation the generates $\rho = \{ X \mapsto f \}$, since X is mapped to B and f is the η-contracted version of $\lambda x.f \cdot x$.

Example of progress-η-deduplicate. A very basic example of link-η deduplication, is given below:

$$\mathbb{P} = \{ \; \lambda x.(X \cdot x) \; \simeq_o \; \lambda x.(Y \cdot x) \; \}$$
$$\mathbb{T} = \{ \qquad A \; \simeq_\lambda \; C \qquad \}$$
$$\mathbb{M} = \{ \; X \mapsto B^1 \qquad Y \mapsto D^1 \; \}$$
$$\mathbb{L} = \{ \; \vdash A =_\eta \lambda x.B_x \quad \vdash C =_\eta \lambda x.D_x \; \}$$

The result of $A \simeq_\lambda C$ is that the two link-η share the same lhs. By unifying the two rhs we get $\sigma = \{A \mapsto C, B \mapsto D \}$. In turn, given the map $\mathbb{M}$, this second assignment is decompiled to $\rho = \{ X \mapsto Y \}$ as expected.

We delay at the end of next section an example of link-η progression due to *progress-η-right*

7 ENFORCING INVARIANT 1

We report here the problem given in section 5.5 where X is used with two different arities and the output of the compilation does not respect invariant 3 (merging the two mappings for s would break invariant 1). In this section we explain how to replace the duplicate mapping with some link-η in order to restore the invariants.

$$\mathbb{P} = \{ \; \lambda x.\lambda y.(X \cdot y \cdot x) \; \simeq_o \; \lambda x.\lambda y.x \qquad \lambda x.(f \cdot (X \cdot x) \cdot x) \; \simeq_o \; Y \; \}$$
$$\mathbb{T} = \{ \qquad A \; \simeq_\lambda \; \lambda x.\lambda y.x \qquad\qquad D \; \simeq_\lambda \; F \; \}$$
$$\mathbb{M} = \{ \; X \mapsto E^1 \qquad Y \mapsto F^0 \qquad X \mapsto C^2 \; \}$$
$$\mathbb{L} = \left\{ \begin{array}{ll} \vdash D =_\eta \lambda x.(f \cdot E_x \cdot x) & \vdash A =_\eta \lambda x.B_x \\ x \vdash B_x =_\eta \lambda y.C_{yx} \end{array} \right\}$$

We see that the maybe-eta as identified $\lambda xy.X \cdot y \cdot x$ and $\lambda x.f \cdot (X \cdot x) \cdot x$ and the compiler has replaced them with A and D respectively. However, the mapping $\mathbb{M}$ breaks invariant 3: the $\mathcal{F}_o$ variable X is mapped

to two different $\mathcal{H}_o$ variables. To address this problem we adjust the compiler's output with a map-deduplication procedure.

Definition 7.1 (align-arity). Given two mappings $m_1 : X \mapsto A^m$ and $m_2 : X \mapsto C^n$ where $m < n$ and $d = n - m$, *align-arity* m_1 m_2 generates the following d links, one for each i such that $0 \le i < d$,

$$x_0 \ldots x_{m+i} \vdash B^i_{x_0 \ldots x_{m+i}} =_\eta \lambda x_{m+i+1}.B^{i+1}_{x_0 \ldots x_{m+i+1}}$$

where B^i is a fresh variable of arity $m + i$, and $B^0 = A$ as well as $B^d = C$.

The intuition is that we η-expand the occurrence of the variable with lower arity to match the higher arity. Since each link-η can add exactly one lambda, we need as many links as the difference between the two arities.

Definition 7.2 (map-deduplication). Forall mappings $m_1, m_2 \in \mathbb{M}$ such that $m_1 : X \mapsto A^m$ and $m_2 : X \mapsto C^n$ and $m < n$ we remove m_1 from $\mathbb{M}$ and add to $\mathbb{L}$ the result of *align-arity* m_1 m_2.

If we look back the example give at the beginning of this section, we can deduplicate $X \mapsto E^1, X \mapsto C^2$ by removing the first mapping and adding the auxiliary link-η: $x \vdash E_x =_\eta \lambda y.C_{xy}$. After deduplication the compiler output is as follows:

$$\mathbb{P} = \{ \; \lambda x.\lambda y.(X \cdot y \cdot x) \; \simeq_o \; \lambda x.\lambda y.x \qquad \lambda x.(f \cdot (X \cdot x) \cdot x) \; \simeq_o \; Y \; \}$$
$$\mathbb{T} = \{ \qquad A \; \simeq_\lambda \; \lambda x.\lambda y.x \qquad\qquad D \; \simeq_\lambda \; F \; \}$$
$$\mathbb{M} = \{ \; Y \mapsto F^0 \qquad X \mapsto C^2 \; \}$$
$$\mathbb{L} = \left\{ \begin{array}{ll} x \vdash E_x =_\eta \lambda y.C_{xy} & \vdash D =_\eta \lambda x.(f \cdot E_x \cdot x) \\ \vdash A =_\eta \lambda x.B_x & x \vdash B_x =_\eta \lambda y.C_{yx} \end{array} \right\}$$

In this example, $\mathbb{T}_1$ assigns A which triggers $\mathbb{L}_3$ and then $\mathbb{L}_4$ by definition 6.7. C_{yx} is therefore assigned to x (the second variable of its scope). We can finally see the *progress-η-right* of $\mathbb{L}_1$: its rhs is now $\lambda y.y$ (C_{xy} gives y). Since it is no more in $\diamond \eta$, $\lambda y.y$ is unified with E_x. Moreover, $\mathbb{L}_2$ is also triggered due to definition 6.8: $\lambda x.(f \cdot (\lambda y.y) \cdot x)$ is η-reducible to $f \cdot (\lambda y.y)$ which is a term not starting with the lam constructor.

E:dire che preserviamo l'invariante che tutte le variable sono fully-applied

8 HANDLING OF $\overline{\mathcal{L}_\lambda}$

D:I've rewritten it, it is clearer?

