

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 [13], Twelf [15], λ Prolog [9] and Isabelle [21] which have been utilized to implement various formal systems such as First Order Logic [3], Set Theory [12], Higher Order Logic [11], and even the Calculus of Constructions [2].

The object logic we are interested in is Coq’s [19] Calculus of Inductive Constructions (CIC), for which we aim to implement a unification procedure \approx_o using the ML Elpi [1], a dialect of λ Prolog. Elpi’s equational theory comprises $\eta\beta$ equivalence and comes equipped with a higher order unification procedure \approx_λ restricted to the pattern fragment [8]. We want \approx_o to be as powerful as \approx_λ but on the object logic CIC. Elpi also comes with an encoding for CIC that works well for meta-programming [18, 17, 6, 4]. Unfortunately this encoding, which we refer to as \mathcal{F}_o , “underuses” \approx_λ by restricting it to first-order unification problems only. To address this issue, we propose a better-behaved encoding, \mathcal{H}_o , demonstrate how to map unification problems in \mathcal{F}_o to related problems in \mathcal{H}_o , and illustrate how to map back the unifiers found by \approx_λ , effectively implementing \approx_o on top of \approx_λ 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/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

Meta languages such as Elf [13], Twelf [15], λ Prolog [9] and Isabelle [21] have been utilized to specify various logics [3, 11, 12,

2]. The use of these meta languages facilitated 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 in proof construction and proof search.

The object logic we are interested in is Coq’s [19] Calculus of Inductive Constructions (CIC) and we want to implement a form of proof search known as type-class [20, 16] resolution. In particular we want to leverage the Elpi [18] meta programming language, a dialect of λ Prolog already used to extend Coq in various ways [18, 17, 6, 4]. 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”, hence we can expect that Elpi is a good fit for implementing such as form of automation. In this paper we focus on one aspect of this work, namely *how to reuse the higher order unification procedure of the meta language in order to implement a type-class solver for the object language*. As it turns out, re-using the unification of the meta language is not a trivial task.

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 Instances state that: 1) the type `fin n`, of natural numbers smaller than `n`, is finite; 2) the predicate `nfact n nf`, linking a natural number `n` to 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:  $\forall n$ , Finite (fin n). (* r1 *)
Instance nfact_dec:  $\forall n$  nf, Decision (nfact n nf). (* r2 *)
Instance forall_dec:  $\forall A$  P, Finite A  $\rightarrow$  (* r3 *)
 $\forall x:A$ , Decision (P x)  $\rightarrow$  Decision ( $\forall x:A$ , P x).
```

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

```
Decision ( $\forall x$ : fin 7, nfact x 3) (* g *)
```

The proof found by the solver back-chains on rule 3 (the only rule about the \forall quantifier), and then solves the premises with rules rules 1 and 2 respectively. Note that rule 3 features a second order parameter `P` that stands for a function of type `A \rightarrow 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 by solving the unification problem `P x = nfact x 3`. This higher order problem falls in the so called pattern-fragment \mathcal{L}_λ [8] and hence admits a unique solution `P = λx . nfact x 3`.

In order to implement such a search in Elpi we shall describe the encoding of CIC terms and then the encoding of rules. Elpi comes

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

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

with an Higher Order Abstract Syntax [14] 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 λ Prolog [9]'s standard syntax, the meta level binding of a variable x in an expression e is written $\langle x \setminus e \rangle$, and square brackets denote a list of terms separated by comma. For example the term $\langle \forall y:t, \text{nfact } y \ 3 \rangle$ 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 $\text{pi } w \setminus p$ 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\ app [con "nfact", y, con "3"]) . (g)
```

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 rules that, on a first approximation, would shape (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"] (p)
```

Once the head of rule (r3') unifies with the goal (g) the premise $\langle \text{link } Pm \ A \ P \rangle$ brings the assignment (ρ) back to the domain `tm` of Coq terms:

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

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

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

The equational theory of the meta language Elpi is strikingly similar, since it it comprises $\eta\beta$ (for the meta language functions), and the unification procedure \approx_λ solves problems in \mathcal{L}_λ as well.

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

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 \approx_λ can be effectively used to simulate a unification procedure \approx_o for the object language that features $\eta\beta$ -conversion in the pattern-fragment.

section 2 formally states the problem and gives the intuition behind our solution. section 9 discusses alternative term encodings and related works. section 3 introduces the languages \mathcal{F}_o and \mathcal{H}_o , section 4 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 λ Prolog 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 variables (fuva term constructor) in \mathcal{F}_o have no explicit scope: the arguments of an higher order variable are given via the `fapp` constructor. For example the term $\langle P \ x \rangle$ is represented as $\langle \text{fapp}[\text{fuva } N, \ x] \rangle$, where N is a memory address and x is a bound variable.

In \mathcal{H}_o the representation of $\langle P \ x \rangle$ is instead $\langle \text{uva } N \ [x] \rangle$, since

```

233 kind fm type.          kind tm type.
234 type fapp list fm -> fm. type app list tm -> tm.
235 type flam (fm -> fm) -> fm. type lam (tm -> tm) -> tm.
236 type fcon string -> fm.   type con string -> tm.
237 type fuva addr -> fm.     type uva addr -> list tm -> tm.

```

Figure 1: The \mathcal{F}_0 and \mathcal{H}_0 languages

unification variables are higher order and come equipped with an explicit scope.

Notational conventions. When we write \mathcal{H}_0 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:

```

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

```

When it is clear from the context we shall use the same syntax for \mathcal{F}_0 terms (although we never subscript unification variables).

We use s, s_1, \dots for terms in \mathcal{F}_0 and $t, t_1 \dots$ for terms in \mathcal{H}_0 .

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 just 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.

