Higher-Order unification for free!

Reusing the meta-language unification for the object 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 proof system 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 [14], Twelf [16], λProlog [9] and Isabelle [22] which have been utilized to implement various formal systems such as First Order Logic [4], Set Theory [13], Higher Order Logic [12], and even the Calculus of Constructions [3].

The object logic we are interested in is Coq's [20] Calculus of Inductive Constructions (CIC), for which we aim to implement a unification procedure $\simeq_o$ using the ML Elpi [2], 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 [8]. We want $\simeq_o$ to be as powerful as $\simeq_\lambda$ but on the object logic CIC. Elpi also comes with an encoding for CIC that works well for meta-programming [19, 18, 6, 5]. 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 $\mathcal{F}_o$.

KEYWORDS

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

ACM Reference Format:
Davide Fissore and Enrico Tassi. 2024. Higher-Order unification for free!: Reusing the meta-language unification for the object language. In . ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Meta languages such as Elf [14], Twelf [16], λProlog [9] and Isabelle [22] have been utilized to specify various logics [4, 12, 13, 3]. The use of these meta languages facilitates this task in two key ways. The first and most well know one is that variable binding and substitution come for free. The second one is that these meta languages come equipped with some form of unification, a cornerstone of proof construction and proof search.

The object logic we are interested in is Coq's [20] Calculus of Inductive Constructions (CIC) and we want to implement a form of proof search known as type-class [21, 17] resolution. Type-class solvers are unification based proof search procedures reminiscent of Prolog that back-chain lemmas taken from a designated database of "type class instances". Given this analogy with Logic Programming we want to leverage the Elpi [19] meta programming language, a dialect of λProlog, already used to extend Coq in various ways [19, 18, 6, 5]. In this paper we focus on one aspect of this work, precisely *how to reuse the higher order unification procedure of the meta language in order to simulate a higher order logic program for the object language.*

We take as an example the `Decision` and `Finite` type classes from the Stdpp [7] library. The class `Decision` identifies predicates equipped with a decision procedure, while `Finite` the types whose inhabitants can be enumerated in a (finite) list. The following three designated type-class instances state that: 1) the type of natural numbers smaller than `n`, called `fin n`, is finite; 2) the predicate `nfact n nf`, relating a natural number `n` to the number of its prime factors `nf`, is decidable; 3) the universal closure of a predicate has a decision procedure if its domain is finite and if the predicate is decidable.

```
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).
```

Given this database a type-class solver is expected to prove the following statement automatically:

```
Decision (∀x: fin 7, nfact x 3)                    (* g *)
```

The proof found by the solver back-chains on rule 3 (the only rule about the ∀ quantifier), and then solves the premises with rules 1 and 2 respectively. Note that rule 3 features a second order parameter `P` that stands for a function of type `A → Prop` (a predicate over `A`). The solver has to infer a value for `P` by unifying the conclusion of rule 3 with the goal, and in particular it has to solve the unification problem `P x = nfact x 3`. This higher order problem falls in the so called pattern-fragment $\mathcal{L}_\lambda$ [8] and admits a unique solution σ that assigns the term λ`x.nfact x 3` to `P`.

In order to implement such a search in Elpi we shall describe the encoding of CIC terms and then the encoding of instances as rules (a.k.a. clauses). Elpi comes equipped with an Higher Order

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

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Abstract Syntax (HOAS [15]) datatype of CIC terms, called `tm`, that features (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 λProlog [9]'s standard syntax, the meta level binding of a variable `x` in an expression `e` is written «`x\ e`», and square brackets delimit a list of terms separated by comma. For example the term «`∀y:t, nfact y 3`» is encoded as follows:

```
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 and `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 intuitive encoding of rule $(r3)$ does not work, since it uses the predicate `P` as a first order term: for the meta language its type is `tm`. If we try to back-chain the rule $(r3)$ on the encoding of the goal (g) given below

```
decision (all (app [con"fin", con"7"]) y\        (g)
  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 CIC terms and rules that, on a first approximation, would reshape $(r3)$ as follows:

```
decision (all A x\ Pm x) :- link Pm P A, finite A,     (r3')
  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"]              (ρ)
```

Once the head of rule $(r3')$ unifies with the goal (g) the premise «`link Pm A P`» brings the assignment (ρ) back to the domain `tm` of Coq terms, obtaining the expected solution σ:

```
P = lam A x\ app [con"nfact", x, 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\`). We show below the premise before and after the instantiation of `P`:

```
decision (app [                P                   , 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 this example is a subtle mismatch between the equational theories of the meta language and the object language, that in turns makes the unification procedures of the meta language weak.
The equational theory of the meta language Elpi encompasses $\eta\beta$-equivalence and its unification procedure can solve higher order problems in the pattern fragment. Although the equational theory of CIC is much richer, for efficiency and predictability reasons automatic proof search procedure typically employ a unification procedure that only captures a $\eta\beta$-equivalence and only operates in $\mathcal{L}_\lambda$. The similarity is striking, but one needs some care in order to simulate a logic program in CIC using the unification of Elpi.

Contributions. In this paper we identify a minimal language $\mathcal{F}_o$ in which the problems sketched in the introduction can be formally described. We detail an encoding of a logic program on $\mathcal{F}_o$ to a strongly related logic program in $\mathcal{H}_o$ (the language of the meta language) and we show that the unification procedure of the meta language $\simeq_\lambda$ can be effectively used to simulate a unification procedure $\simeq_o$ for the object language that features $\eta\beta$-conversion in the pattern-fragment. fix

section 2 formally states the problem and gives the intuition behind our solution. section 9 discusses alternative term encodings and related works. section 3.1 introduces the languages $\mathcal{F}_o$ and $\mathcal{H}_o$, section 3 describes a basic simulation of higher order logic programs. sections 5 and 6 completes its equational theory with support for η-conversion. section 7 deals with the practical necessity of "tolerating" terms outside of the pattern-fragment and discusses how heuristic can be applied. Finally section 8 discusses the implementation in Elpi.

The λProlog code discussed in the paper can be accessed at the URL: https://github.com/FissoreD/paper-ho.

2 PROBLEM STATEMENT AND SOLUTION

Even if we encountered the problem working on CIC we devise a minimal setting to ease its study. In this setting we have a $\mathcal{F}_o$ language (for first order) with a rich(er) equational theory and a $\mathcal{H}_o$ meta language with a simpler one, and we reuse the unification procedure of $\mathcal{H}_o$ in order to implement one for $\mathcal{F}_o$.