Until now, we have only dealt we unification of terms in $\mathcal{L}_\lambda$. However, we want the unification relation to be more robust so that it can work with terms in $\overline{\mathcal{L}_\lambda}$. In general, unification in $\overline{\mathcal{L}_\lambda}$ admits more then one solution and committing one of them in the substitution does not guarantee prop. (*complete*). For instance, $X \cdot a \simeq_o a$ is a unification problem admits two different substitutions: $\rho_1 = \{X \mapsto \lambda x.x\}$ and $\rho_2 = \{X \mapsto \lambda\_.a\}$. Prefer one over the other may break future unifications.

It is the case that, given a list of unification problems, $\mathbb{P}_1 \ldots \mathbb{P}_n$ with $\mathbb{P}_n$ in $\overline{\mathcal{L}_\lambda}$, the resolution of $\bigwedge_{i=0}^{n-1} \mathbb{P}_i$ gives a partial substitution ρ, such that $\rho \mathbb{P}_n$ falls again in $\mathcal{L}_\lambda$.

$$\mathbb{P} = \{ \; X \; \simeq_o \; \lambda x.Y \qquad (X \cdot a) \; \simeq_o \; a \; \}$$
$$\mathbb{T} = \{ \; A \; \simeq_\lambda \; \lambda x.B \qquad (A \cdot a) \; \simeq_\lambda \; a \; \}$$
$$\mathbb{M} = \{ \; Y \mapsto B^0 \qquad X \mapsto A^0 \; \}$$

In the example above, we see that $\mathbb{P}_1$ instantiates X so that $\mathbb{P}_2$, can be solved in $\mathcal{L}_\lambda$.

E:it is even a ground term, there is no unification left to perform actually

D:i don't understand the note

On the other hand, we see that, $\simeq_\lambda$ can't solve the compiled problems $\mathbb{T}$. In fact, the resolution of $\mathbb{T}_1$ gives the substitution $\sigma = \{A \mapsto \lambda x.B\}$, but the dereferencing of $\mathbb{T}_2$ gives the non-unifiable problem $(\lambda x.B) \cdot a \simeq_\lambda a$.

To address this unification problem, term compilation should capture the terms t in $\overline{\mathcal{L}_\lambda}$ and replace them with fresh variables X. The variables X and the terms t are linked through a link-β.

link-β guarantees invariant 2 and the term on the rhs has the following property:

D:Is it clearer?

INVARIANT 5 (link-β rhs). *The* rhs *of any* link-β *has the shape* $X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$ *such that* X *is a unification variable with scope* $s_1 \ldots s_n$ *and* $t_1 \ldots t_m$ *is a list of terms. This is equivalent to* app[uva X S | L]. *where* S $= s_1 \ldots s_n$ *and* L $= t_1 \ldots t_m$.

LEMMA 8.1. *If the* lhs *of a* link-β *is instantiated to a rigid term and its* rhs *counterpart is still in* $\overline{\mathcal{L}_\lambda}$, *the original unification problem is not in* $\mathcal{L}_\lambda$ *and the unification fails.*

PROOF SKETCH. Given $X \cdot t_1 \ldots t_n \simeq_\lambda t$ where t is a rigid term and $t_1 \ldots t_n$ is not in $\mathcal{L}_\lambda$. By construction, $X \cdot t_1 \ldots t_n$ is replaced with a variable Y, and the link-β $\Gamma \vdash Y =_\beta X \cdot t_1 \ldots t_n$ is created. The unification instantiates Y to t, making the lhs of the link a rigid term, while rhs is still in $\overline{\mathcal{L}_\lambda}$. The original problem is in fact outside $\mathcal{L}_\lambda$.   □

8.1 Compilation

Detection of $\overline{\mathcal{L}_\lambda}$ is quite simple to implement in the compiler, since it is sufficient to detect applications with flexible head and argument that are not in $\mathcal{L}_\lambda$.

```
comp (fapp [fuva A|Ag]) (uva B Scope) M1 M3 L1 L3 S1 S4 :- !,
  pattern-fragment-prefix Ag Pf Extra,
  len Pf Arity,
  alloc S1 B S2,
  m-alloc (fv A) (hv C (arity Arity)) M1 M2 S2 S3,
  fold6 comp Pf    Pf1    M2 M2 L1 L1 S3 S3,
  fold6 comp Extra Extra1 M2 M3 L1 L2 S3 S4,
  Beta = app [uva C Pf1 | Extra1],
  get-scope Beta Scope,
  L3 = [val (link-beta (uva B Scope) Beta) | L2].
```

The list Ag is split into the list Pf and Extra such that append Pf Extra Ag and Pf is the largest prefix of Ag such that Pf is in $\mathcal{L}_\lambda$. The rhs of the link-β is the application of a fresh variable C having in scope all the free variables appearing in the compiled version of Pf and Extra. The variable B, returned has the compiled term, is a fresh variable having in scope all the free variables occurring in Pf1 and Extra1

INVARIANT 6. *The* rhs *of a* link-β *has the shape* $X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$.

COROLLARY 8.2. *Let* $X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$ *be the* rhs *of a* link-β, *then* $m > 0$.

PROOF SKETCH. Assume we have a link-β, by contradiction, if $m = 0$, then the original $\mathcal{F}_o$ term has the shape fapp[fuva M | Ag] where Ag is a list of distinct names (i.e. the list Extra is empty). This case is however captured by rule (c_λ) (from section 5.1) and no link-β is produced which contradicts our initial assumption.   □

COROLLARY 8.3. *Let* $X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$ *be the* rhs *of a* link-β, *then* t_1 *either appears in* $s_1 \ldots s_n$ *or it is not a name.*

PROOF SKETCH. By construction, the lists $s_1 \ldots s_n$ and $t_1 \ldots t_m$ are built by splitting the list Ag from the original term fapp [fuva A|Ag]. $s_1 \ldots s_n$ is the longest prefix of the compiled terms in Ag which is in $\mathcal{L}_\lambda$. Therefore, by definition of $\mathcal{L}_\lambda$, t_1 must appear in $s_1 \ldots s_n$, otherwise $s_1 \ldots s_n$ is not the longest prefix in $\mathcal{L}_\lambda$, or it is a term with a constructor of tm as functor.   □

E:Dire che maybe eta fa il detect anche su termini che non sono il llambda, oppure dirlo in section of maybeeta + dare un esempio?