```

270 type ( $=_o$ ) fm -> fm -> o.          ( $=_o$ )
271 fcon X =o fcon X.
272 fapp A =o fapp B :- forall2 ( $=_o$ ) A B.
273 flam F =o flam G :- pi x\ x =o x => F x =o G x.
274 fuva N =o fuva N.
275 flam F =o T :-                               ( $\eta_l$ )
276   pi x\ beta T [x] (T' x), x =o x => F x =o T' x.
277 T =o flam F :-                               ( $\eta_r$ )
278   pi x\ beta T [x] (T' x), x =o x => T' x =o F x.
279 fapp [flam X | L] =o T :- beta (flam X) L R, R =o T. ( $\beta_l$ )
280 T =o fapp [flam X | L] :- beta (flam X) L R, T =o R. ( $\beta_r$ )
281
282 type ( $=_\lambda$ ) tm -> tm -> o.
283 con C =\lambda fcon C.
284 app A =\lambda fapp B :- forall2 ( $=_\lambda$ ) A B.
285 lam F =\lambda flam G :- pi x\ x =\lambda x => F x =\lambda G x.
286 uva N A =\lambda fuva N B :- forall2 ( $=_\lambda$ ) A B.

```

The main point in showing these equality tests is to remark how weaker $=_\lambda$ is, and to identify the four rules that need special treatment in the implementation of \approx_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 $\sigma = \{ X \mapsto t \}$ 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 $\sigma \cup \sigma'$ set union to denote the concatenation of two substitutions whose domains are disjoint. We shall use ρ for \mathcal{F}_0 substitutions, and σ for the \mathcal{H}_0 ones. For brevity, in this section we consider the substitution for \mathcal{F}_0 and \mathcal{H}_0 identical. We defer to section 3 a more precise description pointing out their differences.

Term unification: \approx_o vs. \approx_λ . Although we provide an implementation of \approx_λ in the supplementary material (that we used for testing purposes) we only describe its signature here.

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

The meta language of choice is expected to provide an implementation of \approx_λ that satisfies the following properties:

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

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

We write $\sigma t_1 \approx_\lambda \sigma t_2 \mapsto \sigma'$ when σt_1 and σt_2 unify with substitution σ' . We write $t_1 \approx_\lambda t_2 \mapsto \sigma$ when the initial substitution is empty. Note that if $\sigma t_1 \approx_\lambda \sigma t_2 \mapsto \sigma'$ then the domains of σ and σ' are disjoint.

Although we provide an implementation of \approx_o in section 4.4, 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}_0 as a list *steps* p of length 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 ρ_N , that is the result of the logic-program execution.

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

In order to simulate a \mathcal{F}_0 logic program in \mathcal{H}_0 we compile each term $s \in \mathcal{F}_0$ into a term $t \in \mathcal{H}_0$. We write this step $\langle s \rangle \mapsto (t, m, l)$. The implementation of the compiler is detailed in sections 4.1, 5 and 7, here we just point out that it additionally a variable mapping m and list of links l . The variable map connects unification variables in \mathcal{H}_0 with variables in \mathcal{F}_0 and is used to “decompile” the assignment, $\langle \sigma, m, l \rangle^{-1} \mapsto \rho$. Links are an accessory piece of information whose description is deferred to section 2.4.

We simulate each run in \mathcal{F}_0 with a run in \mathcal{H}_0 as follows.

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

¹If 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.

can
be
made
a
little
more
formal

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

By analogy with \mathbb{P} , we write \mathbb{T}_{p_l} and \mathbb{T}_{p_r} for the two \mathcal{H}_0 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_λ . 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 $\mathbb{P} \subseteq \mathcal{L}_\lambda$

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

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

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

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

In particular this property guarantees that a *failure* in the \mathcal{F}_0 run is matched by a failure in \mathcal{H}_0 at the same step. We consider this property very important from a practical point of view since it guarantees that the execution traces are strongly related and in turn this enables a user to debug a logic program in \mathcal{F}_0 by looking at its execution trace in \mathcal{H}_0 .

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

PROPOSITION 2.3 (FIDELITY RECOVERY).

$$\exists \rho, \rho s_1 =_o \rho s_2 \Rightarrow \langle s_i \rangle \mapsto (t_i, m_i, l_i) \Rightarrow t_1 \simeq_\lambda t_2 \mapsto \sigma \quad (3)$$

$$\sigma_{p-1} \mathbb{T}_p \in \mathcal{L}_\lambda \Rightarrow \quad (4)$$

$$\exists \rho, \text{fstep}(\mathbb{P}, p, \rho) \mapsto \rho' \Leftrightarrow \text{hstep}(\mathbb{T}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p)$$

Property 3 states that two terms for which there is a unifier (given by an oracle and not necessarily a most general one), then the compiler generates two terms that unify in \mathcal{H}_0 .

Property 4 says that if the two terms involved in a step re-enter \mathcal{L}_λ , then hstep succeeds. This is a typical example in which the order of the unification problems in a logic-program run does matter. The simplest example is the sequence $F \simeq \lambda x.a$ and $F a \simeq a$: the second problem is not in \mathcal{L}_λ 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}_λ . In other words proposition 2.2

2.4 The solution (in a nutshell)

A term s is compiled in a term t where every “problematic” sub term p is replaced by a fresh unification variable h and an accessory *link* that represents a suspended unification problem $h \simeq_\lambda p$. As a result \simeq_λ 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.

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

An example of term t in $\diamond \beta_0$ is the application $F x$. 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 t such that $\exists \rho, \rho t$ is an eta expansion.

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

Definition 2.6 ($\diamond \mathcal{L}_\lambda$). $\diamond \mathcal{L}_\lambda$ is the set of terms of the form $X t_1 \dots t_n$ 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}_λ .

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 $X \subseteq \mathcal{H}_0$,

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

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

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

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

Proposition 2.8 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_λ can operate properly. In sections 4.1, 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 and 2.2.

3 GROUND WORK FOR THE COMPILER

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}_0 unification variables has a scope, its solution needs to be abstracted over it to enable the instantiation of a single 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}_0 variable is a plain term.

```
typeabbrev fsubst (mem fm).
```



```

465   kind inctx type -> type.                (· ⊢ ·)
466   type abs (tm -> inctx A) -> inctx A.
467   type val A -> inctx A.
468   typeabbrev assignment (inctx tm).
469   typeabbrev subst (mem assignment).

```

We call `fsubst` the memory of \mathcal{F}_0 , while we call `subst` the one of \mathcal{H}_0 . The compiler establishes a mapping between variables of the two languages.

```

473   kind arity type.
474   type arity nat -> arity.
475
476   kind fvariable type.
477   type fv addr -> fvariable.
478
479   kind hvariable type.
480   type hv addr -> arity -> hvariable.
481
482   kind mapping type.
483   type (<->) fvariable -> hvariable -> mapping.
484   typeabbrev mmap (list mapping).
485

```

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}_0 has a (unique) arity N and each occurrence $(\text{uva } A \text{ } L)$ is such that $(\text{len } L \text{ } N)$ holds*

```

492   type m-alloc fvariable -> hvariable -> mmap -> mmap ->
493     subst -> subst -> o.                (malloc)
494   m-alloc Fv Hv M M S S :- mem M (Fv <-> Hv), !.
495   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. 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}_0 variables faithfully represent the problematic term thanks to their scope.

```

505   kind baselink type.
506   type link-eta tm -> tm -> baselink.
507   type link-beta tm -> tm -> baselink.
508   typeabbrev link (inctx baselink).
509   typeabbrev links (list link).

```

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

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

New links are suspended by construction. If the left hand side variable is assigned during a step, then the link is considered for progress and possibly eliminated. This is discussed in section 5 and section 7.

Applying the substitution corresponds to dereferencing a term with respect to the memory. To ease the comparison we split \mathcal{F}_0

dereferencing into a `fder` step and a `napp` one. The former step replaces references to memory cells that are set with their values, and has a corresponding operation in \mathcal{H}_0 , namely `deref`. On the contrary `napp` has no corresponding operation in \mathcal{H}_0 , and only ensures that terms of the form $\llbracket \text{fapp } L1 \mid L2 \rrbracket$ are replaced by $\llbracket \text{fapp } L3 \rrbracket$ where $L3$ is the concatenation of $L1$ and $L2$. The reasons for this asymmetry is that an `fapp` node with a flexible head is always mapped to a `uva` (as per sections 4 and 7), preventing nested applications to materialize.

```

532   type fder fsubst -> fm -> fm -> o.
533   fder _ (fcon C) (fcon C).
534   fder S (fapp A) (fapp B) :- map (fder S) A B.
535   fder S (flam F) (flam G) :-
536     pi x\ fder S x x => fder S (F x) (G x).
537   fder S (fuva N) R :- set? N S T, fder S T R.
538   fder S (fuva N) (fuva N) :- unset? N S.
539

```

```

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

```

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

```

547   type deref subst -> tm -> tm -> o.                (st)
548   deref _ (con C) (con C).
549   deref S (app A) (app B) :- map (deref S) A B.
550   deref S (lam F) (lam G) :-
551     pi x\ deref S x x => deref S (F x) (G x).
552   deref S (uva N L) R :- set? N S A,
553     move A L T, deref S T R.
554   deref S (uva N A) (uva N B) :- unset? N S,
555     map (deref S) A B.

```

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

```

560   type move assignment -> list tm -> tm -> o.
561   move (abs Bo) [H|L] R :- move (Bo H) L R.
562   move (val A) [] A.
563

```

3.1 Notational conventions

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

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

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

```

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

```

4 BASIC SIMULATION OF \mathcal{F}_0 IN \mathcal{H}_0

In this section we describe a basic compilation scheme that we refine later, in the following sections. This scheme is sufficient to implement a `hstep` that respects β -conversion for terms in \mathcal{L}_λ . The

notation
for
problem
compila-
tion

extension to $\eta\beta$ -conversion is described in section 5 and the support for terms outside \mathcal{L}_λ in section 7.

4.1 Compilation

E:manca beta normal in entrata

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

The signature of the comp predicate below allows for the generation of links (suspended unification problems) that play no role in this section but play a major role in 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 [L]) 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).

```
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 $\pi x y \dots$ is syntactic sugar for iterated π abstraction, as in $\pi x \backslash \pi y \dots$.

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.

4.1.1 Compilation of terms in $\diamond\beta_0$. 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

4.2 Execution

A step in \mathcal{H}_0 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 that the infix notation $((A \approx_\lambda B) C D)$ is syntactic sugar for $((\approx_\lambda) A B C D)$.

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

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

In the base compilation scheme `progress1` is the identity on both the links and the substitution, so the fixpoint trivially terminates. Sections 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} .

Since compilation moves problematic terms out of the sigh of \approx_λ , that procedure can only perform a partial occur check. For example the unification problem $X \approx_\lambda f Y$ cannot generate a cyclic substitution alone, but should be disallowed if a \mathbb{L} contains a link like $\vdash Y =_\eta \lambda z.X_z$: We don't know yet if Y will feature a lambda in head position, but we surely know it contains X , hence $f Y$ and that 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).

4.3 Substitution decompilation

Decompiling the substitution requires to first force the progress of links and then allocating new unassigned variables in the substitution for \mathcal{F}_0 and finally decompiling all assignments. Note that invariant 2 and the occur check allows us to update the subst.

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

```

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

```

E:What
is
commit-
links
and
complete-
mapping?

Decompiling an assignment requires to turn abstractions into lamb-
das. For aesthetic purposes we also eta-contract the result (not
needed since \mathcal{F}_0 equality can do that)

```

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

```

Finally decompiling a term is trivial, now that we have an extended
mapping containing all unassigned variables \approx_λ may have intro-
duced.