2.1 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 as per fig. 1. Unification vari-

```
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

ables in $\mathcal{F}_o$ (fuva term constructor) 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 the memory address of P and x is a bound variable.
In $\mathcal{H}_o$ the representation of «P x» is instead «uva N [x]», since unification variables are higher order and come equipped with an explicit scope.

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 it is clear from the context we shall use the same syntax for $\mathcal{F}_o$ terms (although we never subscripts unification variables).
We use $s, s_1, \ldots$ for terms in $\mathcal{F}_o$ and $t, t_1 \ldots$ for terms in $\mathcal{H}_o$.

2.2 Equational theories an unification

In order to specify unification we need to define the equational theory and substitution (unification-variable assignment).

2.2.1 *Term equality: $=_o$ and $=_\lambda$.*

For both languages we extend the equational theory over ground terms to the full language by adding the reflexivity of unification variables (a variable is equal to itself).
The first four rules are common to both equalities and define the usual congruence over terms. 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 :-                                               (η_l)
  pi x\ beta T [x] (T' x), x =_o x => F x =_o T' x.
T =_o flam F :-                                               (η_r)
  pi x\ beta T [x] (T' x), x =_o x => T' x =_o F x.
fapp [flam X | L] =_o T :- beta (flam X) L R, R =_o T.      (β_l)
T =_o fapp [flam X | L] :- beta (flam X) L R, T =_o R.      (β_r)
```

```
type (=_λ) tm -> tm -> o.
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 brevity we omit the code of beta: it is sufficient to know that «beta F L R» computes in R the weak head normal form of «app[F|L]».

Substitution: ρs and σt. We write $\sigma = \{ X \mapsto t \}$ for the substitution that assigns the term t to the variable X. We write σt for the application of the substitution to a term t, and $\sigma X = \{ \sigma t \mid t \in X \}$ when X is a set of terms. We write $\sigma \subseteq \sigma'$ when σ is more general than σ'. The domain of a substitution is the set of unification variables for which it provides an assignment. We write $\sigma \cup \sigma'$ to denote the concatenation of two substitutions whose domains are disjoint. We shall use ρ for $\mathcal{F}_o$ substitutions, and σ for the $\mathcal{H}_o$ ones. For brevity, in this section we consider the substitution for $\mathcal{F}_o$ and $\mathcal{H}_o$ identical. We defer to section 3.1 a more precise description pointing out theirs differences.

Term unification: $\simeq_o$ vs. $\simeq_\lambda$. Although we provide an implementation of the meta-language unification $\simeq_\lambda$ in the supplementary material (that we used for testing purposes) we only describe its signature here.

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

We write $\sigma t_1 \simeq_\lambda \sigma t_2 \mapsto \sigma'$ when σt_1 and σt_2 unify with substitution σ'. We write $t_1 \simeq_\lambda t_2 \mapsto \sigma$ when the initial substitution is empty. Note that if $\sigma t_1 \simeq_\lambda \sigma t_2 \mapsto \sigma'$ then the domains of σ and σ' are disjoint.
The meta language of choice is expected to provide an implementation of $\simeq_\lambda$ that satisfies the following properties:

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

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

meglio dire che sigma' estende sigma, togliere unione e quindi dominio da sopra

Even if we provide an implementation of the object-language unification $\simeq_o$ in section 3.7, our real goal is the simulation of an entire logic program.

2.3 The problem: logic-program simulation

We represent a logic program *run* in $\mathcal{F}_o$ as a list *steps* of length $\mathcal{N}$. At each step p we unify two terms $\mathbb{P}_{p_l}$ and $\mathbb{P}_{p_r}$ taken from the set of all terms $\mathbb{P}$. [1] The composition of these steps starting from the empty substitution ρ_0 produces the final substitution $\rho_\mathcal{N}$, that is the result of the logic-program execution.

spiegare llam (first use after intro)

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

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

In order to simulate a $\mathcal{F}_o$ logic program in $\mathcal{H}_o$ we compile each $\mathcal{F}_o$ term in $\mathbb{P}$ into a $\mathcal{H}_o$ term t. We write this translation $\langle s \rangle \mapsto (t, m, l)$. The implementation of the compiler is detailed in sections 3, 5 and 7, here we just point out that it additionally produce a variable mapping m and list of links l. The variable map connects unification variables in $\mathcal{H}_o$ to variables in $\mathcal{F}_o$ and is used to "decompile" the assignment, $\langle \sigma, m, l \rangle^{-1} \mapsto \rho$. Links are an accessory piece of information whose description is deferred to section 2.4.
We simulate each run in $\mathcal{F}_o$ with a run in $\mathcal{H}_o$ as follows.

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

dire che P e T sono liste di coppie, dire che P e T vengono anche usati

[1] If the same rule is used multiple times in a run we just consider as many copies as needed of the terms composing the rules, with fresh unification variables each time.

$$
\begin{aligned}
&\mathrm{hrun}(\mathbb{P}, \mathcal{N}) \mapsto \rho_{\mathcal{N}} \stackrel{def}{=\!=}\\
&\quad \mathbb{T} \times \mathbb{M} \times \mathbb{L}_0 = \{(t, m, l) | s \in \mathbb{P}, \langle s \rangle \mapsto (t, m, l)\}\\
&\quad \textstyle\bigwedge_{p=1}^{\mathcal{N}} \mathrm{hstep}(\mathbb{T}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p)\\
&\quad \langle \sigma_{\mathcal{N}}, \mathbb{M}, \mathbb{L}_{\mathcal{N}} \rangle^{-1} \mapsto \rho_{\mathcal{N}}
\end{aligned}
$$

By analogy with $\mathbb{P}$, we write $\mathbb{T}_{p_l}$ and $\mathbb{T}_{p_r}$ for the two $\mathcal{H}_o$ terms being unified at step p, and we write $\mathbb{T}_p$ for the set $\{\mathbb{T}_{p_l}, \mathbb{T}_{p_r}\}$.
hstep is made of two sub-steps: a call to the meta language unification and a check for progress on the set of links, that intuitively will compensate for the weaker equational theory honoured by $\simeq_\lambda$.
hrun compiles all terms in $\mathbb{P}$, then executes each step and finally decompiles the solution. We claim:

Proposition 2.1 (Simulation). *$\forall \mathbb{P}, \forall \mathcal{N}$, if $\mathbb{P} \subseteq \mathcal{L}_\lambda$*

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

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

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

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

In particular this property guarantees that a *failure* in the $\mathcal{F}_o$ run is matched by a failure in $\mathcal{H}_o$ *at the same step*. We consider this property very important from a practical point of view since it guarantees that the execution traces are strongly related and in turn this enables a user to debug a logic program in $\mathcal{F}_o$ by looking at its execution trace in $\mathcal{H}_o$.

We also claim that hrun handles terms outside $\mathcal{L}_\lambda$ in the following sense:

Proposition 2.3 (Fidelity recovery). *In the context of* hrun, *if $\rho_{p-1}\mathbb{P}_p \in \mathcal{L}_\lambda$ (even if $\mathbb{P}_p \notin \mathcal{L}_\lambda$) then*

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

In other words if the two terms involved in a step re-enter $\mathcal{L}_\lambda$, then hstep and fstep are again related, even if $\mathbb{P} \notin \mathcal{L}_\lambda$ and hence proposition 2.2 does not apply.

This property has a practical relevance since in many logic programming implementations, including Elpi, the order in which unification problems are tackled does matter. The simplest example is the sequence $F \simeq \lambda x.a$ and $F \cdot a \simeq a$: the second problem is not in $\mathcal{L}_\lambda$ and has two unifiers, namely $\sigma_1 = \{ F \mapsto \lambda x.x \}$ and $\sigma_2 = \{ F \mapsto \lambda x.a \}$. The first problem picks σ_2 making the second problem re-enter $\mathcal{L}_\lambda$.

say that this is run-time, while the other fidelity is compile time

2.4 The solution (in a nutshell)

A term s is compiled to a term t where every "problematic" sub term p is replaced by a fresh unification variable h with an accessory *link* that represents a suspended unification problem $h \simeq_\lambda p$. As a result $\simeq_\lambda$ is "well behaved" on t, in the sense that it does not contradict $=_o$ as it would otherwise do on the "problematic" sub-terms.
We now define "problematic" and "well behaved" more formally and we pick the $\Diamond$ symbol since it stands for "possibly" in modal logic and all problematic terms are characterized by some "uncertainty".

Definition 2.4 ($\Diamond\beta_0$). $\Diamond\beta_0$ is the set of terms of the form $X \cdot x_1 \dots x_n$ such that $x_1 \dots x_n$ are distinct names (of bound variables).

An example of term $\Diamond\beta_0$ is the application $F \cdot x$. This term is problematic since the application node of its syntax tree cannot be used to justify a unification failure, i.e. by properly instantiating F the term head constructor may become a λ, or a constant or stay an application.

Definition 2.5 ($\Diamond\eta$). $\Diamond\eta$ is the set of terms s such that $\exists\rho, \rho s$ is an eta expansion.

An example of term s in $\Diamond\eta$ is $\lambda x.\lambda y.F \cdot y \cdot x$ since the substitution $\rho = \{F \mapsto \lambda a.\lambda b.f \cdot b \cdot a\}$ makes $\rho s = \lambda x.\lambda y.f \cdot x \cdot y$ that is the eta long form of f. This term is problematic since its leading λ abstraction cannot justify a unification failure against a constant f.

Definition 2.6 ($\Diamond\mathcal{L}_\lambda$). $\Diamond\mathcal{L}_\lambda$ is the set of terms of the form $X \cdot t_1 \dots t_n$ such that $t_1 \dots t_n$ are not distinct names.

These terms are problematic for the very same reason terms in $\Diamond\beta_0$ are, but cannot be handled directly by the unification of the meta language, that is only required to handle terms in $\mathcal{L}_\lambda$. Still, there exists a substitution ρ such that $\rho s \in \mathcal{L}_\lambda$.

We write $\mathcal{P}(t)$ the set of sub-terms of t, and we write $\mathcal{P}(X) = \bigcup_{t \in X} \mathcal{P}(t)$ when X is a set of terms.

Definition 2.7 (Well behaved set). Given a set of terms $X \subseteq \mathcal{H}_o$,

$$\mathcal{W}(X) \Leftrightarrow \forall t \in \mathcal{P}(X), t \notin (\Diamond\beta_0 \ \cup \ \Diamond\eta \ \cup \ \Diamond\mathcal{L}_\lambda)$$

We write $\mathcal{W}(t)$ as a short for $\mathcal{W}(\{t\})$. We claim our compiler validates the following property:

Proposition 2.8 ($\mathcal{W}$-enforcing). *Given two terms s_1 and s_2, if $\exists\rho, \rho s_1 =_o \rho s_2$, then*

$$\langle s_i \rangle \mapsto (t_i, m_i, l_i) \text{ for } i \in \{1, 2\} \Rightarrow t_1 \simeq_\lambda t_2 \mapsto \sigma$$

Note that the property holds for any substitution, it could be given by an oracle and not necessarily a most general one, and for any terms, in particular terms in $\Diamond\beta_0 \ \cup \ \Diamond\eta \ \cup \ \Diamond\mathcal{L}_\lambda$.

dire perchè W(T), dire che il converse non vale (alcuni check li fa la progress)

Proposition 2.9 ($\mathcal{W}$-preservation). *$\forall \mathbb{T}, \forall \mathbb{L}, \forall p, \forall \sigma, \forall \sigma'$*

$$
\begin{aligned}
&\mathcal{W}(\sigma\mathbb{T}) \land \sigma\mathbb{T}_{p_l} \simeq_\lambda \sigma\mathbb{T}_{p_r} \mapsto \sigma' \Rightarrow \mathcal{W}(\sigma'\mathbb{T})\\
&\mathcal{W}(\sigma\mathbb{T}) \land \mathrm{progress}(\mathbb{L}, \sigma) \mapsto (\_, \sigma') \Rightarrow \mathcal{W}(\sigma'\mathbb{T})
\end{aligned}
$$

Proposition 2.9 is key to prove propositions 2.1 and 2.2: informally it says that the problematic terms moved on the side by the compiler are not put back by hstep, hence $\simeq_\lambda$ can continue to operate properly. In sections 3, 5 and 7 we describe how the compiler recognizes terms in $\Diamond\beta_0$, $\Diamond\eta$ and $\Diamond\mathcal{L}_\lambda$ and how progress takes care of them preserving $\mathcal{W}$ and granting propositions 2.1 to 2.3.

3 BASIC COMPILATION AND SIMULATION

3.1 Memory map ($\mathbb{M}$) and substitution (ρ and σ)

Unification variables are identified by a natural number that represents a memory addresses. The memory and its associated operations are described below:

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

type set? addr -> mem A -> A -> o.
type unset? addr -> mem A -> o.
```

```
type assign addr -> mem A -> A -> mem A -> o.
type new mem A -> addr -> mem A -> o.
```