8.2 Progress

The activation of a link-β is performed when its rhs falls under $\mathcal{L}_\lambda$ under a given substitution.

Definition 8.4 (progress-beta-$\mathcal{L}_\lambda$). Given a substitution σ and a link-β $\Gamma \vdash T =_\beta X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$ such that σt_1 is a name, say t, and $t \notin s_1 \ldots s_n$. If $m = 0$, then the link-β is removed and lhs is unified with $X_{s_1 \ldots s_n}$. If $m > 0$, then the link-β is replaced by a refined version $\Gamma \vdash T =_\beta Y_{s_1 \ldots s_n, t} t_2 \ldots t_m$ with reduced list of arguments and Y being a fresh variable. Moreover, the new link-β, $\Gamma \vdash X_{s_1 \ldots s_n} =_\eta \lambda x.Y_{s_1 \ldots s_n, x}$ is added to $\mathbb{L}$.

D:L'ho riri-for-mu-lato

Definition 8.5 (progress-beta-rigid-head). A link $\Gamma \vdash X =_\beta X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$ is removed from $\mathbb{L}$ if $X_{s_1 \ldots s_n}$ is instantiated to a term t and the β-reduced term t' obtained from the application of t to $l_1 \ldots l_m$ is in $\mathcal{L}_\lambda$. Moreover, X is unified to t.

LEMMA 8.6. progress *terminates*

PROOF SKETCH. Let l a link-β in the store $\mathbb{L}$. If l is activated by *progress-beta-rigid-head*, then it disappears from $\mathbb{L}$ and progress terminates. Otherwise, the rhs of l is made by a variable applied to m arguments. At each activation of *progress-beta-$\mathcal{L}_\lambda$*, l is replaced by a new link-β l^1 having $m - 1$ arguments. At the mth iteration, the link-β l^m has no more arguments and is removed from $\mathbb{L}$. Note that at the mth iteration, m new link-η have been added to $\mathbb{L}$, however, by lemma 6.14, the algorithm terminates.

D:L'ho riscritto

E:funziona. per essere più precisi io parlerei di ordine lessicografico (tipico ordine ben fondato usato per dimostrare terminazione). Nl nostro caso è la tripla (argomenti extra dei beta, numero di beta, numero di eta).

□

COROLLARY 8.7. *Given a* link-β, *the variables occurring in its* rhs *are in* $\mathcal{L}_\lambda$.

D:is it clearer?

PROOF SKETCH. By construction, the rhs of link-β has the shape $X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$, $s_1 \ldots s_n$ is in $\mathcal{L}_\lambda$ and all the terms $t_1 \ldots t_n$ are in

$\mathcal{L}_\lambda$, too. If a `link-`β is triggered by *progress-beta-rigid-head*, then, by definition 8.5, that link is removed by $\mathbb{L}$, and the property is satisfied. If the `link-`η is activated by *progress-beta-*$\mathcal{L}_\lambda$, then, by definition 8.4, the new `link-`β as a variable as a scope which is still in $\mathcal{L}_\lambda$. □

LEMMA 8.8. *Given a* $\mathbb{T}$ *and a substitution* σ *then the resolution of* $\sigma\mathbb{T}$ *guarantees proposition 2.2*

PROOF SKETCH. If $\sigma\mathbb{T}$ is in $\mathcal{L}_\lambda$, then by definitions 8.4 and 8.5, then `link-`β disappear and the unification done between terms in $\mathcal{L}_\lambda$. This problem unifies iff its corresponding $\mathcal{F}_o$ problem unifies too. If $\sigma\mathbb{T}$ is in $\overline{\mathcal{L}_\lambda}$, then, by lemma 8.1, the unification fails, as per the corresponding unification in $\mathcal{F}_o$. □

Example of progress-beta-$\mathcal{L}_\lambda$. Consider the `link-`β below:

$$\mathbb{P} = \{ \ X \simeq_o \ \lambda x.x \quad \lambda x.(Y\cdot(X\cdot x)) \simeq_o \ f \ \}$$
$$\mathbb{T} = \{ \ A \simeq_\lambda \ \lambda x.x \qquad\qquad B \simeq_\lambda \ f \ \}$$
$$\mathbb{M} = \{ \ Y \mapsto D^0 \quad X \mapsto A^0 \ \}$$
$$\mathbb{L} = \left\{ \begin{array}{ll} \vdash \ A =_\eta \ \lambda x.E_x & \vdash B =_\eta \ \lambda x.C_x \\ x \vdash C_x =_\beta \ (D\cdot E_x) \end{array} \right\}$$

Initially the `link-`β rhs is a variable D applied to the E_x. The first unification problem results in $\sigma = \{A \mapsto \lambda x.x\}$. In turn this instantiation triggers $\mathbb{L}_1$ by *progress-η-left* and E is assigned to $\lambda x.x$. Under this substitution the `link-`β becomes $x \vdash C_x =_\beta (D\cdot x)$, and by *progress-beta-*$\mathcal{L}_\lambda$ it is replaced with the link: $\vdash E =_\eta \lambda x.D_x$, while C_x is unified with D_x. The second unification problem assigns f to B, that in turn activates the second `link-`η (f is assigned to C), and then all the remaining links are solved. The final $\mathcal{H}_o$ substitution is $\sigma = \{A \mapsto \lambda x.x, B \mapsto f, C_x \mapsto (f\ x), D \mapsto f, E_x \mapsto x, F_x \mapsto C_x\}$ and is decompiled into $\rho = \{X \mapsto \lambda x.x, Y \mapsto f\}$.