```

718   type decomp mmap -> tm -> fm -> o.
719   decomp _ (con C) (fcon C).
720   decomp M (app A) (fapp B) :- map (decomp M) A B.
721   decomp M (lam F) (flam G) :-
722     pi x y\ (pi M\ decomp M x y) => decomp M (F x) (G y).
723   decomp M (uva Hv Ag) R :-
724     mem M (fv Fv <-> hv Hv _),
725     map (decomp M) Ag Bg,
726     beta (fuva Fv) Bg R.

```

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

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

4.4 Definition of \approx_o and its properties

```

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

```

The code given so far applies to terms in $\beta\eta$ -normal form where
unification variables in \mathcal{F}_0 can occur non linearly but always with
the same number of arguments, and where their arguments are
distinct names (as per \mathcal{L}_λ).

LEMMA 4.1 (COMPILATION ROUND TRIP). *If comp S T [] M [] _ [] -
then decomp M T S*

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

LEMMA 4.2. *Properties ?? and ?? hold for the implementation of
 \approx_o above*

PROOF SKETCH. In this setting $=_\lambda$ is as strong as $=_o$ on ground
terms. What we have to show is that whenever two different \mathcal{F}_0 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 \approx_λ on the corresponding \mathcal{H}_0
terms and by decompiling it. If we look at the \mathcal{F}_0 terms, the are two
interesting cases:

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

Since the mapping is a bijection occur check in \mathcal{H}_0 corresponds to
occur check in \mathcal{F}_0 . □

LEMMA 4.3. *Properties simulation (2.1) and fidelity (2.2) hold*

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

4.5 Limitations of by this basic scheme

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

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

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

5 HANDLING OF $\diamond\eta$

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

Following the compilation scheme of section 4.1 the unification
problem \mathbb{P} is compiled as follows:

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

While $\lambda x.X x \approx_o f$ does admit the solution $\rho = \{X \mapsto f\}$, the
corresponding problem in \mathbb{T} does not: $\text{lam } x \backslash \text{uva } A [x]$ and con "f"
start with different, rigid, term constructors hence \approx_λ 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 com-
pilation of the problem \mathbb{P} above is refined to:

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

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 4 (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 4.2).

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 seem, here a few examples:

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

The first two examples are easy, and show how a unification variable can expose or erase a variable 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 $\lambda y.f(A\ x)(B\ y\ x)$ is in $\diamond\eta$ itself. If it is, it could η -contract to $f(A\ x)$ making $\lambda x.f(A\ x)$ 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. $\lambda x_1 \dots x_n.x\ x_1 \dots x_n =_o x$);
- (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). \square

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 disappear 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 \square

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 $\lambda x.f(A\ x)(A\ x)$ since x does not *occurs-rigidly* in the first argument of f , and the second argument of f *may-contract-to* x . In other words $A\ x$ 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 4.1.

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

The rule triggers when the input term `flam F` is in $\diamond\eta$. It compiles `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 4.*

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. \square

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 $X = f a_1 \dots a_n$ we unify the η -expansion of X with T , that is we run $\lambda x.X x \simeq_{\lambda} T$
- (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. *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 2.8). lhs $\in \sigma$, therefore $\mathcal{W}(\text{lhs})$. By definition 5.7, if 1) lhs is a name, a constant or an application, then, $\lambda x.\text{lhs } x$ is unified with rhs. By invariant 4, rhs = $\lambda x.t$, therefore $\mathcal{W}(t)$. Otherwise, 2) lhs has lam as functor, rhs should not be an η -expansion, so, $\mathcal{W}(\text{rhs})$. In both cases, unification is performed between terms in \mathcal{W} . \square

LEMMA 5.11. *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, $\mathcal{W}(\text{rhs})$. Otherwise, rhs can reduce to a term which cannot be a η -expansion, and, so, $\mathcal{W}(\text{rhs})$. In both cases, unification is done between terms in \mathcal{W} . \square

LEMMA 5.12. *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 $\lambda x.t$, with $\mathcal{W}(t)$, therefore, the unification is done between well-behaved terms. \square

LEMMA 5.13. *The introduction of η -link guarantees proposition 2.8 (\mathcal{W} -preservation)*

PROOF SKETCH. By lemmas 5.10 to 5.12, every unification performed by the activation of a η -link is done between terms in \mathcal{W} , therefore, the substitution remains \mathcal{W} .

D:Bisogna aggiungere un lemma nella section 2.4 che dice che unificare due termini in \mathcal{W} , in una σ , tale che $\mathcal{W}(\sigma)$, non invalida \mathcal{W}

\square

LEMMA 5.14. *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_{λ} , η -contraction, η -expansion, relocation (a recursive copy of a finite term). \square

LEMMA 5.15 (FIDELITY WITH η -link). *The introduction of η -link guarantees proposition 2.2 (Simulation fidelity)*

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}_0 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. \square

Example of progress- η -left. The example at the beginning of section 5, once $\sigma = \{ A \mapsto f \}$, triggers *progress- η -left* since the link becomes $\vdash f =_{\eta} \lambda x.B_x$ and the lhs is a constant. In turn the rule runs $\lambda x.f x \simeq_{\lambda} \lambda x.B_x$, 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 $\lambda x.f x$.

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

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

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 ENFORCING INVARIANT 1

We report here the problem given in section 4.5 where X is used with two different arities and the output of the compilation does not respect invariant 3 (merging the two mappings for s would break invariant 1). In this section we explain how to replace the duplicate mapping with some η -link in order to restore the invariants.

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

We see that the maybe-eta as identified $\lambda xy. X y x$ and $\lambda x. f (X x) x$ and the compiler has replaced them with A and D respectively. However, the mapping \mathbb{M} breaks invariant 3: the \mathcal{F}_0 variable X is mapped to two different \mathcal{H}_0 variables. To address this problem we adjust the compiler's output with a map-deduplication procedure.

Definition 6.1 (align-arity). Given two mappings $m_1 : X \mapsto A^m$ and $m_2 : X \mapsto C^n$ where $m < n$ and $d = n - m$, align-arity $m_1 m_2$ generates the following d links, one for each i such that $0 \leq i < d$,

$$x_0 \dots x_{m+i} \vdash B_{x_0 \dots x_{m+i}}^i =_{\eta} \lambda x_{m+i+1}. B_{x_0 \dots x_{m+i+1}}^{i+1}$$

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

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

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

In this example, \mathbb{T}_1 assigns A which triggers \mathbb{L}_3 and then \mathbb{L}_4 by definition 5.7. C_{yx} is therefore assigned to x (the second variable of its scope). We can finally see the *progress- η -right* of \mathbb{L}_1 : its rhs is now $\lambda y. y (C_{xy} \text{ gives } y)$. Since it is no more in $\diamond \eta$, $\lambda y. y$ is unified with E_x . Moreover, \mathbb{L}_2 is also triggered due to definition 5.8: $\lambda x. (f (\lambda y. y) x)$ is η -reducible to $f (\lambda y. y)$ which is a term not starting with the lam constructor.

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

In general, unification between $\diamond \mathcal{L}_{\lambda}$ terms admits more than one solution and committing one of them in the substitution does not guarantee ???. For instance, $X a \approx_o a$ 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.

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

$$\begin{aligned} \mathbb{P} &= \{ X \approx_o \lambda x. Y \quad (X a) \approx_o a \} \\ \mathbb{T} &= \{ A \approx_{\lambda} \lambda x. B \quad (A a) \approx_{\lambda} a \} \\ \mathbb{M} &= \{ Y \mapsto B^0 \quad X \mapsto A^0 \} \end{aligned}$$

In the example above, we see that \mathbb{P}_1 instantiates X so that \mathbb{P}_2 can be solved in \mathcal{L}_{λ} . On the other hand, we see that, \approx_{λ} can't solve the compiled problems \mathbb{T} . In fact, the resolution of \mathbb{T}_1 gives the substitution $\sigma = \{A \mapsto \lambda x. B\}$, but the dereferencing of \mathbb{T}_2 gives the non-unifiable problem $(\lambda x. B) a \neq_{\lambda} a$.

To address this unification problem, term compilation should capture the terms in $\diamond \mathcal{L}_{\lambda}$ and replace them with fresh variables. This replacement should produce links that we call β -link.

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

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

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}_{λ} . 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-beta (uva B Scope) Beta) | L2].
```

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

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

COROLLARY 7.1. Let $X_{s_1 \dots s_n} t_1 \dots t_m$ be the rhs of a β -link, then $m > 0$.