If a memory cell is none, then the corresponding unification variable is not set. assign sets an unset cell to the given value, while new finds the first unused address and sets it to none.

Since each occurrence of a $\mathcal{H}_o$ unification variables has a scope, its assignment needs to be abstracted over it to enable the instantiation of the same assignment to different scopes. This is expressed by the inctx container, and in particular 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 $\mathcal{H}_o$.

The compiler establishes a mapping between variables of the two languages.

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

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

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

Each hvariable is stored in the mapping together with its arity so that the code of (*malloc*) below can preserve:

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* L *has length* N*.*

```
type m-alloc fvariable -> hvariable -> mmap -> mmap ->
  subst -> subst -> o.                                     (malloc)
m-alloc Fv Hv M M S S :- mem M (Fv <-> Hv), !.
m-alloc Fv Hv M [Fv <-> Hv|M] S S1 :- Hv = hv N _, new 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 η-link; this detail is discussed in section 6.

Applying the substitution corresponds to dereferencing a term with respect to the memory. It is worth looking at the code fpr $\mathcal{H}_o$ to remark how assignments are 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 are the same. Hence they have the same type in the meta-level and the number of abs nodes in the assignment matches that length. In turn this grants that move never fails.

convoluted

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

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 ».

3.2 Links ($\mathbb{L}$)

As we mentioned in section 2.4 the compiler replaces terms in $\Diamond\eta$, $\Diamond\beta_0$ and $\Diamond\mathcal{L}_\lambda$ with fresh variables linked to the problematic terms. Terms in $\Diamond\beta_0$ do not need a link since $\mathcal{H}_o$ variables faithfully represent the problematic term thanks to their scope.

```
kind baselink type.
type link-eta  tm -> tm -> baselink.
type link-llam 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 rule $\cdot \vdash \cdot$).

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 5 and section 7.

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 $x \vdash A_x =_{\mathcal{L}_\lambda} F_x\ a$ corresponds to:

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

3.3 Notational conventions

move away

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

add notation for problem compilation

3.4 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 5 and section 7. With respect to section 2 the signature also allows for updates to the substitution.

```
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 A) (app A1) M1 M2 L1 L2 S1 S2 :-          (c@)
  fold6 comp A A1 M1 M2 L1 L2 S1 S2.
```

The code above 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 sections).

explain fold6

```
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 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.

3.5 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).
```

3.5.1 Progress. In the base compilation scheme `progress1` is the identity on both the links and the substitution, so the fixpoint trivially terminates. Sections 5 and 7 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 $\mathbb{L}$.