Example of progress-beta-rigid-head. We can take the example provided in section 8. The problem is compiled into:

$$\mathbb{P} = \{ \ X \simeq_o \ \lambda x.Y \quad (X\cdot a) \simeq_o \ a \ \}$$
$$\mathbb{T} = \{ \ A \simeq_\lambda \ \lambda x.B \qquad C \simeq_\lambda \ a \ \}$$
$$\mathbb{M} = \{ \ Y \mapsto B^0 \quad X \mapsto A^0 \ \}$$
$$\mathbb{L} = \{ \ \vdash C =_\beta \ (A\ a) \ \}$$

The first unification problems is solved by the substitution $\sigma = \{A \mapsto \lambda x.B\}$. The `link-`$\beta$ becomes $\vdash C =_\beta ((\lambda x.B)\ a)$ whose rhs can be β-reduced to B. B is in $\mathcal{L}_\lambda$ and is unified with C. The resolution of the second unification problem gives the final substitution $\sigma = \{A \mapsto \lambda x.B, B \mapsto C, C \mapsto a\}$ which is decompiled into $\rho = \{X \mapsto \lambda x.a, Y \mapsto a\}$.

8.3 Relaxing lemma 8.1

Working with terms in $\mathcal{L}_\lambda$ is sometime too restrictive. There exists systems such as λProlog [12], Abella [7], which delay the resolution of $\overline{\mathcal{L}_\lambda}$ unification problems if the substitution is not able to put them in $\mathcal{L}_\lambda$.

$$\mathbb{P} = \{ \ (X\cdot a) \simeq_o \ a \quad X \simeq_o \ \lambda x.Y \ \}$$

In the example above, $\mathbb{P}_1$ is in $\overline{\mathcal{L}_\lambda}$ and the object language cannot solve it and, by proposition 2.2, the meta language neither. However, we can be more permissive, and relax lemma 8.1. This modification is quite simple to manage: we are introducing a new $\overline{\mathcal{L}_\lambda}$ progress

[margin annotation: E:alla fine non si usa anche la regola beta-rigid-head?]

[margin annotation: D:nope. Si risolve tutto by progress-beta-$\mathcal{L}_\lambda$]

rule, say *progress-beta-*$\overline{\mathcal{L}_\lambda}$, by which, if lhs is rigid and rhs is flexible, the considered `link-`β is kept in the store and no progression is done[4]. *progress-beta-*$\overline{\mathcal{L}_\lambda}$ makes `occur-check-links` partial, since the check is possible only on links with a variable on the lhs. This means that we can have two links $\vdash X =_\beta Y\cdot a$ and $\vdash f\cdot X =_\beta Y\cdot a$ where the occur check does not throw an error. Note however, that the decompilation of the two links will force the unification of X to $Y\cdot a$ and then the unification of $f\cdot(Y\cdot a)$ to $Y\cdot a$, which fails by the occur check of $\simeq_\lambda$.

A second strategy to deal with problem that are in $\overline{\mathcal{L}_\lambda}$ is to make some approximation. This is the case for example of the unification algorithm of Coq used in its type class solver [18]. The approximation consists in forcing a choice (among the others) when the unification problem is in $\overline{\mathcal{L}_\lambda}$. For instance, in $X\cdot a\cdot b = Y\cdot b$, the last argument of the two terms is the same, therefore Y is assigned to Xa. Note that this is of course an approximation, since $\sigma = \{X = \lambda x.Y, Y = \_\}$ is another valid substitution for the original problem. This approximation can be easily introduced in our unification procedure, by adding new custom `link-`β progress rules.

9 PRACTICAL EXAMPLE

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] Andrew Gacek. *The Abella Interactive Theorem Prover (System Description)*. 2008. arXiv: 0803.2305 [cs.LO].

---

[4]This new rule trivially guarantees the termination of progress

[8] 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.

[9] 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.

[10] 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.

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

[12] Gopalan Nadathur and Dale Miller. "An Overview of Lambda-Prolog". In: June 1988, pp. 810–827.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] Matthieu Sozeau and Nicolas Oury. "First-Class Type Classes". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-71067-7.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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).
```

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

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

```
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 :-                                          (ηₗ)
  pi x\ beta T [x] (T' x), x =ₒ x => F x =ₒ T' x.
T =ₒ flam F :-                                          (ηᵣ)
  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.   (βₗ)
T =ₒ fapp [flam X | L] :- beta (flam X) L R, T =ₒ R.   (βᵣ)
```

```
type extend-subst fm -> fsubst -> fsubst -> o.
extend-subst (fuva N) S S' :- mem.alloc N S S'.
extend-subst (flam F) S S' :-
  pi x\ (pi S\extend-subst x S S) => extend-subst (F x) S S'.
extend-subst (fcon _) S S.
extend-subst (fapp L) S S1 :- fold extend-subst L S S1.
```

```
type beta fm -> list fm -> fm -> o.
beta A [] A.
beta (flam Bo) [H | L] R :- napp (Bo H) F, beta F 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 napp fm -> fm -> o.
napp (fcon C) (fcon C).
napp (fuva A) (fuva A).
napp (flam F) (flam G) :- pi x\ napp (F x) (G x).
napp (fapp [fapp L1 |L2]) T :- !,
  append L1 L2 L3, napp (fapp L3) T.
napp (fapp L) (fapp L1) :- map napp L L1.
napp N N :- name N.
```

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

```
type eta-contract fm -> fm -> o.
eta-contract (fcon X) (fcon X).
eta-contract (fapp L) (fapp L1) :- map eta-contract L L1.
eta-contract (flam F) T :- eta-contract-aux [] (flam F) T.
eta-contract (flam F) (flam F1) :-
```

```
1625      pi x\ eta-contract x x => eta-contract (F x) (F1 x).
1626    eta-contract (fuva X) (fuva X).
1627    eta-contract X X :- name X.
1628
1629    type eta-contract-aux list fm -> fm -> fm -> o.
1630    eta-contract-aux L (flam F) T :-
1631      pi x\ eta-contract-aux [x|L] (F x) T. % also checks H Prefix does not x depend x
1632    eta-contract-aux L (fapp [H|Args]) T :-
1633      rev L LRev, append Prefix LRev Args,
1634      if (Prefix = []) (T = H) (T = fapp [H|Prefix]).
```

13   THE META LANGUAGE