PROOF SKETCH. Assume we have a β -link, by contradiction, if $m = 0$, then the original \mathcal{F}_0 term has the shape $\text{fapp}[\text{fuva } M \mid \text{Ag}]$ 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 4.1) and no β -link is produced which contradicts our initial assumption. \square

COROLLARY 7.2. Let $X_{s_1 \dots s_n} t_1 \dots t_m$ be the rhs of a β -link, then t_1 either appears in $s_1 \dots s_n$ or it is not a name.

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

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

Decompilation. During progress, as claimed in invariant 5, the decompilation can only have β -link with not instantiated lhs. In this case, lhs is unified with rhs.

D: not really sure of this, we can have $F a = \lambda x. Gx$. In this case when do we fail: for sure in decompile. But to respect fidelity, we should fail immediately: we have a β -link and a η -link with same lhs

7.2 Progress

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

Definition 7.3 (progress-beta- \mathcal{L}_λ). Given a substitution σ and a β -link $\Gamma \vdash T =_\beta X_{s_1 \dots s_n} t_1 \dots t_m$ such that σt_1 is a name, say t , and $t \notin s_1 \dots s_n$. If $m = 0$, then the β -link is removed and lhs is unified with $X_{s_1 \dots s_n}$. If $m > 0$, then the β -link is replaced by a refined version $\Gamma \vdash T =_\beta Y_{s_1 \dots s_n, t} t_2 \dots t_m$ with reduced list of arguments and Y being a fresh variable. Moreover, the new link $\Gamma \vdash X_{s_1 \dots s_n} =_\eta \lambda x. Y_{s_1 \dots s_n, x}$ is added to \mathbb{L} .

Definition 7.4 (progress-beta-rigid-rhs). A link $\Gamma \vdash X =_\beta X_{s_1 \dots s_n} t_1 \dots t_m$ is removed from \mathbb{L} if $X_{s_1 \dots s_n}$ is instantiated to a term t and the β -reduced term t' obtained from the application of t to $t_1 \dots t_m$ is in \mathcal{L}_λ . Moreover, X is unified to t .

Definition 7.5 (progress-beta-dedup). Given a β -link l_1 and second link $l_2 \in \mathbb{L}$, such that they share the same lhs. The two rhs are unified and a l_2 is removed from \mathbb{L} .

Definition 7.6 (progress-rigid-lhs). Given a β -link with rigid lhs, the unification fails.

LEMMA 7.7. *progress terminates*

PROOF SKETCH. Let l a β -link in the store \mathbb{L} . If l is activated by *progress-beta-rigid-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-beta- \mathcal{L}_λ* , l is replaced by a new β -link l^1 having $m - 1$ arguments. At the m^{th} iteration, the β -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 5.14, the algorithm terminates. Finally *progress-beta-dedup* (resp. *progress-rigid-lhs*) also guarantees termination since it removes a link from \mathbb{L} (resp. immediately fails).

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

LEMMA 7.8 (FIDELITY WITH β -link). *The introduction of β -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}_λ , then by definitions 7.3 and 7.4, the β -link associated to the subterm of \mathbb{T} have been solved and removed. The unification is done between terms in \mathcal{L}_λ and by lemma 5.15 fidelity is guaranteed. If $\sigma \mathbb{T}$ is in $\diamond \mathcal{L}_\lambda$, then, by definition 7.6, the unification fails, as per the corresponding unification in \mathcal{F}_0 . \square

Example of progress-beta- \mathcal{L}_λ . Consider the β -link below:

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

Initially the β -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 β -link becomes $x \vdash C_x =_\beta (D x)$, and by *progress-beta- \mathcal{L}_λ* 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}_0 substitution is $\sigma = \{A \mapsto \lambda x. x, B \mapsto f, C_x \mapsto (f x), D \mapsto f, E_x \mapsto x, F_x \mapsto C_x\}$ and is decompiled into $\rho = \{X \mapsto \lambda x. x, Y \mapsto f\}$.

Example of progress-beta-rigid-rhs. We can take the example provided in section 7. The problem is compiled into:

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

The first unification problems is solved by the substitution $\sigma = \{A \mapsto \lambda x. B\}$. The β -link becomes $\vdash C =_\beta ((\lambda x. B) a)$ whose rhs can be β -reduced to B . B is in \mathcal{L}_λ 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 lemma ??

Working with terms in \mathcal{L}_λ is sometime too restrictive. There exists systems such as λ Prolog [10], Abella [5], which delay the resolution of $\diamond \mathcal{L}_\lambda$ unification problems if the substitution is not able to put them in \mathcal{L}_λ .

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

In the example above, \mathbb{P}_1 is in $\diamond \mathcal{L}_\lambda$ and the object language cannot solve it, and, by proposition 2.2, the meta language neither. However, we can be more permissive, and relax ???. This modification is quite simple to manage: we are introducing a new $\diamond \mathcal{L}_\lambda$ progress rule, say *progress-beta- $\diamond \mathcal{L}_\lambda$* , by which, if lhs is rigid and rhs is flexible, the considered β -link is kept in the store and no progression is done². *progress-beta- $\diamond \mathcal{L}_\lambda$* makes occur-check-links partial, since the check is possible only on links with a variable on the lhs. This means that we can have two links $\vdash X =_\beta Y a$ and $\vdash f X =_\beta Y a$ where the occur check does not throw an error. Note however, that the decompilation of the two links will force the unification of X to $Y a$ and then the unification of $f (Y a)$ to $Y a$, which fails by the occur check of \approx_λ .

A second strategy to deal with problem that are in $\diamond \mathcal{L}_\lambda$ is to make some approximation. This is the case for example of the unification algorithm of Coq used in its type class solver [16]. The approximation consists in forcing a choice (among the others) when

²This new rule trivially guarantees the termination of progress

the unification problem is in $\diamond \mathcal{L}_\lambda$. For instance, in $X a b = Y b$, 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. This approximation can be easily introduced in our unification procedure, by adding new custom β -link progress rules.

Decompilation of β -link is possible by extending commit-link with new heuristics.

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.

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 \approx_o and \approx_λ 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 λ Prolog 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 [2]. 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 λ Prolog 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.

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] Cvetan Dunchev et al. “ELPI: Fast, Embeddable, λ Prolog Interpreter”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis et al. Vol. 9450. 2015, pp. 460–468. doi: 10.1007/978-3-662-48899-7_32. URL: http://dx.doi.org/10.1007/978-3-662-48899-7_32.
- [2] 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.
- [3] 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.

- [4] 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>.
- [5] Andrew Gacek. *The Abella Interactive Theorem Prover (System Description)*. 2008. arXiv: 0803.2305 [cs.LO].
- [6] Benjamin Grégoire, Jean-Christophe L  chenet, and Enrico Tassi. “Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin’s data types with Coq-Elpi”. In: *CPP ’23: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. Boston MA USA, France: ACM, Jan. 2023, pp. 167–181. DOI: 10.1145/3573105.3575683. URL: <https://inria.hal.science/hal-03800154>.
- [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 and Dale Miller. “An Overview of Lambda-Prolog”. In: June 1988, pp. 810–827.
- [11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7.
- [12] Lawrence C. Paulson. “Set theory for verification. I: from foundations to functions”. In: *J. Autom. Reason.* 11.3 (Dec. 1993), pp. 353–389. ISSN: 0168-7433. DOI: 10.1007/BF00881873. URL: <https://doi.org/10.1007/BF00881873>.
- [13] F. 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.
- [14] 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>.
- [15] Frank Pfenning and Carsten Sch  rmann. “System Description: Twelf – A Meta-Logical Framework for Deductive Systems”. In: *Automated Deduction – CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2.
- [16] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, C  sar Mu  oz, and Sof  ene Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-71067-7.
- [17] 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>.
- [18] Enrico Tassi. “Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog dialect)”. In: *The Fourth International Workshop on Coq for Programming Languages*. Los Angeles (CA), United States, Jan. 2018. URL: <https://inria.hal.science/hal-01637063>.
- [19] The Coq Development Team. *The Coq Reference Manual – Release 8.18.0*. <https://coq.inria.fr/doc/V8.18.0/refman>. 2023.
- [20] 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>.
- [21] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. “The Isabelle Framework”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, C  sar Mu  oz, and Sof  ene Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7.

