

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, 21 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

Specifying and implementing a logic or a proof system from scratch requires significant effort. Logical Frameworks and Higher Order

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>

Logic Programming Languages provide dedicated, high-level Meta Languages (ML) to facilitate this task in two key ways. The first and most well know one is that variable binding and substitution come for free is one uses the ML binders to represent the object language ones. The second one that comes to mind is unification, a cornerstone for proof construction and proof search. However reusing that brick may require some additional work that we carry out in this paper.

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]. In the notable of Higher Order Logic [11], the ML Isabelle is such a good fit that it implements an interactive proof system for the object logic, and not just a specification. 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. We want to use 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: they back-chain lemmas taken from a designated database of "type class instances". For this reason we believe that λ Prolog, hence 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 unification of the ML Elpi in order to implement the one used by the type-class solver*.

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: ∀n, Finite (fin n). (* r1 *)
Instance nfact_dec: ∀n nf, Decision (nfact n nf). (* r2 *)
Instance forall_dec: ∀A P, Finite A → (* r3 *)
  ∀x:A, Decision (P x) → Decision (∀x:A, P x).
```

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

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

The proof found by the solver back-chains on rule 3 (the only rule about the \forall quantifier), and then solves the premises with rules 1 and 2 respectively. Note that rule 3 features a second order parameter `P` that stands for a function of type `A → Prop` (a predicate over `A`). The solver has to infer a value for `P` by unifying the

conclusion of rule 3 with the goal, and in particular by solving the unification problem $P \times = \text{nfact } x \ 3$. This higher order problem falls in the so called pattern-fragment \mathcal{L}_λ [8] and hence admits a unique solution $P = \lambda x. \text{nfact } x \ 3$.

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 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 $\llbracket x \rrbracket e$, and square brackets denote a list of terms separated by comma. For example the term $\llbracket \forall y:t, \text{nfact } y \ 3 \rrbracket$ is encoded as follows:

```
all (con"t") y\ app[con"nfact", y, con"3"]
```

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

```
finite (app [con"fin", N]).           (r1)
```

```
decision (app [con"nfact", N, NF]).   (r2)
```

```
decision (all A x\ app [P, x]) :- finite A, (r3)
```

```
pi w\ decision (app [P, w]).
```

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

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

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

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

In this paper we study a more sophisticated encoding of 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 $\llbracket \text{link } Pm \ A \ P \rrbracket$ 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 `pi w\`). We show below the premise before and after the instantiation of P :

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

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

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

The root cause of the problems we sketched in this example is that the unification procedure \approx_λ of the meta language is not aware of the equational theory of the object logic, even if both theories include $\eta\beta$ -conversion and admit most general unifiers for unification problems in the pattern-fragment.

Contributions. In this paper we identify a minimal language \mathcal{F}_0 in which the problems sketched in the introduction can be formally described. We detail an encoding of a logic program on \mathcal{F}_0 to a strongly related logic program in \mathcal{H}_0 (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 3 discusses alternative term encodings and related works. section 4 introduces the languages \mathcal{F}_0 and \mathcal{H}_0 , section 5 describes a basic simulation of higher order logic programs. sections 6 and 7 completes its equational theory with support for η -conversion. section 8 deals with the practical necessity of "tolerating" terms outside of the pattern-fragment and discusses how heuristic can be applied. Finally section 9 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

The equational theory of Coq's Dependent Type Theory is very rich. In addition to the usual $\eta\beta$ -equivalence for functions, terms (hence types) are compared up to proposition unfolding and fix-point unrolling. Still, for efficiency and predictability reasons, most form of automatic proof search employ a unification procedure that captures a simpler one, just $\eta\beta$, and that solves higher-order problems restricted to the pattern fragment \mathcal{L}_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. For example:

$x\ f \ x$	$\approx_\lambda \ f$	$P \ x$	$\approx_\lambda \ x$
$\text{lam } A \ x\ \text{app}[con"f", x]$	$\approx_o \ con"f"$	$\text{app}[P, x]$	$\approx_o \ x$
$\text{lam } A \ x\ \text{app}[con"f", x]$	$\neq_\lambda \ con"f"$	$\text{app}[P, x]$	$\neq_\lambda \ x$

To state precisely the problem we solve we need a \mathcal{F}_0 representation of CIC terms and a \mathcal{H}_0 one. We extend the equality over ground

terms to open ones with the reflexivity of unification variables (that is unification variable X is equal to itself, not to another one such as Y). We call \approx_o that equality in \mathcal{F}_o and \approx_λ its counterpart in \mathcal{H}_o . We call \approx_o the unification procedure we want to implement and \approx_λ the one provided by the meta language. We assume that the unification of our meta language is correct and complete in \mathcal{L}_λ .

We write $t_1 \approx_\lambda t_2 \mapsto \sigma$ when t_1 and t_2 unify with substitution σ ; we write σt for the application of the substitution to t , and $\sigma X = \{\sigma t \mid t \in X\}$ when X is a set; we write $\sigma \subseteq \sigma'$ when σ is more general than σ' and we use set union to denote the concatenation of two substitutions. We shall use ρ for \mathcal{F}_o substitutions, and σ for the \mathcal{H}_o ones.

We illustrate a compilation $\langle s \rangle \mapsto (t, m, l)$ that maps a term s in \mathcal{F}_o to a term t in \mathcal{H}_o and produces a variable mapping m and list of links l . The variable map connects unification variables in \mathcal{H}_o with variables in \mathcal{F}_o and is used to “decompile” the assignment, $\langle \sigma, m, l \rangle^{-1} \mapsto \rho$. Links are an accessory piece of information whose description is deferred to section 2.1.

We represent a logic program *run* in \mathcal{F}_o as a list *steps* p of length N . Each step p represents a unification problem between two terms \mathbb{P}_{p_l} and \mathbb{P}_{p_r} , taken from the set of all terms \mathbb{P} . The composition of these steps starting from the empty substitution ρ_0 produces the final substitution ρ_N .¹

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

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

$$\begin{aligned} \text{hstep}(\mathbb{T}, p, \sigma, \mathbb{L}) &\mapsto (\sigma'', \mathbb{L}') \stackrel{\text{def}}{=} \\ &\sigma \mathbb{T}_{p_l} \approx_\lambda \sigma \mathbb{T}_{p_r} \mapsto \sigma' \wedge \text{progress}(\mathbb{L}, \sigma \cup \sigma') \mapsto (\mathbb{L}', \sigma'') \\ \text{hrun}(\mathbb{P}, N) &\mapsto \rho_N \stackrel{\text{def}}{=} \\ &\mathbb{T} \times \mathbb{M} \times \mathbb{L}_0 = \{(t_j, m_j, l_j) \mid s_j \in \mathbb{P}, \langle s_j \rangle \mapsto (t_j, m_j, l_j)\} \\ &\bigwedge_{p=1}^N \text{hstep}(\mathbb{T}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p) \\ &\langle \sigma_N, \mathbb{M}, \mathbb{L}_N \rangle^{-1} \mapsto \rho_N \end{aligned}$$

Here *hstep* is made of two sub-steps: a call to the meta language unification and a check for progress on the set of links. We claim:

PROPOSITION 2.1 (SIMULATION). $\forall \mathbb{P}, \forall N$, if $\mathbb{P} \subseteq \mathcal{L}_\lambda$

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

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 $\forall p \in 1 \dots N$,*

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

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

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

We can define $s_1 \approx_o s_2$ as follows:

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

PROPOSITION 2.3 (PROPERTIES OF \approx_o IN \mathcal{L}_λ).

$$s_i \in \mathcal{L}_\lambda \Rightarrow s_1 \approx_o s_2 \mapsto \rho \Rightarrow \rho s_1 =_o \rho s_2 \quad (1)$$

$$s_i \in \mathcal{L}_\lambda \Rightarrow \rho s_1 =_o \rho s_2 \Rightarrow \exists \rho', s_1 \approx_o s_2 \mapsto \rho' \wedge \rho' \subseteq \rho \quad (2)$$

In addition to correctness and completeness in \mathcal{L}_λ , we claim that *hrun* handles terms outside \mathcal{L}_λ in the following sense:

PROPOSITION 2.4 (PROPERTIES OF *hrun* OUTSIDE \mathcal{L}_λ).

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

$$\sigma_{p-1} \mathbb{T}_p \in \mathcal{L}_\lambda \Rightarrow \text{hstep}(\mathbb{T}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p) \mapsto \sigma \quad (4)$$

Property 3 states that two terms for which there is a unifier (given by an oracle, not necessarily a most general one), then the compiler generates two terms that unify in \mathcal{H}_o . In other words the compiler moves out of the way all problematic terms.

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 do matter. The simplest example is the sequence $F = \lambda x.a$ and $F a = a$: the second problem is not in \mathcal{L}_λ and has two unifiers, namely $F = \lambda x.x$ and $F = \lambda x.a$. The first problem picks one of the two, making the second problem re-enter \mathcal{L}_λ .

2.1 The intuition in a nutshell

A term s is compiled in a term t where every “problematic” sub term p is replaced by a fresh unification variable h and an accessory *link* that represent a suspended unification problem $h \approx_\lambda p$. As a result \approx_λ is “well behaved” on t , that is it does not contradict $=_o$ as it would otherwise do on “problematic” terms. We now define “problematic” and “well behaved” more formally.

Definition 2.5 ($\diamond \eta$). $\diamond \eta = \{t \mid \exists \rho, \rho t \text{ is an eta expansion}\}$

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

Definition 2.6 ($\diamond \beta_0$). $\diamond \beta_0 = \{X \cdot x_1 \dots x_n \mid x_1 \dots x_n \text{ are distinct names}\}$

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 2.7 ($\overline{\mathcal{L}_\lambda}$). $\overline{\mathcal{L}_\lambda} = \{X \cdot t_1 \dots t_n \mid X \cdot t_1 \dots t_n \notin \mathcal{L}_\lambda\}$.

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 2.8 (Well behaved set). Given a set of terms $X \subseteq \mathcal{H}_0$,

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

PROPOSITION 2.9 (\mathcal{W} -PRESERVATION OF hstep). $\forall \mathbb{T}, \forall \mathbb{L}, \forall p, \forall \sigma, \forall \sigma'$

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

A less formal way to state 2.9 is that hstep and progress never “commit” an unneeded λ -abstraction in σ (a λ that could be erased by an η -contraction), nor put in σ an application whose head is flexible (an application node that could be erased by a β -reduction).

Note that proposition 2.9 does not hold for \approx_o as a whole since decompilation can introduce (actually restore) terms in $\diamond \eta, \diamond \beta_0$ or $\overline{\mathcal{L}_\lambda}$ that were moved out of the way during compilation.

3 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 known in advance (not even defined in Coq) without using introspection primitives such as Prolog’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.

4 PRELIMINARIES: \mathcal{F}_0 AND \mathcal{H}_0

In order to reason about unification we provide a description of the \mathcal{F}_0 and \mathcal{H}_0 languages where unification variables are first class terms, i.e. they have a concrete syntax. We keep these languages minimal, for example, we omit the all quantifier of CIC we used in the example in Section 1 together with the type notation of terms carried by the lam constructor.

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

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

Unification variables (fuva term constructor) in \mathcal{F}_0 have no explicit scope: the arguments of an higher order variable are given via the fapp constructor. For example the term $P \ x$ is represented as $fapp[fuva \ N, \ x]$, where N is a memory address and x is a bound variable.

In \mathcal{H}_0 the representation of $P \ x$ is instead $uva \ N \ [x]$, since unification variables come equipped with an explicit scope. We say that the unification variable occurrence $uva \ N \ L$ is in \mathcal{L}_λ if and only if L is made of distinct names. The predicate to test this condition is called pattern-fragment:

```
type pattern-fragment list A -> o.
```

Natural numbers represent the memory addresses that identify unification variables in both languages. The memory and its associated operations are described below:

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

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

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

E:is new used?

Since in \mathcal{H}_0 unification variables have a scope, their solution needs to be abstracted over it to enable the instantiation of a single solution to different scopes. This is obtained via the inctx container, and in particular via its abs binding constructor. On the contrary a solution to a \mathcal{F}_0 variable is a plain term.


```

465 typeabbrev fsubst (mem fm).
466
467 kind inctx type -> type.
468 type abs (tm -> inctx A) -> inctx A.
469 type val A -> inctx A.
470 typeabbrev assignment (inctx tm).
471 typeabbrev subst (mem assignment).

```

We call fsubst the memory of \mathcal{F}_0 , while we call subst the one of \mathcal{H}_0 . Both have the invariant that they are not cyclic, TODO: explain.

```

472
473 kind arity type.
474 type arity nat -> arity.
475
476
477 kind fvariable type.
478 type fv addr -> fvariable.
479
480
481 kind hvariable type.
482 type hv addr -> arity -> hvariable.
483
484 kind mapping type.
485 type mapping fvariable -> hvariable -> mapping.
486 typeabbrev mmap (list mapping).

```

INVARIANT 1 (UNIFICATION VARIABLE ARITY). *Each variable A in \mathcal{H}_0 has a (unique) arity N and each occurrence (uva A L) is such that (len L N) holds*

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

D: add
ref
to
sec-
tion 7

```

490
491 type m-alloc fvariable -> hvariable -> mmap -> mmap ->
492       subst -> subst -> o.
493       (malloc)
494 m-alloc Fv Hv M M S S :- mem M (mapping Fv Hv), !.
495 m-alloc Fv Hv M [mapping Fv Hv|M] S S1 :- Hv = hv N _,
496       alloc S N S1.

```

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

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

```

500
501 kind baselink type.
502 type link-eta tm -> tm -> baselink.
503 type link-beta tm -> tm -> baselink.
504 typeabbrev link (inctx baselink).
505 typeabbrev links (list link).

```

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

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

New links are suspended by construction. If the left hand side variable is assigned during a step, then the link is considered for

progress and possibly eliminated. This is discussed in section 6 and section 8.

4.1 Notational conventions

When we write \mathcal{H}_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:

```

f a      app[con "f", con "a"]
λx.λy.Fxy lam x\ lam y\ uva F [x, y]
λx.Fx a  lam x\ app[uva F [x], con "a"]
λx.Fx x  lam x\ app[uva F [x], x]

```

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

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

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

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

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

4.2 Equational theory and Unification

In order to express properties ?? we need to equip \mathcal{F}_0 and \mathcal{H}_0 with term equality, substitution application and unification.

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

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

```

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

```

```

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

```

The main point in showing these equality tests is to remark how weaker \approx_λ is, and to identify the four rules that need special treatment in the implementation of \approx_o .

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

```

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

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

```

The name predicate holds only on nominal constants (i.e. bound variables).² The choice of using n-ary application, rather than binary, is to make it easy to access the application's head. The price we pay is that substituting an application in the head of an application should be amended by "flattening" fapp nodes, that is the job of napp.³ Finally note that the cut operator is inessential, it could be removed at the cost of a verbose test on the head of L in the second rule about fapp: L 's head can be fcon, flam or a name.

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

```

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

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

```

Applying the substitution in \mathcal{H}_o is very similar, with the caveat that assignments have to be moved to the current scope, i.e. renaming the

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

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

abs-bound variables with the names in the scope of the unification variable occurrence.

```

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

```

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

```

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

```

Term unification: \approx_o vs. \approx_λ . In this paper we assume to have an implementation of \approx_λ that satisfies properties ?? and ??. Although we provide an implementation in the appendix (that we used for testing purposes) we only describe its signature here. Elpi is expected to provide this brick, as well as any other implementation of λ Prolog.