```
1638    kind inctx type -> type.                          (· ⊢ ·)
1639    type abs (tm -> inctx A) -> inctx A.
1640    type val A -> inctx A.
1641    typeabbrev assignment (inctx tm).
1642    typeabbrev subst (mem assignment).
1643
1644    kind tm  type.
1645    type app list tm -> tm.
1646    type lam (tm -> tm) -> tm.
1647    type con string -> tm.
1648    type uva addr -> list tm -> tm.
1649
1650    type (≃λ) tm -> tm -> subst -> subst -> o.
1651    (con C ≃λ con C) S S.
1652    (app L1 ≃λ app L2) S S1 :- fold2 (≃λ) L1 L2 S S1.
1653    (lam F1 ≃λ lam F2) S S1 :-
1654      pi x\ (pi S\ (x ≃λ x) S S) => (F1 x ≃λ F2 x) S S1.
1655    (uva N Args ≃λ T) S S1 :-
1656      set? N S F,!, move F Args T1, (T1 ≃λ T) S S1.
1657    (T ≃λ uva N Args) S S1 :-
1658      set? N S F,!, move F Args T1, (T ≃λ T1) S S1.
1659    (uva M A1 ≃λ uva N A2) S1 S2 :- !,
1660      pattern-fragment A1, pattern-fragment A2,
1661      prune! M A1 N A2 S1 S2.
1662    (uva N Args ≃λ T) S S1 :- not_occ N S T, pattern-fragment Args,
1663      bind T Args T1, assign N S T1 S1.
1664    (T ≃λ uva N Args) S S1 :- not_occ N S T, pattern-fragment Args,
1665      bind T Args T1, assign N S T1 S1.
1666
1667    type prune! addr -> list tm -> addr ->
1668                list tm -> subst -> subst -> o.
1669    /* no pruning needed */
1670    prune! N A  N A  S S :- !.
1671    prune! M A  N A  S1 S2 :- !, bind (uva M A) A Ass,
1672      assign N S1 Ass S2.
1673    /* prune different arguments */
1674    prune! N A1 N A2 S1 S3 :- !,
1675      new S1 W S2, prune-same-variable W A1 A2 [] Ass,
1676      assign N S2 Ass S3.
1677    /* prune to the intersection of scopes */
1678    prune! N A1 M A2 S1 S4 :- !,
1679      new S1 W S2, prune-diff-variables W A1 A2 Ass1 Ass2,
1680      assign N S2 Ass1 S3,
1681      assign M S3 Ass2 S4.
1682
```

```
1683
1684    type prune-same-variable addr -> list tm -> list tm ->
1685                            list tm -> assignment -> o.
1686    prune-same-variable N [] [] ACC (val (uva N Args)) :-
1687      rev ACC Args.
1688    prune-same-variable N [X|XS] [X|YS] ACC (abs F) :-
1689      pi x\ prune-same-variable N XS YS [x|ACC] (F x).
1690    prune-same-variable N [_|XS] [_|YS] ACC (abs F) :-
1691      pi x\ prune-same-variable N XS YS ACC (F x).
1692
1693    type permute list nat -> list tm -> list tm -> o.
1694    permute [] _ [].
1695    permute [P|PS] Args [T|TS] :-
1696      nth P Args T,
1697      permute PS Args TS.
1698
1699    type build-perm-assign addr -> list tm -> list bool ->
1700                    list nat -> assignment -> o.
1701    build-perm-assign N ArgsR [] Perm (val (uva N PermutedArgs)) :-
1702      rev ArgsR Args, permute Perm Args PermutedArgs.
1703    build-perm-assign N Acc [tt|L] Perm (abs T) :-
1704      pi x\ build-perm-assign N [x|Acc] L Perm (T x).
1705    build-perm-assign N Acc [ff|L] Perm (abs T) :-
1706      pi x\ build-perm-assign N Acc L Perm (T x).
1707
1708    type keep list A -> A -> bool -> o.
1709    keep L A tt :- mem L A, !.
1710    keep _ _ ff.
1711
1712    type prune-diff-variables addr -> list tm -> list tm ->
1713                    assignment -> assignment -> o.
1714    prune-diff-variables N Args1 Args2 Ass1 Ass2 :-
1715      map (keep Args2) Args1 Bits1,
1716      map (keep Args1) Args2 Bits2,
1717      filter Args1 (mem Args2) ToKeep1,
1718      filter Args2 (mem Args1) ToKeep2,
1719      map (index ToKeep1) ToKeep1 IdPerm,
1720      map (index ToKeep1) ToKeep2 Perm21,
1721      build-perm-assign N [] Bits1 IdPerm Ass1,
1722      build-perm-assign N [] Bits2 Perm21 Ass2.
1723
1724    type beta tm -> list tm -> tm -> o.
1725    beta A [] A :- !.
1726    beta (lam Bo) [H | L] R :- beta (Bo H) L R1, beta-aux R1 R.
1727    beta (app A) L (app X) :- append A L X.
1728    beta (con H) L (app [con H | L]).
1729    beta X L (app[X|L]) :- name X.
1730
1731    type beta-aux tm -> tm -> o.
1732    beta-aux (app [HD|TL]) R :- !, beta HD TL R.
1733    beta-aux A A.
1734
1735    /* occur check for N before crossing a functor */
1736    type not_occ  addr -> subst -> tm -> o.
1737    not_occ N S (uva M Args) :- set? M S F,
1738      move F Args T, not_occ N S T.
1739    not_occ N S (uva M Args) :- unset? M S, not (M = N),
1740
```