APPENDIX

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

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

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

11 THE MEMORY

```

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

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

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

type assign-aux nat -> mem A -> A -> mem A -> o.
assign-aux z (none :: L) Y (some Y :: L).
assign-aux (s N) (X :: L) Y (X :: L1) :- assign-aux N L Y L1.

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

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

type alloc-aux nat -> mem A -> mem A -> o.
alloc-aux z [] [none] :- !.
alloc-aux z L L.
alloc-aux (s N) [] [none | M] :- alloc-aux N [] M.
alloc-aux (s N) [X | L] [X | M] :- alloc-aux N L M.

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

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

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

```

12 THE OBJECT LANGUAGE

```

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

typeabbrev fsubst (mem fm).

```

```

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

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

```

```

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

```

```

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

```

```

type beta fm -> list fm -> fm -> o.
beta A [] A.
beta (flam Bo) [H | L] R :- napp (Bo H) F, beta F L R.
beta (fapp A) L (fapp X) :- append A L X.
beta (fuva N) L (fapp [fuva N | L]).
beta (fcon H) L (fapp [fcon H | L]).
beta N L (fapp [N | L]) :- name N.

```

```

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

```

```

type 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 :- !,

```

```

1625     beta (flam B) L T1, beta-reduce T1 T2.
1626 beta-reduce (fapp L) (fapp L1) :-
1627     map beta-reduce L L1.
1628
1629 type mk-app fm -> list fm -> fm -> o.
1630 mk-app T L S :- beta T L S.
1631
1632 type eta-contract fm -> fm -> o.
1633 eta-contract (fcon X) (fcon X).
1634 eta-contract (fapp L) (fapp L1) :- map eta-contract L L1.
1635 eta-contract (flam F) T :- eta-contract-aux [] (flam F) T.
1636 eta-contract (flam F) (flam F1) :-
1637     pi x\ eta-contract x x => eta-contract (F x) (F1 x).
1638 eta-contract (fuva X) (fuva X).
1639 eta-contract X X :- name X.
1640
1641 type eta-contract-aux list fm -> fm -> fm -> o.
1642 eta-contract-aux L (flam F) T :-
1643     pi x\ eta-contract-aux [x|L] (F x) T. % also checks H Prefix does not
1644 eta-contract-aux L (fapp [H|Args]) T :-
1645     rev L LRev, append Prefix LRev Args,
1646     if (Prefix = []) (T = H) (T = fapp [H|Prefix]).
1647
1648
1649
1650 kind inctx type -> type.
1651 type abs (tm -> inctx A) -> inctx A.
1652 type val A -> inctx A.
1653 typeabbrev assignment (inctx tm).
1654 typeabbrev subst (mem assignment).
1655
1656 kind tm type.
1657 type app list tm -> tm.
1658 type lam (tm -> tm) -> tm.
1659 type con string -> tm.
1660 type uva addr -> list tm -> tm.
1661
1662 type ( $\approx_\lambda$ ) tm -> tm -> subst -> subst -> o.
1663 (con C  $\approx_\lambda$  con C) S S.
1664 (app L1  $\approx_\lambda$  app L2) S S1 :- fold2 ( $\approx_\lambda$ ) L1 L2 S S1.
1665 (lam F1  $\approx_\lambda$  lam F2) S S1 :-
1666     pi x\ (pi S\ (x  $\approx_\lambda$  x) S S) => (F1 x  $\approx_\lambda$  F2 x) S S1.
1667 (uva N Args  $\approx_\lambda$  T) S S1 :-
1668     set? N S F,!, move F Args T1, (T1  $\approx_\lambda$  T) S S1.
1669 (T  $\approx_\lambda$  uva N Args) S S1 :-
1670     set? N S F,!, move F Args T1, (T  $\approx_\lambda$  T1) S S1.
1671 (uva M A1  $\approx_\lambda$  uva N A2) S1 S2 :- !,
1672     pattern-fragment A1, pattern-fragment A2,
1673     prune! M A1 N A2 S1 S2.
1674 (uva N Args  $\approx_\lambda$  T) S S1 :- not_occ N S T, pattern-fragment Args,
1675     bind T Args T1, assign N S T1 S1.
1676 (T  $\approx_\lambda$  uva N Args) S S1 :- not_occ N S T, pattern-fragment Args,
1677     bind T Args T1, assign N S T1 S1.
1678
1679 type prune! addr -> list tm -> addr ->
1680     list tm -> subst -> subst -> o.
1681 /* no pruning needed */
1682

```

```

1683 prune! N A N A S S :- !.
1684 prune! M A N A S1 S2 :- !, bind (uva M A) A Ass,
1685     assign N S1 Ass S2.
1686 /* prune different arguments */
1687 prune! N A1 N A2 S1 S3 :- !,
1688     new S1 W S2, prune-same-variable W A1 A2 [] Ass,
1689     assign N S2 Ass S3.
1690 /* prune to the intersection of scopes */
1691 prune! N A1 M A2 S1 S4 :- !,
1692     new S1 W S2, prune-diff-variables W A1 A2 Ass1 Ass2,
1693     assign N S2 Ass1 S3,
1694     assign M S3 Ass2 S4.
1695
1696 type prune-same-variable addr -> list tm -> list tm ->
1697     list tm -> assignment -> o.
1698 prune-same-variable N [] [] ACC (val (uva N Args)) :-
1699     rev ACC Args.
1700 prune-same-variable N [X|XS] [X|YS] ACC (abs F) :-
1701     pi x\ prune-same-variable N XS YS [x|ACC] (F x).
1702 prune-same-variable N [_|XS] [_|YS] ACC (abs F) :-
1703     pi x\ prune-same-variable N XS YS ACC (F x).
1704
1705 type permute list nat -> list tm -> list tm -> o.
1706 permute [] _ [].
1707 permute [P|PS] Args [T|TS] :-
1708     nth P Args T,
1709     permute PS Args TS.
1710
1711 type build-perm-assign addr -> list tm -> list bool ->
1712     list nat -> assignment -> o.
1713 build-perm-assign N ArgsR [] Perm (val (uva N PermutedArgs)) :-
1714     rev ArgsR Args, permute Perm Args PermutedArgs.
1715 build-perm-assign N Acc [tt|L] Perm (abs T) :-
1716     pi x\ build-perm-assign N [x|Acc] L Perm (T x).
1717 build-perm-assign N Acc [ff|L] Perm (abs T) :-
1718     pi x\ build-perm-assign N Acc L Perm (T x).
1719
1720 type keep list A -> A -> bool -> o.
1721 keep L A tt :- mem L A, !.
1722 keep _ _ ff.
1723
1724 type prune-diff-variables addr -> list tm -> list tm ->
1725     assignment -> assignment -> o.
1726 prune-diff-variables N Args1 Args2 Ass1 Ass2 :-
1727     map (keep Args2) Args1 Bits1,
1728     map (keep Args1) Args2 Bits2,
1729     filter Args1 (mem Args2) ToKeep1,
1730     filter Args2 (mem Args1) ToKeep2,
1731     map (index ToKeep1) ToKeep1 IdPerm,
1732     map (index ToKeep1) ToKeep2 Perm21,
1733     build-perm-assign N [] Bits1 IdPerm Ass1,
1734     build-perm-assign N [] Bits2 Perm21 Ass2.
1735
1736 type beta tm -> list tm -> tm -> o.
1737 beta A [] A :- !.
1738 beta (lam Bo) [H | L] R :- beta (Bo H) L R1, beta-aux R1 R.
1739 beta (app A) L (app X) :- append A L X.
1740