```

type (approx_lambda) tm -> tm -> subst -> subst -> o.

```

The only detail worth discussing is the fact that the procedure updates a substitution, rather than just crafting one as presented in section 2. The reason is that the algorithm folds over a term, updating a substitution while it traverses it.

E:explain better

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

In this section we describe a basic compilation scheme that we refine later, in the following sections. This scheme is sufficient to implement an \approx_o that respects β -conversion for terms in \mathcal{L}_λ . The extension to $\eta\beta$ -conversion is described in Section 6 and the support for terms outside \mathcal{L}_λ in Section 8.

5.1 Compilation

E:manca beta normal in entrata

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

The signature of the comp predicate below allows for the generation of links (suspended unification problems) that play no role in this section but play a major role in section 6 and section 8. With respect to section 2 the signature also allows for updates to the substitution.

```

type comp fm -> tm -> mmap -> mmap -> links -> links ->
  subst -> subst -> o.
comp (fcon C) (con C) M M L L S S.

```

```

697 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :- (cλ)
698   comp-lam F F1 M1 M2 L1 L2 S1 S2.
699 comp (fuva A) (uva B [J]) M1 M2 L L S1 S2 :-
700   m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
701 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :- (c@)
702   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).

```

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

```

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

The auxiliary function `close-links` tests if the bound variable `v` really occurs in the link. If it is the case the link is wrapped into an additional `abs` node binding `v`. In this way links generated deep inside the compiled terms can be moved outside their original context of binders.

```

720 type close-links (tm -> links) -> links -> o.
721 close-links (v\ [X L v]) [X|R] :- !, close-links L R.
722 close-links (v\ [X v | L v]) [abs X|R] :- close-links L R.
723 close-links (_\ [J]) [J].

```

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

5.1.1 Detection of $\diamond\beta_0$. The following rule is inserted just before rule (c@).

```

730 comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
731   pattern-fragment Ag, !,
732   fold6 comp Ag Ag1 M1 M1 L L S1 S1,
733   len Ag Arity,
734   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.

5.2 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.

```

745 type hstep tm -> tm -> links -> links -> subst -> subst -> o.
746 hstep T1 T2 L1 L2 S1 S3 :-
747   (T1 ≈λ T2) S1 S2,
748   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.

```

755 type progress links -> links -> subst -> subst -> o.
756 progress L L2 S1 S3 :-
757   progress1 L L1 S1 S2,
758   occur-check-links L1,
759   if (L = L1, S1 = S2)
760     (L2 = L1, S3 = S1)
761     (progress L1 L2 S2 S3).

```

In the base compilation scheme `progress1` is the identity on both the links and the substitution, so the fixpoint trivially terminates. Sections 6 and 8 add rules to `progress1` and justify why the don't hinder termination. For brevity we omit the code that applies the substitution `S1` to all terms in `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 `L` contains a link like $\vdash Y =_\eta \lambda z.X_z$: We don't know yet if `Y` will feature a lambda in head position, but we surely know it contains `X`, hence $f Y$ and that fails the occur check. The procedure `occur-check-links` is in charge of ensuring that each link does not represent a (suspended) unification problem doomed to fail because of occur check. This check is needed in order to guarantee proposition 2.2 (simulation fidelity).

5.3 Substitution decompilation

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

```

785 type decompile mmap -> links -> subst ->
786   fsubst -> fsubst -> o.
787 decompile M1 L S F1 F3 :-
788   commit-links L S S1,
789   complete-mapping S1 S1 M1 M2 F1 F2,
790   decomp M2 M2 S1 F2 F3.

```

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

```

797 type decomp mmap -> mmap -> subst -> fsubst -> fsubst -> o.
798 decomp _ [J] _ F F.
799 decomp M [mapping (fv V) (hv H _)] [MS] S F1 F3 :- set? H S A,
800   deref-assmt S A A1,
801   abs->lam A1 T, decomp M T T1,
802   eta-contract T1 T2,
803   assign V F1 T2 F2,
804   decomp M MS S F2 F3.
805 decomp M [mapping _ (hv H _)] [MS] S F1 F2 :- unset? H S,
806   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 introduced.

```

808 type decomp mmap -> tm -> fm -> o.
809 decomp _ (con C) (fcon C).
810 decomp M (app A) (fapp B) :- map (decomp M) A B.
811 decomp M (lam F) (flam G) :-

```

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

```

813   pi x y\ (pi M\ decomp M x y) => decomp M (F x) (G y).
814   decomp M (uva Hv Ag) R :-
815     mem M (mapping (fv Fv) (hv Hv _)),
816     map (decomp M) Ag Bg,
817     beta (fuva Fv) Bg R.

```

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

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

5.4 Definition of \approx_o and its properties