3.5.2 Occur check. Since compilation moves problematic terms out of the sigh of $\simeq_\lambda$, that procedure can only perform a partial occur check. For example the unification problem $X \simeq_\lambda f\ Y$ 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).

3.6 Substitution decompilation

Decompiling the substitution involves three steps.

First and foremost problematic terms stored in $\mathbb{L}$ have to be moved back into the game. Since links are of the form uvar = term (invariant 2 (Link left hand side)) and are duplicate free (see dedup-beta dedup-eta), one can turn a link X = t into an assignment $X \mapsto t$. This can in general be achieved by unifying X with t. The case where t is not in $\mathcal{L}_\lambda$ (link beta/llam) is discussed in section xx.

The second step amounts at allocating new variables in the memory of $\mathcal{F}_o$. In particular some unif problems such as $Fxy = Fxz$ requires to allocate a variable G so that the assignment $F_{ab} \mapsto G_a$ can be used to perform required pruning.

The last step amounts at decompiling each assignment. Decompiling a term is trivial. An assignment has an abs node, as in move, can be eliminated by replacing the bound variable by the actual term in scope. In order to do this, one needs the $\mathbb{M}$ to be a bijection. This is the job of section 6.

polish

dire che però si passa per una subst in cui ste abs le cambio in lam. Nel codice Coq ci scrivevamo il tipo nella arity, e quindi sappiamo fare i lambda bene, senza perdita di informazione. Qui i lam non hanno info, facile. Ma in generale bisogna spiegare come ci si salva. Ci dormo su: o non generiamo la subst ma solo il primo termine (la query iniziale) istanziato (funziona sempre, la prova è quella sopra) oppure bisogna siegare tutto sto casino e serve un po' di spazio.

3.7 Definition of $\simeq_o$ 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 still makes no use of the higher order nature of the ML unif language, indeed the scope of unif variables generated by the compiler is always empty, so $\simeq_\lambda$ is first order.

Still, if $\mathbb{P}$ is already $\mathcal{W}$, we can set up a proof that will also work when comp enforces $\mathcal{W}$ and hstep preserves it, and when terms in $\mathcal{L}_\lambda$ are mapped to ho variables with a scope.

Lemma 3.1 (Compilation round trip). *If* `comp S T [] M [] _ [] _` *then* `decomp M T S`

Proof sketch. trivial if the mapping is a bijection and the terms are beta normal. some discussion about commit maybellam to be done later. □

Lemma 3.2. *Properties* (1) *and* (2) *hold for the implementation of* $\simeq_o$ *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 is only one 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 $\rho = \{Y \mapsto s\}$.

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

Theorem 3.3 (Fidelity in $\mathcal{W}$). *Proposition 2.1 (Simulation) and proposition 2.2 (Simulation fidelity) 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 all is $\mathcal{W}$, $\simeq_\lambda$ is equivalent to $\simeq_o$. □

4 HANDLING OF $\Diamond\beta_0$

Detection. trivial, pattern-fragment.

4.1 Compilation and decompilation

The following rule is inserted just before rule $(c_@)$.

```
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.
```

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. Also note that this rule generates no links.

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

Decompilation. since no link, nothing new

Progress. since ML unif is complete here, no need to move terms aside, just use uva is enough.

Lemma 4.1. *Properties* (1) *and* (2) *hold for the implementation of* $\simeq_o$ *in section 3.7*

Proof sketch. If we look at the $\mathcal{F}_o$ terms, the is one more case interesting cases:

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

ste x es- cono da L

Lemma 4.2 ($\mathcal{W}$-enforcement). *Even if* $\mathbb{P} \cap \Diamond\beta_0 \neq \emptyset, \mathbb{T} \cup \Diamond\beta_0 = \emptyset$

Proof sketch. problematic terms are mapped to uva by comp, the problematic fapp node is gone. □

Theorem 4.3 (Fidelity in $\Diamond\beta_0$). *Proposition 2.1 (Simulation) and proposition 2.2 (Simulation fidelity) hold*

Proof sketch. thanks to lemma 4.2 it is the same as in section 3, even if now we really need $\simeq_\lambda$ to deal with $\mathcal{L}_\lambda$, while before a FO unif would have done. □

5 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 3.4 the unification problem $\mathbb{P}$ is compiled as follows:

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

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 5.1) 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 5.2). The compilation of the problem $\mathbb{P}$ above is refined to:

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

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

Invariant 3 (η-`link` rhs). *The* rhs *of any* η-`link` *has the shape* $\lambda x.t$ *and* t *is not a lambda.*

η-`link` 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 3.5).

5.1 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:

$$
\begin{array}{lll}
\lambda x.f\cdot(A\cdot x) & \in \Diamond\eta & \rho = \{\, A \mapsto \lambda x.x \,\} \\
\lambda x.f\cdot(A\cdot x)\cdot x & \in \Diamond\eta & \rho = \{\, A \mapsto \lambda x.a \,\} \\
\lambda x.f\cdot x\cdot(A\cdot x) & \notin \Diamond\eta & \\
\lambda x.\lambda y.f\cdot(A\cdot x)\cdot(B\cdot y\cdot x) & \in \Diamond\eta & \rho = \{\, A \mapsto \lambda x.x,\ B \mapsto \lambda y.\lambda x.y \,\}
\end{array}
$$

The first two examples are easy, and show how a unification variable can expose or erase a variable in their scope 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:

Definition 5.1 (may-contract-to). A β-normal term s *may-contract-to* a name x if there exists a substitution ρ such that $\rho s =_o x$.

Lemma 5.2. *A β-normal term $s = \lambda x_1 \dots x_n.t$* 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. $\lambda x_1 \dots x_n.x\cdot x_1 \dots x_n =_o x$);*
(3) t is a unification variable with scope W, and for any $v \in \{x, x_1 \dots x_n\}$, 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 is under the spine of binders $x_1 \dots x_n$, t can either be x 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). ◻

Definition 5.3 (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 a unification variable X, otherwise an instantiation of X can make x disappears from t. Moreover, note that η-contracting t cannot make x disappear, since x is not a locally bound variable inside t.

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

Definition 5.4 (maybe-eta). Given a β-normal term $s = \lambda x_1 \dots x_n.t$, *maybe-eta* s holds if any of the following holds:

(1) t is a constant or a name applied to the arguments $l_1 \dots l_m$ such that $m \geq n$ and for every i such that $m - n < i \leq m$ the term l_i *may-contract-to* x_i, and no x_i *occurs-rigidly* in $l_1 \dots l_{m-n}$;
(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 5.5 ($\Diamond\eta$ detection). *If t is a β-normal term and* maybe-eta t *holds, then $t \in \Diamond\eta$.*

Proof sketch. Follows from definition 5.3 and lemma 5.2 ◻

Remark that the converse of lemma 5.5 does not hold: there exists a term t satisfying the criteria (1) of definition 5.4 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.