```

```

1741 beta (con H) L (app [con H | L]).
1742 beta X L (app[X|L]) :- name X.
1743
1744 type beta-aux tm -> tm -> o.
1745 beta-aux (app [HD|TL]) R :- !, beta HD TL R.
1746 beta-aux A A.
1747
1748 /* occur check for N before crossing a functor */
1749 type not_occ addr -> subst -> tm -> o.
1750 not_occ N S (uva M Args) :- set? M S F,
1751   move F Args T, not_occ N S T.
1752 not_occ N S (uva M Args) :- unset? M S, not (M = N),
1753   forall1 (not_occ_aux N S) Args.
1754 not_occ _ _ (con _).
1755 not_occ N S (app L) :- not_occ_aux N S (app L).
1756 /* Note: lam is a functor for the meta language! */
1757 not_occ N S (lam L) :- pi x\ not_occ_aux N S (L x).
1758 not_occ _ _ X :- name X.
1759 /* finding N is ok */
1760 not_occ N _ (uva N _).
1761
1762 /* occur check for X after crossing a functor */
1763 type not_occ_aux addr -> subst -> tm -> o.
1764 not_occ_aux N S (uva M _) :- unset? M S, not (N = M).
1765 not_occ_aux N S (uva M Args) :- set? M S F,
1766   move F Args T, not_occ_aux N S T.
1767 not_occ_aux N S (app L) :- forall1 (not_occ_aux N S) L.
1768 not_occ_aux N S (lam F) :- pi x\ not_occ_aux N S (F x).
1769 not_occ_aux _ _ (con _).
1770 not_occ_aux _ _ X :- name X.
1771 /* finding N is ko, hence no rule */
1772
1773 /* copy T T' fails if T contains a free variable, i.e. it
1774   performs scope checking for bind */
1775 type copy tm -> tm -> o.
1776 copy (con C) (con C).
1777 copy (app L) (app L') :- map copy L L'.
1778 copy (lam T) (lam T') :- pi x\ copy x x => copy (T x) (T' x).
1779 copy (uva A L) (uva A L') :- map copy L L'.
1780
1781 type bind tm -> list tm -> assignment -> o.
1782 bind T [] (val T') :- copy T T'.
1783 bind T [X | TL] (abs T') :- pi x\ copy X x => bind T TL (T' x).
1784
1785 type deref subst -> tm -> tm -> o. (σt)
1786 deref _ (con C) (con C).
1787 deref S (app A) (app B) :- map (deref S) A B.
1788 deref S (lam F) (lam G) :-
1789   pi x\ deref S x x => deref S (F x) (G x).
1790 deref S (uva N L) R :- set? N S A,
1791   move A L T, deref S T R.
1792 deref S (uva N A) (uva N B) :- unset? N S,
1793   map (deref S) A B.
1794
1795 type move assignment -> list tm -> tm -> o.
1796 move (abs Bo) [H|L] R :- move (Bo H) L R.
1797 move (val A) [] A.

```

```

type deref-assmt subst -> assignment -> assignment -> o.

```

```

deref-assmt S (abs T) (abs R) :- pi x\ deref-assmt S (T x) (R x).

```

```

deref-assmt S (val T) (val R) :- deref S T R.

```

14 THE COMPILER

```

kind arity type.

```

```

type arity nat -> arity.

```

```

kind fvariable type.

```

```

type fv addr -> fvariable.

```

```

kind hvariable type.

```

```

type hv addr -> arity -> hvariable.

```

```

kind mapping type.

```

```

type (<=>) 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-beta tm -> tm -> baselink.

```

```

typeabbrev link (inctx baselink).

```

```

typeabbrev links (list link).

```

```

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

```

```

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

```

```

type get-lhs link -> tm -> o.

```

```

get-lhs (val (link-beta A _)) A.

```

```

get-lhs (val (link-eta A _)) A.

```

```

type get-rhs link -> tm -> o.

```

```

get-rhs (val (link-beta _ 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}.

```



```

1857
1858 type maybe-eta fm -> list fm -> o. (◇η)
1859 maybe-eta (fapp [fuva _] Args] L :- !,
1860   forall1 (x\ exists (reducible-to [] x) Args) L, !.
1861 maybe-eta (flam B) L :- !, pi x\ maybe-eta (B x) [x | L].
1862 maybe-eta (fapp [T|Args] L :- (name T; T = fcon _),
1863   split-last-n {len L} Args First Last,
1864   none (x\ exists (y\ occurs-rigidly x y) First) L,
1865   forall2 (reducible-to []) {rev L} Last.
1866
1867
1868 type locally-bound tm -> o.
1869 type get-scope-aux tm -> list tm -> o.
1870 get-scope-aux (con _) [].
1871 get-scope-aux (uva _ L) L1 :-
1872   forall2 get-scope-aux L R,
1873   flatten R L1.
1874 get-scope-aux (lam B) L1 :-
1875   pi x\ locally-bound x => get-scope-aux (B x) L1.
1876 get-scope-aux (app L) L1 :-
1877   forall2 get-scope-aux L R,
1878   flatten R L1.
1879 get-scope-aux X [X] :- name X, not (locally-bound X).
1880 get-scope-aux X [] :- name X, (locally-bound X).
1881
1882 type names1 list tm -> o.
1883 names1 L :-
1884   names L1,
1885   new_int N,
1886   if (1 is N mod 2) (L1 = L) (rev L1 L).
1887
1888 type get-scope tm -> list tm -> o.
1889 get-scope T Scope :-
1890   get-scope-aux T ScopeDuplicata,
1891   undup ScopeDuplicata Scope.
1892 type rigid fm -> o.
1893 rigid X :- not (X = fuva _).
1894
1895 type comp-lam (fm -> fm) -> (tm -> tm) ->
1896   mmap -> mmap -> links -> links -> subst -> subst -> o.
1897 comp-lam F G M1 M2 L1 L3 S1 S2 :-
1898   pi x y\ (pi M L S\ comp x y M M L L S S) =>
1899     comp (F x) (G y) M1 M2 L1 (L2 y) S1 S2,
1900     close-links L2 L3.
1901
1902 type close-links (tm -> links) -> links -> o.
1903 close-links (v\ [X | L v]) [X|R] :- !, close-links L R.
1904 close-links (v\ [X v | L v]) [abs X|R] :- close-links L R.
1905 close-links (_ []) [].
1906 type comp fm -> tm -> mmap -> mmap -> links -> links ->
1907   subst -> subst -> o.
1908 comp (fcon C) (con C) M M L L S S.
1909 comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
1910   maybe-eta (flam F) [], !,
1911   alloc S1 A S2,
1912   comp-lam F F1 M1 M2 L1 L2 S2 S3,
1913   get-scope (lam F1) Scope,
1914
1915   L3 = [val (link-eta (uva A Scope) (lam F1)) | L2].
1916 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :- (cλ)
1917   comp-lam F F1 M1 M2 L1 L2 S1 S2.
1918 comp (fuva A) (uva B []) M1 M2 L L S1 S2 :-
1919   m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
1920 comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
1921   pattern-fragment Ag, !,
1922   fold6 comp Ag Ag1 M1 M1 L L S1 S1,
1923   len Ag Arity,
1924   m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
1925 comp (fapp [fuva A|Ag]) (uva B Scope) M1 M3 L1 L3 S1 S4 :- !,
1926   pattern-fragment-prefix Ag Pf Extra,
1927   len Pf Arity,
1928   alloc S1 B S2,
1929   m-alloc (fv A) (hv C (arity Arity)) M1 M2 S2 S3,
1930   fold6 comp Pf Pf1 M2 M2 L1 L1 S3 S3,
1931   fold6 comp Extra Extra1 M2 M3 L1 L2 S3 S4,
1932   Beta = app [uva C Pf1 | Extra1],
1933   get-scope Beta Scope,
1934   L3 = [val (link-beta (uva B Scope) Beta) | L2].
1935 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :- (c@)
1936   fold6 comp A A1 M1 M2 L1 L2 S1 S2.
1937
1938 type alloc mem A -> addr -> mem A -> o.
1939 alloc S N S1 :- mem.new S N S1.
1940
1941 type compile-terms-diagnostic
1942   triple diagnostic fm fm ->
1943   triple diagnostic tm tm ->
1944   mmap -> mmap ->
1945   links -> links ->
1946   subst -> subst -> o.
1947 compile-terms-diagnostic (triple D F01 F02) (triple D H01 H02) M1 M3 L1
1948   fo.beta-reduce F01 F01',
1949   fo.beta-reduce F02 F02',
1950   comp F01' H01 M1 M2 L1 L2 S1 S2,
1951   comp F02' H02 M2 M3 L2 L3 S2 S3.
1952
1953 type compile-terms
1954   list (triple diagnostic fm fm) ->
1955   list (triple diagnostic tm tm) ->
1956   mmap -> links -> subst -> o.
1957 compile-terms T H M L S :-
1958   fold6 compile-terms-diagnostic T H [] M_ [] L_ [] S_,
1959   print-compil-result T H L_ M_,
1960   deduplicate-map M_ M S_ S L_ L.
1961
1962 type make-eta-link-aux nat -> addr -> addr ->
1963   list tm -> links -> subst -> subst -> o.
1964 make-eta-link-aux z Ad1 Ad2 Scope1 L H1 H1 :-
1965   rev Scope1 Scope, eta-expand (uva Ad2 Scope) T1,
1966   L = [val (link-eta (uva Ad1 Scope) T1)].
1967 make-eta-link-aux (s N) Ad1 Ad2 Scope1 L H1 H3 :-
1968   rev Scope1 Scope, alloc H1 Ad H2,
1969   eta-expand (uva Ad Scope) T2,
1970   (pi x\ make-eta-link-aux N Ad Ad2 [x|Scope1] (L1 x) H2 H3),
1971   close-links L1 L2,
1972