```

825   type ( $\approx_o$ ) fm -> fm -> fsubst -> o.
826   (A  $\approx_o$  B) F :-
827     fo.beta-reduce A A',
828     fo.beta-reduce B B',
829     comp A' A'' [] M1 [] [] S1,
830     comp B' B'' M1 M2 [] [] S1 S2,
831     hstep A'' B'' [] [] S2 S3,
832     decomp M2 M2 S3 [] F.

```

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

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

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

LEMMA 5.2. *Properties 1 and 2 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}_o terms can be made equal by a substitution ρ (plus the β_l and β_r if needed) we can find this ρ by finding a σ via \approx_λ on the corresponding \mathcal{H}_o terms and by decompiling it. If we look at the \mathcal{F}_o terms, there 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 $\sigma = \{\vec{y} \mapsto Y \mapsto t[\vec{x}/\vec{y}]\}$ that in turn is decompiled to $\rho = \{Y \mapsto \lambda \vec{y}.s[\vec{x}/\vec{y}]\}$. Thanks to β_l ($\lambda \vec{y}.s[\vec{x}/\vec{y}]\vec{x} =_o s$).

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

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

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

5.5 Limitations of by this basic scheme

The basic compilation scheme is not about to deal with 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.

6 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 5.1 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 6.1) and we modify the compiled terms by putting fresh unification variables in their place: the problematic term is moved from \mathbb{T} to \mathbb{L} (section 6.2). The compilation of the problem \mathbb{P} above is refined to:

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

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 5.2).

6.1 Detection of $\diamond\eta$

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

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

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 6.1 (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 6.2. 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 6.3 (occurs-rigidly). A name x occurs-rigidly in a β -normal term t , if $\forall \rho, x \in \mathcal{P}(\rho t)$

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

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

Definition 6.4 (maybe-eta). Given a β -normal term $s = \lambda x_1 \dots x_n.t$, 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 6.5 ($\Diamond\eta$ DETECTION). If t is a β -normal term and maybe-eta t holds, then $t \in \Diamond\eta$.

PROOF SKETCH. Follows from definition 6.3 and lemma 6.2 \square

Remark that the converse of lemma 6.5 does not hold: there exists a term t satisfying the criteria (1) of definition 6.4 that is not in $\Diamond\eta$, i.e. there exists no substitution ρ such that ρt is an η -expansion. A

simple counter example is $\lambda x.f(A\ x)(A\ x)$ since x does not occur-rigidly in the first argument of f , and the second argument of f may-contract-to x . In other words $A\ x$ may either use or discard x , but our analysis does not take into account that the same term cannot have two contrasting behaviors.

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

6.2 Compilation and decompilation

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

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

The rule triggers when the input term `flam F` is in $\Diamond\eta$. It compiles `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 6.6. 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 η -link is performed by adding new rules to the `commit-link` predicate. In particular, given $\Gamma \vdash X =_\eta t$, we can note that this unification never fails, since X is a flexible term and no other η -link has X as lhs (by definition 6.9). The link is removed from \mathbb{L} and `commit-links` terminates.

6.3 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 6.7 (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 6.8 (progress- η -right). A link $\Gamma \vdash X =_\eta T$ is removed from \mathbb{L} when either 1) maybe-eta T does not hold (anymore) or 2) by η -contracting T to T' , T' is a term not starting with the `lam` constructor. In the first case, X is unified with T and in the second one, X is unified with T' (under the context Γ).

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

D:Not too clear?

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

D:Below the proof of proposition 2.9, ho usato 3 lemmi ausiliari, forse si può compattare in una prova più piccola?

LEMMA 6.10. *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, such that $\mathcal{W}(\sigma)$. lhs $\in \sigma$, therefore $\mathcal{W}(\text{lhs})$. By definition 6.7, if 1) lhs is a name, a constant of an application, then, lhs is unified with the η -reduced term t obtain from rhs. By corollary 6.6, rhs has one lambda, therefore $\mathcal{W}(t)$. Otherwise, 2) lhs has lam as functor, rhs should not be an η -expansion ans, so, $\mathcal{W}(\text{rhs})$. In both cases, unification is performed between terms in \mathcal{W} . \square

LEMMA 6.11. *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 6.8, rhs is either no more a $\diamond\eta$, i.e. rhs is not a η -expansion and, so, $\mathcal{W}(\text{rhs})$. Otherwise, rhs can reduce to a term which cannot be a η -expansion, and, so, $\mathcal{W}(\text{rhs})$. In both cases, unification is done between terms in \mathcal{W} . \square

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

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

LEMMA 6.13. *Proposition 2.9 holds, i.e., given a substitution σ and a η -link l , after the activation of l , $\mathcal{W}(\sigma)$ holds.*

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

D:Bisogna aggiungere un lemma nella section 2.1 che dice che unificare due termini in \mathcal{W} , in una σ , tale che $\mathcal{W}(\sigma)$, non invalida \mathcal{W}

LEMMA 6.14. *progress terminates.*

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

D:Proove simulation fidelity, dicendo che *progress- η -right* è inutile

Example of progress- η -left. The example at the beginning of section 6, once $\sigma = \{ A \mapsto f \}$, triggers this rule since the link becomes $\vdash f =_{\eta} \lambda x. B_x$ and the lhs is a constant. In turn the rule runs $\lambda x. f x \approx_{\lambda} \lambda x. B_x$, resulting in $\sigma = \{ A \mapsto f ; B_x \mapsto f \}$. Decompile 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:

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

The result of $A \approx_{\lambda} C$ 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*

7 ENFORCING INVARIANT 1

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

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

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

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

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 7.2 (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 6.7. C_{yx} is therefore assigned to x (the second variable of its scope). We can finally see the *progress- η -right* of \mathbb{L}_1 : its rhs is now $\lambda y. y$ (C_{xy} gives y). Since it is no more in $\Diamond\eta$, $\lambda y. y$ is unified with E_x . Moreover, \mathbb{L}_2 is also triggered due to definition 6.8: $\lambda x. (f \ (\lambda y. y) \ x)$ is η -reducible to $f \ (\lambda y. y)$ which is a term not starting with the lam constructor.

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

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

Until now, we have only dealt we unification of terms in \mathcal{L}_λ . However, we want the unification relation to be more robust so that it can work with terms in $\overline{\mathcal{L}_\lambda}$. In general, unification in $\overline{\mathcal{L}_\lambda}$ admits more then one solution and committing one of them in the substitution does not guarantee prop. 2. For instance, $X \ a \approx_o a$ is a unification problem admits two different substitutions: $\rho_1 = \{X \mapsto \lambda x. x\}$ and $\rho_2 = \{X \mapsto \lambda_. a\}$. Prefer one over the other may break future unifications.

It is the case that, given a list of unification problems, $\mathbb{P}_1 \dots \mathbb{P}_n$ with \mathbb{P}_n in $\overline{\mathcal{L}_\lambda}$, 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 .

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

D:i don't understand the note

On the other hand, we see that, \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 $(\lambda x. B) \ a \approx_\lambda a$.

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

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

D:Is it clearer?

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 $L = t_1 \dots t_m$.*

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

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

8.1 Compilation and decompilation

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

```
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 $\text{append Pf Extra Ag}$ and Pf is the largest prefix of Ag 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 8.2. *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 5.1) and no β -link is produced which contradicts our initial assumption. \square

COROLLARY 8.3. *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 $\text{fapp}[\text{fuva } A|\text{Ag}]$. $s_1 \dots s_n$ 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

8.2 Progress

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

Definition 8.4 (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 8.5 (progress-beta-rigid-head). 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 8.6 (progress-beta-dedup). Given a β -link l_1 and second link $l_2 \in \mathbb{L}$, such that they share the same lhs. In this case, the two rhs are unified and a l_2 is removed from \mathbb{L} .

LEMMA 8.7. *progress terminates*

PROOF SKETCH. Let l a β -link in the store \mathbb{L} . If l is activated by *progress-beta-rigid-head*, then it disappears from \mathbb{L} and progress terminates. Otherwise, the rhs of l is made by a variable applied to m arguments. At each activation of *progress-beta- \mathcal{L}_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 6.14, the algorithm terminates. Finally *progress-beta-dedup* also guarantees termination since it makes a unification \mathbb{U} and if \mathbb{U} fails, then progress terminates and if \mathbb{U} succeeds, the recursive calls to progress have a specialized β -link ...

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).

□

COROLLARY 8.8. *Given a β -link, the variables occurring in its rhs are in \mathcal{L}_λ .*

D: is it clearer?

PROOF SKETCH. By construction, the rhs of β -link has the shape $X_{s_1 \dots s_n} t_1 \dots t_m$, $s_1 \dots s_n$ is in \mathcal{L}_λ and all the terms $t_1 \dots t_n$ are in \mathcal{L}_λ , too. If a β -link is triggered by *progress-beta-rigid-head*, then, by definition 8.5, that link is removed by \mathbb{L} , and the property is satisfied. If the η -link is activated by *progress-beta- \mathcal{L}_λ* , then, by

definition 8.4, the new β -link as a variable as a scope which is still in \mathcal{L}_λ . □

LEMMA 8.9. *Given a \mathbb{T} and a substitution σ then the resolution of $\sigma \mathbb{T}$ guarantees proposition 2.2*

PROOF SKETCH. If $\sigma \mathbb{T}$ is in \mathcal{L}_λ , then by definitions 8.4 and 8.5, then β -link disappear and the unification done between terms in \mathcal{L}_λ . This problem unifies iff its corresponding \mathcal{F}_0 problem unifies too. If $\sigma \mathbb{T}$ is in $\overline{\mathcal{L}_\lambda}$, then, by lemma 8.1, the unification fails, as per the corresponding unification in \mathcal{F}_0 . □

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

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

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-head. We can take the example provided in section 8. The problem is compiled into:

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

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 $\rho = \{X \mapsto \lambda x.a, Y \mapsto a\}$.

8.3 Relaxing lemma β -link with rigid lhs

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 $\overline{\mathcal{L}_\lambda}$ unification problems if the substitution is not able to put them in \mathcal{L}_λ .