5.2 Compilation and decompilation

Compilation. The following rule is inserted just before rule (c_λ) from the code in section 3.4.

```
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 `flam F` to `lam F1` but puts the fresh variable `A` in its place. This variable sees all the names free in `lam F1`. The critical part of this rule is the creation of the η-`link`, which relates the variable `A` with `lam F1`. This link clearly validates invariant 2.

Corollary 5.6. *The* rhs *of any η-`link` has exactly one lambda abstraction, hence the rule above respects invariant 3.*

Proof sketch. By contradiction, suppose that the rule above is 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. ◻

Decompilation. Decompilation of the remaining η-`link` (i.e. the η-`link` that have been activated) is performed by iterating over them and unifying lhs and rhs. Note that this unification never fails, since lhs is a flexible term not appearing in any other η-`link` (by definition 5.9).

5.3 Progress

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

Definition 5.7 (progress-η-left). A link $\Gamma \vdash X =_\eta T$ is removed from $\mathbb{L}$ when X becomes rigid. Let $y \in \Gamma$, 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 X with T, that is we run $\lambda x.X \cdot x \simeq_\lambda T$
(2) if $X = \lambda x.t$ we run $X \simeq_\lambda T$.

Definition 5.8 (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 η-`link`.

Definition 5.9 (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).

Lemma 5.10. *Let $\lambda x.t$ the* rhs *of a η-`link`, then $\mathcal{W}(t)$.*

Proof sketch. By construction, every "problematic" term in $\mathcal{F}_o$ is replaced with a variable in the corresponding $\mathcal{H}_o$ term. Therefore, t is $\mathcal{W}$. □

Lemma 5.11. *Given a η-`link` l, the unification done by* progress-η-left *is between terms in $\mathcal{W}$*

Proof sketch. Let σ be the substitution, which is $\mathcal{W}(\sigma)$ (by proposition 2.9). lhs $\in \sigma$, therefore $\mathcal{W}$(lhs). By *progress-η-left*, if 1) lhs is a name, a constant or an application, then, $\lambda x.\text{lhs} \cdot x$ is unified with rhs. By invariant 3 and lemma 5.10, rhs $= \lambda x.t$ and $\mathcal{W}(t)$. Otherwise, 2) lhs has `lam` as functor. In both cases, unification is performed between terms in $\mathcal{W}$. □

Lemma 5.12. *Given a η-`link` l, the unification done by* progress-η-right *is between terms in $\mathcal{W}$.*

Proof sketch. lhs is variable, and, by definition 5.8, rhs is either no more a $\Diamond\eta$, i.e. rhs is not a η-expansion and, so, $\mathcal{W}$(rhs), otherwise, rhs can reduce to a term which cannot be a η-expansion, and, so, $\mathcal{W}$(rhs). In both cases, the unification between rhs and lhs is done between terms that are in $\mathcal{W}$. □

Lemma 5.13. *Given a η-`link` l, the unification done by* progress-η-deduplicate *is between terms in $\mathcal{W}$.*

Proof. The unification is done between the rhs of two η-`link`. Both rhs has the shape $\lambda x.t$, and by lemma 5.10, $\mathcal{W}(t)$. Therefore, the unification is done between well-behaved terms. □

Lemma 5.14. *The introduction of η-`link` guarantees proposition 2.9 ($\mathcal{W}$-preservation)*

is this W-enforcing?

Proof sketch. By lemmas 5.11 to 5.13, every unification performed by the activation of a η-`link` is done between terms in $\mathcal{W}$, therefore, the substitution remains $\mathcal{W}$. □

Lemma 5.15. `progress` *terminates.*

Proof sketch. Rules definitions 5.7 and 5.8 and definition 5.9 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). □

Theorem 5.16 (Fidelity in $\Diamond\eta$). *Given a list of unification problems $\mathbb{P}$, such that $\forall t, t \in \mathcal{P}(\mathbb{P}) \wedge t \notin \Diamond\mathcal{L}_\lambda$, the introduction of η-`link` guarantees proposition 2.2 (Simulation fidelity).* [2]

Proof sketch. *progress-η-left* and *progress-η-deduplicate* activate a η-`link` when, in the original unification problem, a $\Diamond\eta$ term is unified with respectively a well-behaved term or another $\Diamond\eta$ term. In both cases, the links trigger a unification which succeeds iff the same unification in $\mathcal{F}_o$ succeeds, guaranteeing proposition 2.2. *progress-η-right* never fails, in fact, this progression refines a variable to a rigid term and plays no role in proposition 2.2. □

Example of progress-η-left. The example at the beginning of section 5, once $\sigma = \{\, A \mapsto f \,\}$, triggers *progress-$\eta$-left* 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 η-`link` deduplication, is given below:

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

The result of $A \simeq_\lambda C$ is that the two η-`link` 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 η-`link` progression due to *progress-η-right*

6 MAKING $\mathbb{M}$ A BIJECTION

In section 3.1, we introduced the definition of "memory map" ($\mathbb{M}$). This memory allows to decompile the $\mathcal{H}_o$ terms back to the object language. It is the case that, while solving unification problems, a same unification variable X is used multiple times with different arities.

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

In the unification problems $\mathbb{P}$ above, we see that X is used with arity 2 in $\mathbb{P}_1$ and with arity 1 in $\mathbb{P}_2$. By invariant 1 (Unification-variable arity), we are not allowed to use a same $\mathcal{H}_o$ variable to represent the two occurrences of X. If we execute hrun, we remark that the unification fails. There is in fact a major problem: hstep is not conscious of the connection between the variables C and

[2] We also suppose that any higher-order variable is always applied with the same number of arguments. This problem is addressed in section 6

E (both corresponding to X), since no link in $\mathbb{L}$ puts C and E in relation and decompilation does not work properly if a $\mathcal{F}_o$ variable is mapped to two distinct $\mathcal{H}_o$ variables. The two main drawbacks connected to this situation are firstly the lost of proposition 2.2 (Simulation fidelity) and secondly, if we want to guarantee at least proposition 2.1 (Simulation), we should overcomplicate the decompilation phase. In order to ease the second drawback, we pose the following property:

Proposition 6.1 ($\mathbb{M}$ is a bijection). *Given a list of unification problems $\mathbb{P}$, then the memory map $\mathbb{M}$ compiled from $\mathbb{P}$ is a bijection relating the $\mathcal{F}_o$ and the $\mathcal{H}_o$ variables.*

We finally adjust the compiler's output with a `map-deduplication` procedure.

*Definition 6.2 (*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 η-`link` can add exactly one lambda, we need as many links as the difference between the two arities.

*Definition 6.3 (*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.

Theorem 6.4 (Fidelity with *map-deduplication*). *Given a list of unification problems $\mathbb{P}$, such that $\forall t, t \in \mathcal{P}(\mathbb{P}) \Rightarrow \mathcal{W}(t) \vee t \in \Diamond\eta$, if $\mathbb{P}$ contains two same $\mathcal{F}_o$ variables with different arities, then* map-deduplication *guarantees proposition 2.2 (Simulation fidelity)*

Proof sketch. By the definition of *map-deduplication*, any two same $\mathcal{F}_o$ variables X_1, X_2 with different arities are related with η-`link`. If one of the two variables is instantiated, the corresponding η-`link` is triggered instantiating the related variable. This allows to make unification fail if X_1 and X_2 are unified with different terms. Finally, since $\mathbb{P}$ contains only terms thar are either $\mathcal{W}$ or $\Diamond\eta$, by theorem 5.16, we can conclude the proof. □

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 η-`link`: $x \vdash E_x =_\eta \lambda y.C_{xy}$. After deduplication the compiler output is as follows:

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

In this example, $\mathbb{T}_1$ assigns A which triggers $\mathbb{L}_3$ and then $\mathbb{L}_4$ by *progress-η-left*. 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 $\lambda y.y$ (the term C_{xy} reduces to y). Since it is no more in $\Diamond\eta$, $\lambda y.y$ is unified with E_x. After the execution of the remaining hstep, we obtain the following $\mathcal{F}_o$ substitution $\rho = \{X := \lambda x.\lambda y.y, Y := (f \cdot \lambda x.x)\}$.

7 HANDLING OF $\Diamond\mathcal{L}_\lambda$

In this section we suppose the unification of the object language between two terms t_1 and t_2 to fail each time at least one of the between t_1 or t_2 is outside $\mathcal{L}_\lambda$. This means for instance that $X \not\simeq_o Y \cdot Z$ and $X \cdot Y \not\simeq_o X \cdot Y$.

In general, unification between $\Diamond\mathcal{L}_\lambda$ terms admits more then one solution and committing one of them in the substitution does not guarantee property (2). For instance, $X \cdot a \simeq_o a$ 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.

Given a list of unification problems, $\mathbb{P}_1 \ldots \mathbb{P}_n$ with $\mathbb{P}_n$ in $\Diamond\mathcal{L}_\lambda$, it is often the case that 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 $\mathcal{L}_\lambda$.

$$\begin{array}{l}
\mathbb{P} = \{\ X \simeq_o \lambda x.a \quad (X \cdot a) \simeq_o a\ \} \\
\mathbb{T} = \{\ A \simeq_\lambda \lambda x.a \quad (A \cdot a) \simeq_\lambda a\ \} \\
\mathbb{M} = \{\ X \mapsto A^0\ \}
\end{array}$$

In the example above, we see that $\mathbb{P}_1$ instantiates X so that $\mathbb{P}_2$ can be solved in $\mathcal{L}_\lambda$. 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 $\sigma = \{A \mapsto \lambda x.a\}$, but the dereferencing of $\mathbb{T}_2$ gives the non-unifiable problem $(\lambda x.a) \cdot a \not\simeq_\lambda a$.

To address this unification problem, term compilation must recognize and replace $\Diamond\mathcal{L}_\lambda$ terms with fresh variables. This replacement produces links that we call $\mathcal{L}_\lambda$-`link`.

$\mathcal{L}_\lambda$-`link` respects invariant 2 and the term on the rhs has the following property:

Invariant 4 ($\mathcal{L}_\lambda$-`link` rhs). *The* rhs *of any $\mathcal{L}_\lambda$-`link` has the shape $X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$ such that X is a unification variable with scope $s_1 \ldots s_n$[3] 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` $= t_1 \ldots t_m$*.*

7.1 Compilation and decompilation

Detection of $\Diamond\mathcal{L}_\lambda$ is quite simple to implement in the compiler, since it is sufficient to detect applications with flexible head and argument that are not in $\mathcal{L}_\lambda$. The following rule for $\Diamond\mathcal{L}_\lambda$ compilation is inserted just before rule ($c_@$).