```

```

1973   L = [val (link-eta (uva Ad1 Scope) T2) | L2].
1974
1975   type make-eta-link nat -> nat -> addr -> addr ->
1976     list tm -> links -> subst -> subst -> o.
1977   make-eta-link (s N) z Ad1 Ad2 Vars L H H1 :-
1978     make-eta-link-aux N Ad2 Ad1 Vars L H H1.
1979   make-eta-link z (s N) Ad1 Ad2 Vars L H H1 :-
1980     make-eta-link-aux N Ad1 Ad2 Vars L H H1.
1981   make-eta-link (s N) (s M) Ad1 Ad2 Vars Links H H1 :-
1982     (pi x\ make-eta-link N M Ad1 Ad2 [x]Vars) (L x) H H1),
1983     close-links L Links.
1984
1985   type deduplicate-map mmap -> mmap ->
1986     subst -> subst -> links -> links -> o.
1987   deduplicate-map [] [] H H L L.
1988   deduplicate-map [(fvs 0 <-> hv M (arity LenM)) as X1] | Map1 Map2 progress-beta-link-aux T1 T2 S1 S2 [] :- is-in-pf T2, !,
1989     take-list Map1 ((fvs 0 <-> hv M' (arity LenM'))), !,
1990     std.assert! (not (LenM = LenM')) "Deduplicate map, there is a bug",
1991     print "arity-fix links:" {ppmapping X1} "~!~" {ppmapping ((fvs 0 <-> hv M' (arity LenM')))},
1992     make-eta-link LenM LenM' M M' [] New H1 H2,
1993     print "new eta link" {pplinks New},
1994     append New L1 L2,
1995     deduplicate-map Map1 Map2 H2 H3 L2 L3.
1996   deduplicate-map [A|As] [A|Bs] H1 H2 L1 L2 :-
1997     deduplicate-map As Bs H1 H2 L1 L2, !.
1998   deduplicate-map [A|_] _ H _ _ :-
1999     halt "deduplicating mapping error" {ppmapping A} {ho.ppsubst H}.
2000

```

15 THE PROGRESS FUNCTION

```

2001   macro @one :- s z.
2002
2003   type contract-rigid list ho.tm -> ho.tm -> ho.tm -> o.
2004
2005   contract-rigid L (ho.lam F) T :-
2006     pi x\ contract-rigid [x|L] (F x) T. % also checks H Prefix does not
2007   contract-rigid L (ho.app [H|Args]) T :-
2008     rev L LRev, append Prefix LRev Args,
2009     if (Prefix = []) (T = H) (T = ho.app [H|Prefix]).
2010
2011   type progress-eta-link ho.tm -> ho.tm -> ho.subst -> ho.subst -> link-eta
2012   progress-eta-link (ho.app _ as T) (ho.lam x\ _ as T1) H H1 [] :- !, not (T1 = ho.uva _), !, fail.
2013   ({eta-expand T @one} ==1 T1) H H1.
2014   progress-eta-link (ho.con _ as T) (ho.lam x\ _ as T1) H H1 [] :- !
2015   ({eta-expand T @one} ==1 T1) H H1.
2016   progress-eta-link (ho.lam _ as T) T1 H H1 [] :- !,
2017   (T ==1 T1) H H1.
2018   progress-eta-link (ho.uva _ _ as X) T H H1 [] :-
2019   contract-rigid [] T T1, !, (X ==1 T1) H H1.
2020   progress-eta-link (ho.uva Ad _ as T1) T2 H H [@eval-link-eta T1 T2] :-
2021   if (ho.not_occ Ad H T2) true fail.
2022
2023   type is-in-pf ho.tm -> o.
2024   is-in-pf (ho.app [ho.uva _ _ | _]) :- !, fail.
2025   is-in-pf (ho.lam B) :- !, pi x\ is-in-pf (B x).
2026   is-in-pf (ho.con _).
2027   is-in-pf (ho.app L) :- forall1 is-in-pf L.
2028   is-in-pf N :- name N.
2029

```

```

2030   is-in-pf (ho.uva _ L) :- pattern-fragment L.
2031
2032   type arity ho.tm -> nat -> o.
2033   arity (ho.con _) z.
2034   arity (ho.app L) A :- len L A.
2035
2036   type occur-check-err ho.tm -> ho.tm -> ho.subst -> o.
2037   occur-check-err (ho.con _) _ _ :- !.
2038   occur-check-err (ho.app _) _ _ :- !.
2039   occur-check-err (ho.lam _) _ _ :- !.
2040   occur-check-err (ho.uva Ad _) T S :-
2041     not (ho.not_occ Ad S T).
2042
2043   type progress-beta-link-aux ho.tm -> ho.tm ->
2044     ho.subst -> ho.subst -> links -> o.
2045   progress-beta-link-aux T1 T2 S1 S2 [] :- is-in-pf T2, !,
2046     (T1 ==1 T2) S1 S2.
2047   progress-beta-link-aux T1 T2 S S S [@eval-link-beta T1 T2] :- !.
2048   type progress-beta-link ho.tm -> ho.tm -> ho.subst ->
2049     ho.subst -> links -> o.
2050   progress-beta-link T (ho.app[ho.uva V Scope | L] as T2) S S2 [@eval-link-
2051     arity T Arity, len L ArgsNb, ArgsNb >n Arity, !,
2052     minus ArgsNb Arity Diff, mem.new S V1 S1,
2053     eta-expand (ho.uva V1 Scope) Diff T1,
2054     ((ho.uva V Scope) ==1 T1) S1 S2.
2055   progress-beta-link (ho.uva _ _ as T) (ho.app[ho.uva Ad1 Scope1 | S1] as
2056     append Scope1 L1 Scope1L,
2057     pattern-fragment-prefix Scope1L Scope2 L2,
2058     not (Scope1 = Scope2), !,
2059     mem.new S1 Ad2 S2,
2060     len Scope1 Scope1Len,
2061     len Scope2 Scope2Len,
2062     make-eta-link Scope1Len Scope2Len Ad1 Ad2 [] LinkEta S2 S3,
2063     if (L2 = []) (NewLinks = LinkEta, T2 = ho.uva Ad2 Scope2)
2064     (T2 = ho.app [ho.uva Ad2 Scope2 | L2],
2065     NewLinks = [@eval-link-beta T T2 | LinkEta]).
2066   progress-beta-link T1 (ho.app[ho.uva _ _ | _] as T2) _ _ _ :-
2067     progress-beta-link (ho.uva _ _ as T) (ho.app[ho.uva _ _ | _] as T2) S1
2068     occur-check-err T T2 S1, !, fail.
2069   progress-beta-link T1 (ho.app[ho.uva _ _ | _] as T2) H H [@eval-link-beta
2070   progress-beta-link T1 (ho.app [Hd | T1]) S1 S2 B :-
2071     ho.lam beta Hd T1 T3,
2072     progress-beta-link-aux T1 T3 S1 S2 B.
2073
2074   type solve-link-abs link -> links -> ho.subst -> ho.subst -> o.
2075   solve-link-abs (ho.abs X) R H H1 :-
2076     pi x\ ho.copy x x => (pi S\ ho.deref S x x) =>
2077     solve-link-abs (X x) (R' x) H H1,
2078     close-links R' R.
2079