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

In the example above, \mathbb{P}_1 is in $\overline{\mathcal{L}_\lambda}$ and the object language cannot solve it, and, by proposition 2.2, the meta language neither. However, we can be more permissive, and relax lemma 8.1. This modification is quite simple to manage: we are introducing a new $\overline{\mathcal{L}_\lambda}$ progress rule, say *progress-beta- $\overline{\mathcal{L}_\lambda}$* , by which, if lhs is rigid and rhs is flexible, the considered β -link is kept in the store and no progression is

done⁴. $\text{progress-beta-}\overline{\mathcal{L}}_\lambda$ makes occur-check-links partial, since the check is possible only on links with a variable on the lhs. This means that we can have two links $\vdash X =_\beta Y a$ 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 $\overline{\mathcal{L}}_\lambda$ is to make some approximation. This is the case for example of the unification algorithm of Coq used in its type class solver [16]. The approximation consists in forcing a choice (among the others) when the unification problem is in $\overline{\mathcal{L}}_\lambda$. For instance, in $X a b = Y b$, the last argument of the two terms is the same, therefore Y is assigned to $X a$. Note that this is of course an approximation, since $\sigma = \{X = \lambda x.Y, Y = _ \}$ is another valid substitution for the original problem. This approximation can be easily introduced in our unification procedure, by adding new custom β -link progress rules.

The commit-link predicate can be extended to add heuristics if during the decompilation phase β -link remain. For example, the same approximation explained above can be delayed and applied only if the terms in $\diamond\beta_0$ never falls in \mathcal{L}_λ after the execution of all the unification problems. We want to point out, that we call this *approximation*, since we are making a choice among all the possible unifiers and therefore, we can pick the wrong one.

9 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`.

⁴This new rule trivially guarantees the termination of progress

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 sub-system 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 Schürmann. “System Description: Twelf — A Meta-Logical Framework for Deductive Systems”. In: *Automated Deduction — CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2.
- [16] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-71067-7.
- [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, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7.

APPENDIX

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

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

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

11 THE MEMORY

```

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

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

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

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

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

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

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

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

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

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

```

12 THE OBJECT LANGUAGE

```

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

typeabbrev fsubst (mem fm).

```

```

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

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

```

```

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

```

```

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

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

```

```

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

```

```

type 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 :- !,

```

```

1741   beta (flam B) L T1, beta-reduce T1 T2.
1742   beta-reduce (fapp L) (fapp L1) :-
1743     map beta-reduce L L1.
1744
1745   type mk-app fm -> list fm -> fm -> o.
1746   mk-app T L S :- beta T L S.
1747
1748   type eta-contract fm -> fm -> o.
1749   eta-contract (fcon X) (fcon X).
1750   eta-contract (fapp L) (fapp L1) :- map eta-contract L L1.
1751   eta-contract (flam F) T :- eta-contract-aux [] (flam F) T.
1752   eta-contract (flam F) (flam F1) :-
1753     pi x\ eta-contract x x => eta-contract (F x) (F1 x).
1754   eta-contract (fuva X) (fuva X).
1755   eta-contract X X :- name X.
1756
1757   type eta-contract-aux list fm -> fm -> fm -> o.
1758   eta-contract-aux L (flam F) T :-
1759     pi x\ eta-contract-aux [x|L] (F x) T. % also checks H Prefix does not x
1760   eta-contract-aux L (fapp [H|Args]) T :-
1761     rev L LRev, append Prefix LRev Args,
1762     if (Prefix = []) (T = H) (T = fapp [H|Prefix]).
1763
1764
1765 13 THE META LANGUAGE
1766   kind inctx type -> type.
1767   type abs (tm -> inctx A) -> inctx A.
1768   type val A -> inctx A.
1769   typeabbrev assignment (inctx tm).
1770   typeabbrev subst (mem assignment).
1771
1772   kind tm type.
1773   type app list tm -> tm.
1774   type lam (tm -> tm) -> tm.
1775   type con string -> tm.
1776   type uva addr -> list tm -> tm.
1777
1778   type ( $\approx_\lambda$ ) tm -> tm -> subst -> subst -> o.
1779   (con C  $\approx_\lambda$  con C) S S.
1780   (app L1  $\approx_\lambda$  app L2) S S1 :- fold2 ( $\approx_\lambda$ ) L1 L2 S S1.
1781   (lam F1  $\approx_\lambda$  lam F2) S S1 :-
1782     pi x\ (pi S\ (x  $\approx_\lambda$  x) S S) => (F1 x  $\approx_\lambda$  F2 x) S S1.
1783   (uva N Args  $\approx_\lambda$  T) S S1 :-
1784     set? N S F,!, move F Args T1, (T1  $\approx_\lambda$  T) S S1.
1785   (T  $\approx_\lambda$  uva N Args) S S1 :-
1786     set? N S F,!, move F Args T1, (T  $\approx_\lambda$  T1) S S1.
1787   (uva M A1  $\approx_\lambda$  uva N A2) S1 S2 :- !,
1788     pattern-fragment A1, pattern-fragment A2,
1789     prune! M A1 N A2 S1 S2.
1790   (uva N Args  $\approx_\lambda$  T) S S1 :- not_occ N S T, pattern-fragment Args,
1791     bind T Args T1, assign N S T1 S1.
1792   (T  $\approx_\lambda$  uva N Args) S S1 :- not_occ N S T, pattern-fragment Args,
1793     bind T Args T1, assign N S T1 S1.
1794
1795   type prune! addr -> list tm -> addr ->
1796     list tm -> subst -> subst -> o.
1797   /* no pruning needed */
1798

```

```

1799   prune! N A N A S S :- !.
1800   prune! M A N A S1 S2 :- !, bind (uva M A) A Ass,
1801     assign N S1 Ass S2.
1802   /* prune different arguments */
1803   prune! N A1 N A2 S1 S3 :- !,
1804     new S1 W S2, prune-same-variable W A1 A2 [] Ass,
1805     assign N S2 Ass S3.
1806   /* prune to the intersection of scopes */
1807   prune! N A1 M A2 S1 S4 :- !,
1808     new S1 W S2, prune-diff-variables W A1 A2 Ass1 Ass2,
1809     assign N S2 Ass1 S3,
1810     assign M S3 Ass2 S4.
1811
1812   type prune-same-variable addr -> list tm -> list tm ->
1813     list tm -> assignment -> o.
1814   prune-same-variable N [] [] ACC (val (uva N Args)) :-
1815     rev ACC Args.
1816   prune-same-variable N [X|XS] [X|YS] ACC (abs F) :-
1817     pi x\ prune-same-variable N XS YS [x|ACC] (F x).
1818   prune-same-variable N [_|XS] [_|YS] ACC (abs F) :-
1819     pi x\ prune-same-variable N XS YS ACC (F x).
1820
1821   type permute list nat -> list tm -> list tm -> o.
1822   permute [] _ [].
1823   permute [P|PS] Args [T|TS] :-
1824     nth P Args T,
1825     permute PS Args TS.
1826
1827   type build-perm-assign addr -> list tm -> list bool ->
1828     list nat -> assignment -> o.
1829   build-perm-assign N ArgsR [] Perm (val (uva N PermutedArgs)) :-
1830     rev ArgsR Args, permute Perm Args PermutedArgs.
1831   build-perm-assign N Acc [tt|L] Perm (abs T) :-
1832     pi x\ build-perm-assign N [x|Acc] L Perm (T x).
1833   build-perm-assign N Acc [ff|L] Perm (abs T) :-
1834     pi x\ build-perm-assign N Acc L Perm (T x).
1835
1836   type keep list A -> A -> bool -> o.
1837   keep L A tt :- mem L A, !.
1838   keep _ _ ff.
1839
1840   type prune-diff-variables addr -> list tm -> list tm ->
1841     assignment -> assignment -> o.
1842   prune-diff-variables N Args1 Args2 Ass1 Ass2 :-
1843     map (keep Args2) Args1 Bits1,
1844     map (keep Args1) Args2 Bits2,
1845     filter Args1 (mem Args2) ToKeep1,
1846     filter Args2 (mem Args1) ToKeep2,
1847     map (index ToKeep1) ToKeep1 IdPerm,
1848     map (index ToKeep1) ToKeep2 Perm21,
1849     build-perm-assign N [] Bits1 IdPerm Ass1,
1850     build-perm-assign N [] Bits2 Perm21 Ass2.
1851
1852   type beta tm -> list tm -> tm -> o.
1853   beta A [] A :- !.
1854   beta (lam Bo) [H | L] R :- beta (Bo H) L R1, beta-aux R1 R.
1855   beta (app A) L (app X) :- append A L X.
1856