```
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-llam (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 $\mathcal{L}_\lambda$-`link` is the application of a fresh variable `C` having in scope all the free variables appearing in the compiled version of `Pf` and

[3] with $s_1 \ldots s_n$ that are distinct names

`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`. Note that this construction enforce invariant 4.

D:Si puo togliere se serve spazio

Corollary 7.1. *Let $X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$ be the* rhs *of a* $\mathcal{L}_\lambda$`-link`, *then* $m > 0$.

Proof sketch. Assume we have a $\mathcal{L}_\lambda$`-link`, 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 3.4) and no $\mathcal{L}_\lambda$`-link` is produced which contradicts our initial assumption. ◻

D:Si puo togliere se serve spazio

Corollary 7.2. *Let $X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$ be the* rhs *of a* $\mathcal{L}_\lambda$`-link`, *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. ◻

Decompilation. A failure is thrown if any $\mathcal{L}_\lambda$`-link` remains in $\mathbb{L}$ at the begin of decompilation, i.e. all $\mathcal{L}_\lambda$`-link` should be solved before decompilation.

7.2 Progress

Given a $\mathcal{L}_\lambda$`-link` l of the form $\Gamma \vdash T =_{\mathcal{L}_\lambda} X_{s_1 \ldots s_n} \cdot t_1 \ldots t_m$, we provide 4 different activation rules:

Definition 7.3 (progress-$\mathcal{L}_\lambda$-refine). Given a substitution σ, where σt_1 is a name, say t, and $t \notin s_1 \ldots s_n$. If $m = 0$, then l is removed and lhs is unified with $X_{s_1 \ldots s_n}$. If $m > 0$, then l is replaced by a refined version $\Gamma \vdash T =_{\mathcal{L}_\lambda} Y_{s_1 \ldots s_n, t} \cdot t_2 \ldots t_m$ with reduced list of arguments and Y being a fresh variable. Moreover, the new link $\Gamma \vdash X_{s_1 \ldots s_n} =_\eta \lambda x. Y_{s_1 \ldots s_n, x}$ is added to $\mathbb{L}$.

Definition 7.4 (progress-$\mathcal{L}_\lambda$-rhs). l 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 with t.

Definition 7.5 (progress-$\mathcal{L}_\lambda$-fail). If it exists a link $l' \in \mathbb{L}$ with same lhs as l, or the lhs of l become rigid, then unification fail.

Lemma 7.6. `progress` *terminates*

Proof sketch. Let l a $\mathcal{L}_\lambda$`-link` in the store $\mathbb{L}$. If l is activated by *progress-$\mathcal{L}_\lambda$-rhs*, 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-$\mathcal{L}_\lambda$-refine*, l is replaced by a new $\mathcal{L}_\lambda$`-link` l^1 having $m - 1$ arguments. At the m^{th} iteration, the $\mathcal{L}_\lambda$`-link` l^m has no more arguments and is removed from $\mathbb{L}$. Note that at the m^{th} iteration, m new η`-link` have been added to $\mathbb{L}$, however, by lemma 5.15, the algorithm terminates. Finally *progress-$\mathcal{L}_\lambda$-fail* also guarantees termination since it makes `progress` immediately fails.

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).