```

```

2089 solve-link-abs (@val-link-eta A B) NewLinks S S1 :- !,
2090   progress-eta-link A B S S1 NewLinks.
2091
2092 solve-link-abs (@val-link-beta A B) NewLinks S S1 :- !,
2093   progress-beta-link A B S S1 NewLinks.
2094
2095 type take-link link -> links -> link -> links -> o.
2096 take-link A [B|XS] B XS :- link-abs-same-lhs A B, !.
2097 take-link A [L|XS] B [L|YS] :- take-link A XS B YS.
2098
2099 type link-abs-same-lhs link -> link -> o.
2100 link-abs-same-lhs (ho.abs F) B :-
2101   pi x\ link-abs-same-lhs (F x) B.
2102 link-abs-same-lhs A (ho.abs G) :-
2103   pi x\ link-abs-same-lhs A (G x).
2104 link-abs-same-lhs (@val-link-eta (ho.uva N _) _) (@val-link-eta (ho.uva N S2) B) H H1 :-
2105
2106 type same-link-eta link -> link -> ho.subst -> ho.subst -> o.
2107 same-link-eta (ho.abs F) B H H1 :- !, pi x\ same-link-eta (F x) B H H1 -> fm L (ho.uva VM TL) T :- mem L (mapping (fv V0) (hv VM _)),
2108 same-link-eta A (ho.abs G) H H1 :- !, pi x\ same-link-eta A (G x) H H1 -> fm L (ho.uva N S2) B) H H1 :-
2109   same-link-eta (@val-link-eta (ho.uva N S1) A)
2110     (@val-link-eta (ho.uva N S2) B) H H1 :-
2111     std.map2 S1 S2 (x\y\r\ r = ho.copy x y) Perm,
2112     Perm => ho.copy A A',
2113     (A' ==l B) H H1.
2114
2115 type progress1 links -> links -> ho.subst -> ho.subst -> o.
2116 progress1 [] [] X X.
2117 progress1 [A|L1] [A|L3] S S2 :- take-link A L1 B L2, !,
2118   same-link-eta A B S S1,
2119   progress1 L2 L3 S1 S2.
2120 progress1 [L0|L1] L3 S S2 :- deref-link S L0 L,
2121   solve-link-abs L R S S1, !,
2122   progress1 L1 L2 S1 S2, append R L2 L3.
2123
2124
2125
2126 type abs->lam ho.assignment -> ho.tm -> o.
2127 abs->lam (ho.abs T) (ho.lam R) :- !, pi x\ abs->lam (T x) (R x).
2128 abs->lam (ho.val A) A.
2129
2130 type commit-links-aux link -> ho.subst -> ho.subst -> o.
2131 commit-links-aux (@val-link-eta T1 T2) H1 H2 :-
2132   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2133   (T1' ==l T2') H1 H2.
2134 commit-links-aux (@val-link-beta T1 T2) H1 H2 :-
2135   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2136   (T1' ==l T2') H1 H2.
2137 commit-links-aux (ho.abs B) H H1 :-
2138   pi x\ commit-links-aux (B x) H H1.
2139
2140 type commit-links links -> links -> ho.subst -> ho.subst -> o.
2141 commit-links [] [] H H.
2142 commit-links [Abs | Links] L H H2 :-
2143   commit-links-aux Abs H H1, !, commit-links Links L H1 H2.
2144
2145 type decomp1-subst map -> map -> ho.subst ->
2146
2147 fo.fsubst -> fo.fsubst -> o.
2148 decomp1-subst _ [A|_] _ _ _ :- fail.
2149 decomp1-subst _ [] _ F F.
2150 decomp1-subst Map [mapping (fv V0) (hv VM _)|T1] H F F2 :-
2151   mem.set? VM H T, !,
2152   ho.deref-assmt H T TTT,
2153   abs->lam TTT T', tm->fm Map T' T1,
2154   fo.eta-contract T1 T2, mem.assign V0 F T2 F1,
2155   decomp1-subst Map T1 H F1 F2.
2156 decomp1-subst Map [mapping _ (hv VM _)|T1] H F F2 :-
2157   mem.unset? VM H, decomp1-subst Map T1 H F F2.
2158
2159 type tm->fm map -> ho.tm -> fo.fm -> o.
2160 tm->fm _ (ho.con C) (fo.fcon C).
2161 tm->fm L (ho.lam B1) (fo.flam B2) :-
2162   tm->fm L (ho.uva N S2) B) H H1 -> fm L (ho.uva VM TL) T :- mem L (mapping (fv V0) (hv VM _)),
2163   tm->fm L (ho.app L1) T :- map (tm->fm L) L1 [Hd|T1],
2164   fo.mk-app Hd T1 T.
2165 tm->fm L (ho.uva VM TL) T :- mem L (mapping (fv V0) (hv VM _)),
2166   map (tm->fm L) TL T1, fo.mk-app (fo.fuva V0) T1 T.
2167
2168 type add-new-map-aux ho.subst -> list ho.tm -> map ->
2169   map -> fo.fsubst -> fo.fsubst -> o.
2170 add-new-map-aux _ [] _ [] S S.
2171 add-new-map-aux H [T|Ts] L L2 S S2 :-
2172   add-new-map H T L L1 S S1,
2173   add-new-map-aux H Ts L1 L2 S1 S2.
2174
2175 type add-new-map ho.subst -> ho.tm -> map ->
2176   map -> fo.fsubst -> fo.fsubst -> o.
2177 add-new-map _ (ho.uva N _) Map [] F1 F1 :-
2178   mem Map (mapping _ (hv N _)), !.
2179 add-new-map H (ho.uva N L) Map [Map1 | MapL] F1 F3 :-
2180   mem.new F1 M F2,
2181   len L Arity, Map1 = mapping (fv M) (hv N (arity Arity)),
2182   add-new-map H (ho.app L) [Map1 | Map] MapL F2 F3.
2183 add-new-map H (ho.lam B) Map NewMap F1 F2 :-
2184   pi x\ add-new-map H (B x) Map NewMap F1 F2.
2185 add-new-map H (ho.app L) Map NewMap F1 F3 :-
2186   add-new-map-aux H L Map NewMap F1 F3.
2187 add-new-map _ (ho.con _) _ [] F F :- !.
2188 add-new-map _ N _ [] F F :- name N.
2189
2190 type complete-mapping-under-ass ho.subst -> ho.assignment ->
2191   map -> map -> fo.fsubst -> fo.fsubst -> o.
2192 complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
2193   add-new-map H Val Map1 Map2 F1 F2.
2194 complete-mapping-under-ass H (ho.abs Abs) Map1 Map2 F1 F2 :-
2195   pi x\ complete-mapping-under-ass H (Abs x) Map1 Map2 F1 F2.
2196
2197 type complete-mapping ho.subst -> ho.subst ->
2198   map -> map -> fo.fsubst -> fo.fsubst -> o.
2199 complete-mapping _ [] L L F F.
2200 complete-mapping H [none | T1] L1 L2 F1 F2 :-
2201   complete-mapping H T1 L1 L2 F1 F2.
2202 complete-mapping H [some T0 | T1] L1 L3 F1 F3 :-
2203   ho.deref-assmt H T0 T,
2204

```

```

2205     complete-mapping-under-ass H T L1 L2 F1 F2,
2206     append L1 L2 LAll,
2207     complete-mapping H T1 LAll L3 F2 F3.

```

```

2209 type decompile map -> links -> ho.subst ->
2210     fo.fsubst -> fo.fsubst -> o.
2211 decompile Map1 L H0 F0 F02 :-
2212     commit-links L L1_ H0 H01, !,
2213     complete-mapping H01 H01 Map1 Map2 F0 F01,
2214     decomp1-subst Map2 Map2 H01 F01 F02.

```

17 AUXILIARY FUNCTIONS

```

2217 type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A ->
2218     list A1 -> B -> B -> C -> C -> o.
2219 fold4 _ [] [] A A B B.
2220 fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0,
2221     fold4 F XS YS A0 A1 B0 B1.
2223 type len list A -> nat -> o.
2224 len [] z.
2225 len [_|L] (s X) :- len L X.

```