```
1741      forall1 (not_occ_aux N S) Args.
1742   not_occ _ _ (con _).
1743   not_occ N S (app L) :- not_occ_aux N S (app L).
1744   /* Note: lam is a functor for the meta language! */
1745   not_occ N S (lam L) :- pi x\ not_occ_aux N S (L x).
1746   not_occ _ _ X :- name X.
1747   /* finding N is ok */
1748   not_occ N _ (uva N _).
1749
1750   /* occur check for X after crossing a functor */
1751   type not_occ_aux addr -> subst -> tm -> o.
1752   not_occ_aux N S (uva M _) :- unset? M S, not (N = M).
1753   not_occ_aux N S (uva M Args) :- set? M S F,
1754      move F Args T, not_occ_aux N S T.
1755   not_occ_aux N S (app L) :- forall1 (not_occ_aux N S) L.
1756   not_occ_aux N S (lam F) :- pi x\ not_occ_aux N S (F x).
1757   not_occ_aux _ _ (con _).
1758   not_occ_aux _ _ X :- name X.
1759   /* finding N is ko, hence no rule */
1760
1761   /* copy T T' vails if T contains a free variable, i.e. it
1762      performs scope checking for bind */
1763   type copy tm -> tm -> o.
1764   copy (con C)   (con C).
1765   copy (app L)   (app L') :- map copy L L'.
1766   copy (lam T)   (lam T') :- pi x\ copy x x => copy (T x) (T' x).
1767   copy (uva A L) (uva A L') :- map copy L L'.
1768
1769   type bind tm -> list tm -> assignment -> o.
1770   bind T [] (val T') :- copy T T'.
1771   bind T [X | TL] (abs T') :- pi x\ copy X x => bind T TL (T' x).
1772
1773   type deref subst -> tm -> tm -> o.              (σt)
1774   deref _ (con C) (con C).
1775   deref S (app A) (app B) :- map (deref S) A B.
1776   deref S (lam F) (lam G) :-
1777      pi x\ deref S x x => deref S (F x) (G x).
1778   deref S (uva N L) R :- set? N S A,
1779      move A L T, deref S T R.
1780   deref S (uva N A) (uva N B) :- unset? N S,
1781      map (deref S) A B.
1782
1783   type move assignment -> list tm -> tm -> o.
1784   move (abs Bo) [H|L] R :- move (Bo H) L R.
1785   move (val A)  [] A.
1786
1787
1788   type deref-assmt subst -> assignment -> assignment -> o.
1789   deref-assmt S (abs T) (abs R) :- pi x\ deref-assmt S (T x) (R x).
1790   deref-assmt S (val T) (val R) :- deref S T R.
1791
```

14 THE COMPILER

```
1794   kind arity type.
1795   type arity nat -> arity.
1796
1797   kind fvariable type.
1798
```

```
1799   type fv addr -> fvariable.
1800
1801   kind hvariable type.
1802   type hv addr -> arity -> hvariable.
1803
1804   kind mapping type.
1805   type mapping fvariable -> hvariable -> mapping.
1806   typeabbrev mmap (list mapping).
1807
1808   typeabbrev scope (list tm).
1809   typeabbrev inctx ho.inctx.
1810   kind baselink type.
1811   type link-eta  tm -> tm -> baselink.
1812   type link-beta tm -> tm -> baselink.
1813   typeabbrev link (inctx baselink).
1814   typeabbrev links (list link).
1815
1816   macro @val-link-eta T1 T2 :- ho.val (link-eta T1 T2).
1817   macro @val-link-beta T1 T2 :- ho.val (link-beta T1 T2).
1818
1819
1820
1821   type occurs-rigidly fm -> fm -> o.
1822   occurs-rigidly N N.
1823   occurs-rigidly _ (fapp [fuva _|_]) :- !, fail.
1824   occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.
1825   occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).
1826
1827   type reducible-to list fm -> fm -> fm -> o.
1828   reducible-to _ N N :- !.
1829   reducible-to L N (fapp[fuva _|Args]) :- !,
1830      forall1 (x\ exists (reducible-to [] x) Args) [N|L].
1831   reducible-to L N (flam B) :- !,
1832      pi x\ reducible-to [x | L] N (B x).
1833   reducible-to L N (fapp [N|Args]) :-
1834      last-n {len L} Args R,
1835      forall2 (reducible-to []) R {rev L}.
1836
1837   type maybe-eta fm -> list fm -> o.              (◇η)
1838   maybe-eta (fapp[fuva _|Args]) L :- !,
1839      forall1 (x\ exists (reducible-to [] x) Args) L, !.
1840   maybe-eta (flam B) L :- !, pi x\ maybe-eta (B x) [x | L].
1841   maybe-eta (fapp [T|Args]) L :- (name T; T = fcon _),
1842      split-last-n {len L} Args First Last,
1843      none (x\ exists (y\ occurs-rigidly x y) First) L,
1844      forall2 (reducible-to []) {rev L} Last.
1845
1846
1847   type locally-bound tm -> o.
1848   type get-scope-aux tm -> list tm -> o.
1849   get-scope-aux (con _) [].
1850   get-scope-aux (uva _ L) L1 :-
1851      forall2 get-scope-aux L R,
1852      flatten R L1.
1853   get-scope-aux (lam B) L1 :-
1854      pi x\ locally-bound x => get-scope-aux (B x) L1.
1855   get-scope-aux (app L) L1 :-
1856
```