◻

Theorem 7.7 (Fidelity with $\mathcal{L}_\lambda$`-link`). *The introduction of* $\mathcal{L}_\lambda$`-link` *guarantees proposition 2.2 (Simulation fidelity)*

Proof sketch. Let $\mathbb{T}$ a unification problem and σ a substitution such that $\mathbb{T} \in \Diamond\mathcal{L}_\lambda$. If $\sigma\mathbb{T}$ is in $\mathcal{L}_\lambda$, then by definitions 7.3 and 7.4, the $\mathcal{L}_\lambda$`-link` associated to the subterm of $\mathbb{T}$ have been solved and removed. The unification is done between terms in $\mathcal{L}_\lambda$ and by theorem 5.16 fidelity is guaranteed. If $\sigma\mathbb{T}$ is in $\Diamond\mathcal{L}_\lambda$, then, by **??**, the unification fails, as per the corresponding unification in $\mathcal{F}_o$. ◻

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

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

Initially the $\mathcal{L}_\lambda$`-link` 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_x is assigned to x. Under this substitution the $\mathcal{L}_\lambda$`-link` becomes $x \vdash C_x =_{\mathcal{L}_\lambda} (D \cdot x)$, and by *progress-$\mathcal{L}_\lambda$-refine* 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 η`-link` (f is assigned to C), and then all the remaining links are solved. The final $\mathcal{H}_o$ substitution is $\sigma = \{A \mapsto \lambda x.x, B \mapsto f, C_x \mapsto (f \cdot x), D \mapsto f, E_x \mapsto x, F_x \mapsto C_x\}$ and is decompiled into $\rho = \{X \mapsto \lambda x.x, Y \mapsto f\}$.

Example of progress-$\mathcal{L}_\lambda$-rhs. We can take the example provided in section 7. The problem is compiled into:

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

The first unification problems is solved by the substitution $\sigma = \{A \mapsto \lambda x.B\}$. The $\mathcal{L}_\lambda$`-link` becomes $\vdash C =_{\mathcal{L}_\lambda} ((\lambda x.B) \cdot a)$ 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 $\rho = \{X \mapsto \lambda x.a, Y \mapsto a\}$.

7.3 Relaxing definition 7.5 (*progress-$\mathcal{L}_\lambda$-fail*)

Working with terms in $\mathcal{L}_\lambda$ is sometime too restrictive [1]. There exists systems such as Teyjus [10] and λProlog [11] which delay the resolution of $\Diamond\mathcal{L}_\lambda$ unification problems if the substitution is not able to put them in $\mathcal{L}_\lambda$.

In this section we want to show how we can adapt the unification of the object language in the meta language by simply adding (or removing) rules to the `progress` predicate.

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

In the example above, $\mathbb{P}_1$ is in $\diamond\mathcal{L}_\lambda$. *If* the object language delays the first unification problem waiting X to be be instantiated in a future unification, we can relax definition 7.5. Instead of failing because the lhs of the considered $\mathcal{L}_\lambda$-`link` l becomes rigid, we keep it in $\mathbb{L}$ until the head of its rhs also become rigid. In this case, since lhs and rhs have rigid heads, they can be unified just before removing l from $\mathbb{L}$. We can note that this rule trivially guarantees proposition 2.2 (Simulation fidelity). On the other hand, the occur check becomes partial: there exists $\mathcal{L}_\lambda$-`link` with a non-flexible lhs.

A second strategy to deal with problem that are in $\diamond\mathcal{L}_\lambda$ is to make approximations. This is the case for example of the unification algorithm of Coq used in its type class solver [17]. The approximation consists in forcing a choice (among the others) when the unification problem is outside $\mathcal{L}_\lambda$. 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 $\sigma = \{X \mapsto \lambda x.Y, Y \mapsto \_\}$ is another valid substitution for the original problem. We stress the fact that, again, our unification procedure in the meta language can be accommodated for this new behavior: given a $\mathcal{L}_\lambda$-`link`, if lhs is not in $\mathcal{L}_\lambda$, then `progress` can try to align the rightmost arguments and unify the resulting heads.

Note that delaying unification outside $\mathcal{L}_\lambda$ can leave $\mathcal{L}_\lambda$-`link` during the decompilation phase. Therefore, new rules to `commit-links` should be added accordingly.

cita teyjus (1) era 2nd order HO (huet's algorithm), teyjus 2 è llam ma sospende i disagreement pairs fuori da llam

8 ACTUAL IMPLEMENTATION IN ELPI

In this paper we show a minimized example. The full code is there. But we also have to code things in Coq-Elpi.

The main difference between the presentation in the previous sections and the actual implementation for Coq is that the main loop hrun is replaced by the one of Prolog that chains calls to the unification procedure. In order implement the store of links we resort to Elpi's CLP engine and use constraints (suspended goals) to represent links, and constraint handling rules to implement progress operations involving more than one link.

finish

about the progress of 1 link:

```
link-eta L R :- suspend-condition L R Holes, !,
  declare_constraint (link-eta L R) Holes.
link-eta L R :-
  progress. % e.g. L = R.
```

about the progress of 2 links:

```
constraint link-eta {
  rule (N1 ⊳ G1 ?- link-eta (uvar X LX1) T1) % match
    /  (N2 ⊳ G2 ?- link-eta (uvar X LX2) T2) % remove
    |  (relocate LX1 LX2 T2 T2')             % condition
   <=> (N1 ⊳ G1 ?- T1 = T2').                % new goal
}
```

Remark how the invariant about uvar arity makes this easy, since LX1 and LX2 have the same length. Also note that N1 only contains the names of the first link (while relocate runs in the disjoint union) and Elpi ensures that T2' can live in N1.

9 OTHER ENCODINGS AND RELATED WORK

One could ignore the similarity between $\simeq_o$ and $\simeq_\lambda$ and "just" describe the object language unification procedure in the meta language by crafting a `unif` routine and using it 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 meta language since, by removing any datum from the head of rules, indexing degenerates. Moreover the unification procedure `unif` programmed in the meta language is likely to be an order of magnitude slower than one that is built-in.