```



```

1857 beta (con H) L (app [con H | L]).
1858 beta X L (app[X|L]) :- name X.
1859
1860 type beta-aux tm -> tm -> o.
1861 beta-aux (app [HD|TL]) R :- !, beta HD TL R.
1862 beta-aux A A.
1863
1864 /* occur check for N before crossing a functor */
1865 type not_occ addr -> subst -> tm -> o.
1866 not_occ N S (uva M Args) :- set? M S F,
1867   move F Args T, not_occ N S T.
1868 not_occ N S (uva M Args) :- unset? M S, not (M = N),
1869   forall1 (not_occ_aux N S) Args.
1870 not_occ _ _ (con _).
1871 not_occ N S (app L) :- not_occ_aux N S (app L).
1872 /* Note: lam is a functor for the meta language! */
1873 not_occ N S (lam L) :- pi x\ not_occ_aux N S (L x).
1874 not_occ _ _ X :- name X.
1875 /* finding N is ok */
1876 not_occ N _ (uva N _).
1877
1878 /* occur check for X after crossing a functor */
1879 type not_occ_aux addr -> subst -> tm -> o.
1880 not_occ_aux N S (uva M _) :- unset? M S, not (N = M).
1881 not_occ_aux N S (uva M Args) :- set? M S F,
1882   move F Args T, not_occ_aux N S T.
1883 not_occ_aux N S (app L) :- forall1 (not_occ_aux N S) L.
1884 not_occ_aux N S (lam F) :- pi x\ not_occ_aux N S (F x).
1885 not_occ_aux _ _ (con _).
1886 not_occ_aux _ _ X :- name X.
1887 /* finding N is ko, hence no rule */
1888
1889 /* copy T T' fails if T contains a free variable, i.e. it
1890   performs scope checking for bind */
1891 type copy tm -> tm -> o.
1892 copy (con C) (con C).
1893 copy (app L) (app L') :- map copy L L'.
1894 copy (lam T) (lam T') :- pi x\ copy x x => copy (T x) (T' x).
1895 copy (uva A L) (uva A L') :- map copy L L'.
1896
1897 type bind tm -> list tm -> assignment -> o.
1898 bind T [] (val T') :- copy T T'.
1899 bind T [X | TL] (abs T') :- pi x\ copy X x => bind T TL (T' x).
1900
1901 type deref subst -> tm -> tm -> o. (σt)
1902 deref _ (con C) (con C).
1903 deref S (app A) (app B) :- map (deref S) A B.
1904 deref S (lam F) (lam G) :-
1905   pi x\ deref S x x => deref S (F x) (G x).
1906 deref S (uva N L) R :- set? N S A,
1907   move A L T, deref S T R.
1908 deref S (uva N A) (uva N B) :- unset? N S,
1909   map (deref S) A B.
1910
1911 type move assignment -> list tm -> tm -> o.
1912 move (abs Bo) [H|L] R :- move (Bo H) L R.
1913 move (val A) [] A.

```

```

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

```

```

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

```

```

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

```

14 THE COMPILER

```

kind arity type.

```

```

type arity nat -> arity.

```

```

kind fvariable type.

```

```

type fv addr -> fvariable.

```

```

kind hvariable type.

```

```

type hv addr -> arity -> hvariable.

```

```

kind mapping type.

```

```

type mapping fvariable -> hvariable -> mapping.

```

```

typeabbrev 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 occurs-rigidly fm -> fm -> o.

```

```

occurs-rigidly N N.

```

```

occurs-rigidly _ (fapp [fuva _|Args]) :- !, fail.

```

```

occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.

```

```

occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).

```

```

type reducible-to list fm -> fm -> fm -> o.

```

```

reducible-to _ N N :- !.

```

```

reducible-to L N (fapp [fuva _|Args]) :- !,
  forall1 (x\ exists (reducible-to [] x) Args) [N|L].

```

```

reducible-to L N (flam B) :- !,

```

```

  pi x\ reducible-to [x | L] N (B x).

```

```

reducible-to L N (fapp [N|Args]) :-

```

```

  last-n {len L} Args R,

```

```

  forall2 (reducible-to []) R {rev L}.

```

```

type maybe-eta fm -> list fm -> o. (◇η)

```

```

maybe-eta (fapp [fuva _|Args]) L :- !,

```

```

  forall1 (x\ exists (reducible-to [] x) Args) L, !.

```

```

maybe-eta (flam B) L :- !, pi x\ maybe-eta (B x) [x | L].