```
1857      forall2 get-scope-aux L R,
1858      flatten R L1.
1859    get-scope-aux X [X] :- name X, not (locally-bound X).
1860    get-scope-aux X [] :- name X, (locally-bound X).
1861
1862    type names1 list tm -> o.
1863    names1 L :-
1864      names L1,
1865      new_int N,
1866      if (1 is N mod 2) (L1 = L) (rev L1 L).
1867
1868    type get-scope tm -> list tm -> o.
1869    get-scope T Scope :-
1870      get-scope-aux T ScopeDuplicata,
1871      undup ScopeDuplicata Scope.
1872    type rigid fm -> o.
1873    rigid X :- not (X = fuva _).
1874
1875    type comp-lam (fm -> fm) -> (tm -> tm) ->
1876      mmap -> mmap -> links -> links -> subst -> subst -> o.
1877    comp-lam F G M1 M2 L1 L3 S1 S2 :-
1878      pi x y\ (pi M L S\ comp x y M M L L S S) =>
1879        comp (F x) (G y) M1 M2 L1 (L2 y) S1 S2,
1880      close-links L2 L3.
1881
1882    type close-links (tm -> links) -> links -> o.
1883    close-links (v\[X  |L v]) [X|R] :- !, close-links L R.
1884    close-links (v\[X v|L v]) [abs X|R] :- close-links L R.
1885    close-links (_\[]) [].
1886    type comp fm -> tm -> mmap -> mmap -> links -> links ->
1887      subst -> subst -> o.
1888    comp (fcon C) (con C) M M L L S S.
1889    comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
1890      maybe-eta (flam F) [], !,
1891        alloc S1 A S2,
1892        comp-lam F F1 M1 M2 L1 L2 S2 S3,
1893        get-scope (lam F1) Scope,
1894        L3 = [val (link-eta (uva A Scope) (lam F1)) | L2].
1895    comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :-               (c_λ)
1896      comp-lam F F1 M1 M2 L1 L2 S1 S2.
1897    comp (fuva A) (uva B []) M1 M2 L L S1 S2 :-
1898      m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
1899    comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
1900      pattern-fragment Ag, !,
1901        fold6 comp Ag Ag1 M1 M1 L L S1 S1,
1902        len Ag Arity,
1903        m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
1904    comp (fapp [fuva A|Ag]) (uva B Scope) M1 M3 L1 L3 S1 S4 :- !,
1905      pattern-fragment-prefix Ag Pf Extra,
1906      len Pf Arity,
1907      alloc S1 B S2,
1908      m-alloc (fv A) (hv C (arity Arity)) M1 M2 S2 S3,
1909      fold6 comp Pf    Pf1    M2 M2 L1 L1 S3 S3,
1910      fold6 comp Extra Extra1 M2 M3 L1 L2 S3 S4,
1911      Beta = app [uva C Pf1 | Extra1],
1912      get-scope Beta Scope,
1913      L3 = [val (link-beta (uva B Scope) Beta) | L2].
1914
```

```
1915    comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :-
1916      fold6 comp A A1 M1 M2 L1 L2 S1 S2.
1917
1918    type alloc mem A -> addr -> mem A -> o.
1919    alloc S N S1 :- mem.new S N S1.
1920
1921    type compile-terms-diagnostic
1922      triple diagnostic fm fm ->
1923      triple diagnostic tm tm ->
1924      mmap -> mmap ->
1925      links -> links ->
1926      subst -> subst -> o.
1927    compile-terms-diagnostic (triple D FO1 FO2) (triple D HO1 HO2) M1 M3 L1
1928      comp FO1 HO1 M1 M2 L1 L2 S1 S2,
1929      comp FO2 HO2 M2 M3 L2 L3 S2 S3.
1930
1931    type compile-terms
1932      list (triple diagnostic fm fm) ->
1933      list (triple diagnostic tm tm) ->
1934      mmap -> links -> subst -> o.
1935    compile-terms T H M L S :-
1936      fold6 compile-terms-diagnostic T H [] M_ [] L_ [] S_,
1937      print-compil-result T H L_ M_,
1938      deduplicate-map M_ M S_ S L_ L.
1939
1940    type make-eta-link-aux nat -> addr -> addr ->
1941      list tm -> links -> subst -> subst -> o.
1942    make-eta-link-aux z Ad1 Ad2 Scope1 L H1 H1  :-
1943      rev Scope1 Scope, eta-expand (uva Ad2 Scope) T1,
1944      L = [val (link-eta (uva Ad1 Scope) T1)].
1945    make-eta-link-aux (s N) Ad1 Ad2 Scope1 L H1 H3 :-
1946      rev Scope1 Scope, alloc H1 Ad H2,
1947      eta-expand (uva Ad Scope) T2,
1948      (pi x\ make-eta-link-aux N Ad Ad2 [x|Scope1] (L1 x) H2 H3),
1949      close-links L1 L2,
1950      L = [val (link-eta (uva Ad1 Scope) T2) | L2].
1951
1952    type make-eta-link nat -> nat -> addr -> addr ->
1953          list tm -> links -> subst -> subst -> o.
1954    make-eta-link (s N) z Ad1 Ad2 Vars L H H1 :-
1955      make-eta-link-aux N Ad2 Ad1 Vars L H H1.
1956    make-eta-link z (s N) Ad1 Ad2 Vars L H H1 :-
1957      make-eta-link-aux N Ad1 Ad2 Vars L H H1.
1958    make-eta-link (s N) (s M) Ad1 Ad2 Vars Links H H1 :-
1959      (pi x\ make-eta-link N M Ad1 Ad2 [x|Vars] (L x) H H1),
1960      close-links L Links.
1961
1962    type deduplicate-map mmap -> mmap ->
1963        subst -> subst -> links -> links -> o.
1964    deduplicate-map [] [] H H L L.
1965    deduplicate-map [(mapping (fv O) (hv M (arity LenM)) as X1) | Map1] Map2
1966      take-list Map1 (mapping (fv O) (hv M' (arity LenM'))) _, !,
1967      std.assert! (not (LenM = LenM')) "Deduplicate map, there is a bug",
1968      print "arity-fix links:" {ppmapping X1} "~!~" {ppmapping (mapping (fv
1969      make-eta-link LenM LenM' M M' [] New H1 H2,
1970      print "new eta link" {pplinks New},
1971      append New L1 L2,
1972
```

```
     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
  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 -> link
  progress-eta-link (ho.app _ as T) (ho.lam x\ _ as T1) H H1 [] :- !, not (T1 = ho.uva _ _), !, fail.
    ({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
    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 :-
```

```
2089    std.map2 S1 S2 (x\y\r\ r = ho.copy x y) Perm,
2090    Perm => ho.copy A A',
2091    (A' ==l B) H H1.
2092
2093  type progress1 links -> links -> ho.subst -> ho.subst -> o.
2094  progress1 [] [] X X.
2095  progress1 [A|L1] [A|L3] S S2 :- take-link A L1 B L2, !,
2096    same-link-eta A B S S1,
2097    progress1 L2 L3 S1 S2.
2098  progress1 [L0|L1] L3 S S2 :- deref-link S L0 L,
2099    solve-link-abs L R S S1, !,
2100    progress1 L1 L2 S1 S2, append R L2 L3.
```

16 THE DECOMPILER