Another possibility is to avoid having the application and abstraction nodes in the syntax tree, and use the ones of meta language, as in the following:

```
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 CIC it is not always possible to adopt it since the type system of the meta language is too weak to accommodate for the one of the object language. In CIC the lambda abstraction has to carry a type in order to make type checking decidable. Moreover CIC allows for functions 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 type system of the λProlog is too stringent to accept this terms. The second reason is that the CIC encoding provided by Elpi is used for meta programming (extending) the Coq 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 this sense constants have to live in an open world, like the `string` data type used in the examples so far.

In the literature we could find related encoding of the Calculus of Constructions [3]. The goal of that work was to exhibit a logic program performing proof checking in CC and hence relate the proof system of intuitionistic higher-order logic (that animates λProlog programs) with the Calculus of Constructions. The encoding is hence tailored toward a different goal, and utilizes three relations to represent the equational theory of CC. Section 6 contains a discussion about the use of the unification procedure of the meta language in presence of non ground goals, but the authors do not aim at exploiting it to the degree we want.

sucks

cite isabelle's TC, that are baked in

10 CONCLUSION

Benefits: less work, reuse efficient ho unif (3x faster), indexing,

Future: tabling and static analysis (reuse for ML again).

Very little is Coq specific. Applies to all OL that are not a subsystem of HOL, or for ML that are used for meta programming.

REFERENCES

[1] Andreas Abel and Brigitte Pientka. "Extensions to Miller's Pattern Unification for Dependent Types and Records". In: 2018. URL: https://api.semanticscholar.org/CorpusID:51885863.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] Gopalan Nadathur. "The Metalanguage λprolog and Its Implementation". In: *Functional and Logic Programming*. Ed. by Herbert Kuchen and Kazunori Ueda. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–20. ISBN: 978-3-540-44716-0.

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

[12] 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.

[13] 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.

[14] 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.

[15] F. Pfenning and C. Elliott. "Higher-order abstract syntax". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 199–208. ISBN: 0897912691. DOI: 10.1145/53990.54010. URL: https://doi.org/10.1145/53990.54010.

[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] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

type extend-subst fm -> 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 beta-reduce fm -> fm -> o.
beta-reduce (uvar _ _) _ :- halt "Passed uvar to beta-reduce".
beta-reduce A A :- name A.
beta-reduce (fcon A) (fcon A).
beta-reduce (fuva A) (fuva A).
beta-reduce (flam A) (flam B) :-
  pi x\ beta-reduce (A x) (B x).
beta-reduce (fapp [flam B | L]) T2 :- !,
```

```
  beta (flam B) L T1, beta-reduce T1 T2.
beta-reduce (fapp L) (fapp L1) :-
  map beta-reduce L L1.

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

type eta-contract fm -> fm -> o.
eta-contract (fcon X) (fcon X).
eta-contract (fapp L) (fapp L1) :- map eta-contract L L1.
eta-contract (flam F) T :- eta-contract-aux [] (flam F) T.
eta-contract (flam F) (flam F1) :-
  pi x\ eta-contract x x => eta-contract (F x) (F1 x).
eta-contract (fuva X) (fuva X).
eta-contract X X :- name X.

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

13 THE META LANGUAGE

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

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

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

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

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

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

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

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

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

type beta tm -> list tm -> tm -> o.
beta A [] A :- !.
beta (lam Bo) [H | L] R :- beta (Bo H) L R1, beta-aux R1 R.
beta (app A) L (app X) :- append A L X.
```

```
beta (con H) L (app [con H | L]).
beta X L (app[X|L]) :- name X.

type beta-aux tm -> tm -> o.
beta-aux (app [HD|TL]) R :- !, beta HD TL R.
beta-aux A A.

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

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

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

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

type deref subst -> tm -> tm -> o.                    (σt)
deref _ (con C) (con C).
deref S (app A) (app B) :- 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.

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

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

14 THE COMPILER

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

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

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

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

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


type get-lhs link -> tm -> o.
get-lhs (val (link-llam A _)) A.
get-lhs (val (link-eta A _)) A.

type get-rhs link -> tm -> o.
get-rhs (val (link-llam _ A)) A.
get-rhs (val (link-eta _ A)) A.


type occurs-rigidly fm -> fm -> o.
occurs-rigidly N N.
occurs-rigidly _ (fapp [fuva _|_]) :- !, fail.
occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.
occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).

type reducible-to list fm -> fm -> fm -> o.
reducible-to _ N N :- !.
reducible-to L N (fapp[fuva _|Args]) :- !,
  forall1 (x\ exists (reducible-to [] x) Args) [N|L].
reducible-to L N (flam B) :- !,
  pi x\ reducible-to [x | L] N (B x).
reducible-to L N (fapp [N|Args]) :-
  last-n {len L} Args R,
  forall2 (reducible-to []) R {rev L}.
```

```
type maybe-eta fm -> list fm -> o.                          (◇η)
maybe-eta (fapp[fuva _|Args]) L :- !,
  forall1 (x\ exists (reducible-to [] x) Args) L, !.
maybe-eta (flam B) L :- !, pi x\ maybe-eta (B x) [x | L].
maybe-eta (fapp [T|Args]) L :- (name T; T = fcon _),
  split-last-n {len L} Args First Last,
  none (x\ exists (y\ occurs-rigidly x y) First) L,
  forall2 (reducible-to []) {rev L} Last.


type locally-bound tm -> o.
type get-scope-aux tm -> list tm -> o.
get-scope-aux (con _) [].
get-scope-aux (uva _ L) L1 :-
  forall2 get-scope-aux L R,
  flatten R L1.
get-scope-aux (lam B) L1 :-
  pi x\ locally-bound x => get-scope-aux (B x) L1.
get-scope-aux (app L) L1 :-
  forall2 get-scope-aux L R,
  flatten R L1.
get-scope-aux X [X] :- name X, not (locally-bound X).
get-scope-aux X [] :- name X, (locally-bound X).

type names1 list tm -> o.
names1 L :-
  names L1,
  new_int N,
  if (1 is N mod 2) (L1 = L) (rev L1 L).

type get-scope tm -> list tm -> o.
get-scope T Scope :-
  get-scope-aux T ScopeDuplicata,
  undup ScopeDuplicata Scope.
type rigid fm -> o.
rigid X :- not (X = fuva _).

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.

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 (_\[]) [].
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) (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].
```

```
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 [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-llam (uva B Scope) Beta) | L2].
comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :-                (c@)
  fold6 comp A A1 M1 M2 L1 L2 S1 S2.

type alloc mem A -> addr -> mem A -> o.
alloc S N S1 :- mem.new S N S1.

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

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

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

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

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

15 THE PROGRESS FUNCTION

```
macro @one :- s z.

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

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

type is-in-pf ho.tm -> o.
is-in-pf (ho.app [ho.uva _ _ | _]) :- !, fail.
is-in-pf (ho.lam B) :- !, pi x\ is-in-pf (B x).
is-in-pf (ho.con _).
is-in-pf (ho.app L) :- forall1 is-in-pf L.
is-in-pf N :- name N.
is-in-pf (ho.uva _ L) :- pattern-fragment L.

type arity ho.tm -> nat -> o.
arity (ho.con _) z.
arity (ho.app L) A :- len L A.

type occur-check-err ho.tm -> ho.tm -> ho.subst -> o.
occur-check-err (ho.con _) _ _ :- !.
occur-check-err (ho.app _) _ _ :- !.
occur-check-err (ho.lam _) _ _ :- !.
occur-check-err (ho.uva Ad _) T S :-
  not (ho.not_occ Ad S T).

type progress-beta-link-aux ho.tm -> ho.tm ->
       ho.subst -> ho.subst -> links -> o.
progress-beta-link-aux T1 T2 S1 S2 [] :-  is-in-pf T2, !,
  (T1 ==l T2) S1 S2.
progress-beta-link-aux T1 T2 S S [@val-link-llam 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-
  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-llam 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-llam

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-llam 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 N _) _).

  type same-link-eta link -> link -> ho.subst -> ho.subst -> o.
  same-link-eta (ho.abs F) B H H1 :- !, pi x\ same-link-eta (F x) B H H1.
  same-link-eta A (ho.abs G) H H1 :- !, pi x\ same-link-eta A (G x) H H1.
  same-link-eta (@val-link-eta (ho.uva N S1) A)
                (@val-link-eta (ho.uva N S2) B) H H1 :-
    std.map2 S1 S2 (x\y\r\ r = ho.copy x y) Perm,
    Perm => ho.copy A A',
    (A' ==l B) H H1.

  type progress1 links -> links -> ho.subst -> ho.subst -> o.
  progress1 [] [] X X.
  progress1 [A|L1] [A|L3] S S2 :- take-link A L1 B L2, !,
    same-link-eta A B S S1,
    progress1 L2 L3 S1 S2.
  progress1 [L0|L1] L3 S S2 :- deref-link S L0 L,
    solve-link-abs L R S S1, !,
    progress1 L1 L2 S1 S2, append R L2 L3.
```

16 THE DECOMPILER

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

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

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

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

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

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

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

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

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

```
  append L1 L2 LAll,
  complete-mapping H Tl LAll L3 F2 F3.

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

17 AUXILIARY FUNCTIONS

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

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