```

```

maybe-eta (fapp [T|Args]) L :- (name T; T = fcon _),

```

```

  split-last-n {len L} Args First Last,

```

```

  none (x\ exists (y\ occurs-rigidly x y) First) L,

```

```

1973 forall2 (reducible-to []) {rev L} Last.
1974
1975
1976 type locally-bound tm -> o.
1977 type get-scope-aux tm -> list tm -> o.
1978 get-scope-aux (con _) [].
1979 get-scope-aux (uva _ L) L1 :-
1980   forall2 get-scope-aux L R,
1981   flatten R L1.
1982 get-scope-aux (lam B) L1 :-
1983   pi x\ locally-bound x => get-scope-aux (B x) L1.
1984 get-scope-aux (app L) L1 :-
1985   forall2 get-scope-aux L R,
1986   flatten R L1.
1987 get-scope-aux X [X] :- name X, not (locally-bound X).
1988 get-scope-aux X [] :- name X, (locally-bound X).
1989
1990 type names1 list tm -> o.
1991 names1 L :-
1992   names L1,
1993   new_int N,
1994   if (1 is N mod 2) (L1 = L) (rev L1 L).
1995
1996 type get-scope tm -> list tm -> o.
1997 get-scope T Scope :-
1998   get-scope-aux T ScopeDuplicata,
1999   undup ScopeDuplicata Scope.
2000 type rigid fm -> o.
2001 rigid X :- not (X = fuva _).
2002
2003 type comp-lam (fm -> fm) -> (tm -> tm) ->
2004   mmap -> mmap -> links -> links -> subst -> subst -> o.
2005 comp-lam F G M1 M2 L1 L3 S1 S2 :-
2006   pi x y\ (pi M L S\ comp x y M M L L S S) =>
2007     comp (F x) (G y) M1 M2 L1 (L2 y) S1 S2,
2008     close-links L2 L3.
2009
2010 type close-links (tm -> links) -> links -> o.
2011 close-links (v\[X |L v]) [X|R] :- !, close-links L R.
2012 close-links (v\[X v|L v]) [abs X|R] :- close-links L R.
2013 close-links (_\[ ]) [].
2014 type comp fm -> tm -> mmap -> mmap -> links -> links ->
2015   subst -> subst -> o.
2016 comp (fcon C) (con C) M M L L S S.
2017 comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
2018   maybe-eta (flam F) [], !,
2019   alloc S1 A S2,
2020   comp-lam F F1 M1 M2 L1 L2 S2 S3,
2021   get-scope (lam F1) Scope,
2022   L3 = [val (link-eta (uva A Scope) (lam F1)) | L2].
2023 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :- (cλ)
2024   comp-lam F F1 M1 M2 L1 L2 S1 S2.
2025 comp (fuva A) (uva B [ ]) M1 M2 L L S1 S2 :-
2026   m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
2027 comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
2028   pattern-fragment Ag, !,
2029   fold6 comp Ag Ag1 M1 M1 L L S1 S1,
2030   len Ag Arity,
2031   m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
2032 comp (fapp [fuva A|Ag]) (uva B Scope) M1 M3 L1 L3 S1 S4 :- !,
2033   pattern-fragment-prefix Ag Pf Extra,
2034   len Pf Arity,
2035   alloc S1 B S2,
2036   m-alloc (fv A) (hv C (arity Arity)) M1 M2 S2 S3,
2037   fold6 comp Pf Pf1 M2 M2 L1 L1 S3 S3,
2038   fold6 comp Extra Extra1 M2 M3 L1 L2 S3 S4,
2039   Beta = app [uva C Pf1 | Extra1],
2040   get-scope Beta Scope,
2041   L3 = [val (link-beta (uva B Scope) Beta) | L2].
2042 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :- (c@)
2043   fold6 comp A A1 M1 M2 L1 L2 S1 S2.
2044
2045 type alloc mem A -> addr -> mem A -> o.
2046 alloc S N S1 :- mem.new S N S1.
2047
2048 type compile-terms-diagnostic
2049   triple diagnostic fm fm ->
2050   triple diagnostic tm tm ->
2051   mmap -> mmap ->
2052   links -> links ->
2053   subst -> subst -> o.
2054 compile-terms-diagnostic (triple D F01 F02) (triple D H01 H02) M1 M3 L1
2055   fo.beta-reduce F01 F01',
2056   fo.beta-reduce F02 F02',
2057   comp F01' H01 M1 M2 L1 L2 S1 S2,
2058   comp F02' H02 M2 M3 L2 L3 S2 S3.
2059
2060 type compile-terms
2061   list (triple diagnostic fm fm) ->
2062   list (triple diagnostic tm tm) ->
2063   mmap -> links -> subst -> o.
2064 compile-terms T H M L S :-
2065   fold6 compile-terms-diagnostic T H [ ] M_ [ ] L_ [ ] S_,
2066   print-compil-result T H L_ M_,
2067   deduplicate-map M_ M S_ S L_ L.
2068
2069 type make-eta-link-aux nat -> addr -> addr ->
2070   list tm -> links -> subst -> subst -> o.
2071 make-eta-link-aux z Ad1 Ad2 Scope1 L H1 H1 :-
2072   rev Scope1 Scope, eta-expand (uva Ad2 Scope) T1,
2073   L = [val (link-eta (uva Ad1 Scope) T1)].
2074 make-eta-link-aux (s N) Ad1 Ad2 Scope1 L H1 H3 :-
2075   rev Scope1 Scope, alloc H1 Ad H2,
2076   eta-expand (uva Ad Scope) T2,
2077   (pi x\ make-eta-link-aux N Ad Ad2 [x|Scope1] (L1 x) H2 H3),
2078   close-links L1 L2,
2079   L = [val (link-eta (uva Ad1 Scope) T2) | L2].
2080
2081 type make-eta-link nat -> nat -> addr -> addr ->
2082   list tm -> links -> subst -> subst -> o.
2083 make-eta-link (s N) z Ad1 Ad2 Vars L H H1 :-
2084   make-eta-link-aux N Ad2 Ad1 Vars L H H1.
2085 make-eta-link z (s N) Ad1 Ad2 Vars L H H1 :-
2086   make-eta-link-aux N Ad1 Ad2 Vars L H H1.
2087
2088

```

```

2089 make-eta-link (s N) (s M) Ad1 Ad2 Vars Links H H1 :-
2090   (pi x\ make-eta-link N M Ad1 Ad2 [x|Vars] (L x) H H1),
2091   close-links L Links.
2092
2093 type deduplicate-map mmap -> mmap ->
2094   subst -> subst -> links -> links -> o.
2095 deduplicate-map [] [] H H L L.
2096 deduplicate-map [(mapping (fv O) (hv M (arity LenM)) as X1) | Map1] progress-beta-link-aux T1 T2 S1 S2 [] :- is-in-pf T2, !,
2097   take-list Map1 (mapping (fv O) (hv M' (arity LenM')) _) !, (T1 ==1 T2) S1 S2.
2098   std.assert! (not (LenM = LenM')) "Deduplicate map, there is a bug"
2099   print "arity-fix links:" {ppmapping X1} "~!~" {ppmapping (mapping (fv O) (hv M' (arity LenM')))},
2100   make-eta-link LenM LenM' M M' [] New H1 H2,
2101   print "new eta link" {pplinks New},
2102   append New L1 L2,
2103   deduplicate-map Map1 Map2 H2 H3 L2 L3.
2104 deduplicate-map [A|As] [A|Bs] H1 H2 L1 L2 :-
2105   deduplicate-map As Bs H1 H2 L1 L2, !.
2106 deduplicate-map [A|_] _ H _ _ _ :-
2107   halt "deduplicating mapping error" {ppmapping A} {ho.ppsubst H}.
2108
2109
2110 15 THE PROGRESS FUNCTION
2111 macro @one :- s z.
2112
2113 type contract-rigid list ho.tm -> ho.tm -> ho.tm -> o.
2114 contract-rigid L (ho.lam F) T :-
2115   pi x\ contract-rigid [x|L] (F x) T. % also checks H Prefix does not make eta
2116 contract-rigid L (ho.app [H|Args]) T :-
2117   rev L LRev, append Prefix LRev Args,
2118   if (Prefix = []) (T = H) (T = ho.app [H|Prefix]).
2119
2120 type progress-eta-link ho.tm -> ho.tm -> ho.subst -> ho.subst -> links -> links -> o.
2121 progress-eta-link (ho.app _ as T) (ho.lam x\ _ as T1) H H1 [] :- !, not (T1 = ho.uva _ _), !, fail.
2122   ({eta-expand T @one} ==1 T1) H H1.
2123 progress-eta-link (ho.con _ as T) (ho.lam x\ _ as T1) H H1 [] :- !, not (T1 = ho.uva _ _), !, fail.
2124   ({eta-expand T @one} ==1 T1) H H1.
2125 progress-eta-link (ho.lam _ as T) T1 H H1 [] :- !,
2126   (T ==1 T1) H H1.
2127 progress-eta-link (ho.uva _ _ as X) T H H1 [] :-
2128   contract-rigid [] T T1, !, (X ==1 T1) H H1.
2129 progress-eta-link (ho.uva Ad _ as T1) T2 H H [eval-link-eta T1 T2] :- !,
2130   if (ho.not_occ Ad H T2) true fail.
2131
2132 type is-in-pf ho.tm -> o.
2133 is-in-pf (ho.app [ho.uva _ _ | _]) :- !, fail.
2134 is-in-pf (ho.lam B) :- !, pi x\ is-in-pf (B x).
2135 is-in-pf (ho.con _).
2136 is-in-pf (ho.app L) :- forall1 is-in-pf L.
2137 is-in-pf N :- name N.
2138 is-in-pf (ho.uva _ L) :- pattern-fragment L.
2139
2140 type arity ho.tm -> nat -> o.
2141 arity (ho.con _) z.
2142 arity (ho.app L) A :- len L A.
2143
2144 type occur-check-err ho.tm -> ho.tm -> ho.subst -> o.
2145 occur-check-err (ho.con _) _ _ :- !.
2146
2147 occur-check-err (ho.app _) _ _ :- !.
2148 occur-check-err (ho.lam _) _ _ :- !.
2149 occur-check-err (ho.uva Ad _) T S :-
2150   not (ho.not_occ Ad S T).
2151
2152 type progress-beta-link-aux ho.tm -> ho.tm ->
2153   ho.subst -> ho.subst -> links -> o.
2154 progress-beta-link-aux T1 T2 S1 S2 [] :- is-in-pf T2, !,
2155   (T1 ==1 T2) S1 S2.
2156 progress-beta-link-aux T1 T2 S S [eval-link-beta T1 T2] :- !.
2157 type progress-beta-link ho.tm -> ho.tm -> ho.subst ->
2158   ho.subst -> links -> o.
2159 progress-beta-link T (ho.app [ho.uva V Scope | L] as T2) S S2 [eval-link-
2160   arity T Arity, len L ArgsNb, ArgsNb >n Arity, !,
2161   minus ArgsNb Arity Diff, mem.new S V1 S1,
2162   eta-expand (ho.uva V1 Scope) Diff T1,
2163   ((ho.uva V Scope) ==1 T1) S1 S2.
2164 progress-beta-link (ho.uva _ _ as T) (ho.app [ho.uva Ad1 Scope1 | L1] as
2165   append Scope1 L1 Scope1L,
2166   pattern-fragment-prefix Scope1L Scope2 L2,
2167   not (Scope1 = Scope2), !,
2168   mem.new S1 Ad2 S2,
2169   len Scope1 Scope1Len,
2170   len Scope2 Scope2Len,
2171   make-eta-link Scope1Len Scope2Len Ad1 Ad2 [] LinkEta S2 S3,
2172   if (L2 = []) (NewLinks = LinkEta, T2 = ho.uva Ad2 Scope2)
2173   (T2 = ho.app [ho.uva Ad2 Scope2 | L2],
2174   NewLinks = [eval-link-beta T T2 | LinkEta]).
2175 progress-beta-link T1 (ho.app [ho.uva _ _ | _] as T2) _ _ _ :-
2176   progress-beta-link T1 (ho.uva _ _ as T) (ho.app [ho.uva _ _ | _] as T2) S1
2177   occur-check-err T T2 S1, !, fail.
2178 progress-beta-link T1 (ho.app [ho.uva _ _ | _] as T2) H H [eval-link-beta
2179   progress-beta-link T1 (ho.app [Hd | T1]) S1 S2 B :-
2180   ho.lam beta Hd T1 T3,
2181   progress-beta-link-aux T1 T3 S1 S2 B.
2182
2183 type solve-link-abs link -> links -> ho.subst -> ho.subst -> o.
2184 solve-link-abs (ho.abs X) R H H1 :-
2185   pi x\ ho.copy x x => (pi S\ ho.deref S x x) =>
2186   solve-link-abs (X x) (R' x) H H1,
2187   close-links R' R.
2188 solve-link-abs (@eval-link-eta A B) NewLinks S S1 :- !,
2189   progress-eta-link A B S S1 NewLinks.
2190 solve-link-abs (@eval-link-beta A B) NewLinks S S1 :- !,
2191   progress-beta-link A B S S1 NewLinks.
2192
2193 type take-link link -> links -> link -> links -> o.
2194 take-link A [B|XS] B XS :- link-abs-same-lhs A B, !.
2195