```
2104  type abs->lam ho.assignment -> ho.tm -> o.
2105  abs->lam (ho.abs T) (ho.lam R)  :- !, pi x\ abs->lam (T x) (R x).
2106  abs->lam (ho.val A) A.
2107
2108  type commit-links-aux link -> ho.subst -> ho.subst -> o.
2109  commit-links-aux (@val-link-eta T1 T2) H1 H2 :-
2110    ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2111    (T1' ==l T2') H1 H2.
2112  commit-links-aux (@val-link-beta T1 T2) H1 H2 :-
2113    ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2114    (T1' ==l T2') H1 H2.
2115  commit-links-aux (ho.abs B) H H1 :-
2116    pi x\ commit-links-aux (B x) H H1.
2117
2118  type commit-links links -> links -> ho.subst -> ho.subst -> o.
2119  commit-links [] [] H H.
2120  commit-links [Abs | Links] L H H2 :-
2121    commit-links-aux Abs H H1, !, commit-links Links L H1 H2.
2122
2123  type decompl-subst map -> map -> ho.subst ->
2124    fo.fsubst -> fo.fsubst -> o.
2125  decompl-subst _ [A|_] _ _ _ :- fail.
2126  decompl-subst _ [] _ F F.
2127  decompl-subst Map [mapping (fv VO) (hv VM _)|Tl] H F F2 :-
2128    mem.set? VM H T, !,
2129    ho.deref-assmt H T TTT,
2130    abs->lam TTT T', tm->fm Map T' T1,
2131    fo.eta-contract T1 T2, mem.assign VO F T2 F1,
2132    decompl-subst Map Tl H F1 F2.
2133  decompl-subst Map [mapping _ (hv VM _)|Tl] H F F2 :-
2134    mem.unset? VM H, decompl-subst Map Tl H F F2.
2135
2136  type tm->fm map -> ho.tm -> fo.fm -> o.
2137  tm->fm _ (ho.con C)  (fo.fcon C).
2138  tm->fm L (ho.lam B1) (fo.flam B2) :-
2139    pi x y\ tm->fm _ x y => tm->fm L (B1 x) (B2 y).
2140  tm->fm L (ho.app L1) T :- map (tm->fm L) L1 [Hd|Tl],
2141    fo.mk-app Hd Tl T.
2142  tm->fm L (ho.uva VM TL) T :- mem L (mapping (fv VO) (hv VM _)),
2143    map (tm->fm L) TL T1, fo.mk-app (fo.fuva VO) T1 T.
2144
2145  type add-new-map-aux ho.subst -> list ho.tm -> map ->
```

```
2147    map ->  fo.fsubst -> fo.fsubst -> o.
2148  add-new-map-aux _ [] _ [] S S.
2149  add-new-map-aux H [T|Ts] L L2 S S2 :-
2150    add-new-map H T L L1 S S1,
2151    add-new-map-aux H Ts L1 L2 S1 S2.
2152
2153  type add-new-map ho.subst -> ho.tm -> map ->
2154    map ->  fo.fsubst -> fo.fsubst -> o.
2155  add-new-map _ (ho.uva N _) Map [] F1 F1 :-
2156    mem Map (mapping _ (hv N _)), !.
2157  add-new-map H (ho.uva N L) Map [Map1 | MapL] F1 F3 :-
2158    mem.new F1 M F2,
2159    len L Arity, Map1 = mapping (fv M) (hv N (arity Arity)),
2160    add-new-map H (ho.app L) [Map1 | Map] MapL F2 F3.
2161  add-new-map H (ho.lam B) Map NewMap F1 F2 :-
2162    pi x\ add-new-map H (B x) Map NewMap F1 F2.
2163  add-new-map H (ho.app L) Map NewMap F1 F3 :-
2164    add-new-map-aux H L Map NewMap F1 F3.
2165  add-new-map _ (ho.con _) _ [] F F :- !.
2166  add-new-map _ N _ [] F F :- name N.
2167
2168  type complete-mapping-under-ass ho.subst -> ho.assignment ->
2169    map -> map ->  fo.fsubst -> fo.fsubst -> o.
2170  complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
2171    add-new-map H Val Map1 Map2 F1 F2.
2172  complete-mapping-under-ass H (ho.abs Abs) Map1 Map2 F1 F2 :-
2173    pi x\ complete-mapping-under-ass H (Abs x) Map1 Map2 F1 F2.
2174
2175  type complete-mapping ho.subst -> ho.subst ->
2176    map -> map -> fo.fsubst -> fo.fsubst -> o.
2177  complete-mapping _ [] L L F F.
2178  complete-mapping H [none | Tl] L1 L2 F1 F2 :-
2179    complete-mapping H Tl L1 L2 F1 F2.
2180  complete-mapping H [some T0 | Tl] L1 L3 F1 F3 :-
2181    ho.deref-assmt H T0 T,
2182    complete-mapping-under-ass H T L1 L2 F1 F2,
2183    append L1 L2 LAll,
2184    complete-mapping H Tl LAll L3 F2 F3.
2185
2186  type decompile map -> links -> ho.subst ->
2187    fo.fsubst -> fo.fsubst -> o.
2188  decompile Map1 L HO FO FO2 :-
2189    commit-links L L1_ HO HO1, !,
2190    complete-mapping HO1 HO1 Map1 Map2 FO FO1,
2191    decompl-subst Map2 Map2 HO1 FO1 FO2.
```

17 AUXILIARY FUNCTIONS

```
2194  type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A ->
2195    list A1 -> B -> B -> C -> C -> o.
2196  fold4 _ [] [] A A B B.
2197  fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0,
2198    fold4 F XS YS A0 A1 B0 B1.
2199
2200  type len list A -> nat -> o.
2201  len [] z.
2202  len [_|L] (s X) :- len L X.
```