```

```

2205 take-link A [L|XS] B [L|YS] :- take-link A XS B YS.
2206
2207 type link-abs-same-lhs link -> link -> o.
2208 link-abs-same-lhs (ho.abs F) B :-
2209   pi x\ link-abs-same-lhs (F x) B.
2210 link-abs-same-lhs A (ho.abs G) :-
2211   pi x\ link-abs-same-lhs A (G x).
2212 link-abs-same-lhs (@val-link-eta (ho.uva N _) _) (@val-link-eta (ho.uva N2) _) :-
2213   link-abs-same-lhs A (ho.abs G) B.
2214 type same-link-eta link -> link -> ho.subst -> ho.subst -> o.
2215 same-link-eta (ho.abs F) B H H1 :- !, pi x\ same-link-eta (F x) B H H1 :- !, pi x\ same-link-eta A (G x) H H1 :- !,
2216 same-link-eta A (ho.abs G) H H1 :- !, pi x\ same-link-eta A (G x) H H1 :- !,
2217 same-link-eta (@val-link-eta (ho.uva N S1) A)
2218   (@val-link-eta (ho.uva N S2) B) H H1 :-
2219   std.map2 S1 S2 (x\y\r r = ho.copy x y) Perm,
2220   Perm => ho.copy A A',
2221   (A' ==1 B) H H1.
2222
2223 type progress1 links -> links -> ho.subst -> ho.subst -> o.
2224 progress1 [] [] X X.
2225 progress1 [A|L1] [A|L3] S S2 :- take-link A L1 B L2, !,
2226   same-link-eta A B S1,
2227   progress1 L2 L3 S1 S2.
2228 progress1 [L0|L1] L3 S S2 :- deref-link S L0 L,
2229   solve-link-abs L R S S1, !,
2230   progress1 L1 L2 S1 S2, append R L2 L3.
2231
2232 16 THE DECOMPILER
2233
2234 type abs->lam ho.assignment -> ho.tm -> o.
2235 abs->lam (ho.abs T) (ho.lam R) :- !, pi x\ abs->lam (T x) (R x).
2236 abs->lam (ho.val A) A.
2237
2238 type commit-links-aux link -> ho.subst -> ho.subst -> o.
2239 commit-links-aux (@val-link-eta T1 T2) H1 H2 :-
2240   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2241   (T1' ==1 T2') H1 H2.
2242 commit-links-aux (@val-link-beta T1 T2) H1 H2 :-
2243   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2244   (T1' ==1 T2') H1 H2.
2245 commit-links-aux (ho.abs B) H H1 :-
2246   pi x\ commit-links-aux (B x) H H1.
2247
2248 type commit-links links -> links -> ho.subst -> ho.subst -> o.
2249 commit-links [] [] H H.
2250 commit-links [Abs | Links] L H H2 :-
2251   commit-links-aux Abs H H1, !, commit-links Links L H1 H2.
2252
2253 type decomp-subst map -> map -> ho.subst ->
2254   fo.fsubst -> fo.fsubst -> o.
2255 decomp-subst _ [A|_] _ _ :- fail.
2256 decomp-subst _ [] _ F F.
2257 decomp-subst Map [mapping (fv V0) (hv VM _)|T1] H F F2 :-
2258   mem.set? VM H T, !,
2259   ho.deref-assmt H T TTT,
2260   abs->lam TTT T', tm->fm Map T' T1,
2261   fo.eta-contract T1 T2, mem.assign V0 F T2 F1,
2262   decomp-subst Map T1 H F1 F2.
2263
2264 decomp-subst Map [mapping _ (hv VM _)|T1] H F F2 :-
2265   mem.unset? VM H, decomp-subst Map T1 H F F2.
2266
2267 type tm->fm map -> ho.tm -> fo.fm -> o.
2268 tm->fm _ (ho.con C) (fo.fcon C).
2269 tm->fm L (ho.lam B1) (fo.flam B2) :-
2270   tm->fm L (ho.app L1 T) :- map (tm->fm L) L1 [Hd|T1],
2271   fo.mk-app Hd T1 T.
2272 tm->fm L (ho.app L1 T) :- map (tm->fm L) L1 [Hd|T1],
2273   fo.mk-app Hd T1 T.
2274
2275 type add-new-map-aux ho.subst -> list ho.tm -> map ->
2276   map -> fo.fsubst -> fo.fsubst -> o.
2277 map -> fo.fsubst -> fo.fsubst -> o.
2278 add-new-map-aux _ [] _ [] S S.
2279 add-new-map-aux H [T|Ts] L L2 S S2 :-
2280   add-new-map H T L L1 S S1,
2281   add-new-map-aux H Ts L1 L2 S1 S2.
2282
2283 type add-new-map ho.subst -> ho.tm -> map ->
2284   map -> fo.fsubst -> fo.fsubst -> o.
2285 add-new-map _ (ho.uva N _) Map [] F1 F1 :-
2286   mem Map (mapping _ (hv N _)), !.
2287 add-new-map H (ho.uva N L) Map [Map1 | MapL] F1 F3 :-
2288   mem.new F1 M F2,
2289   len L Arity, Map1 = mapping (fv M) (hv N (arity Arity)),
2290   add-new-map H (ho.app L) [Map1 | Map] MapL F2 F3.
2291 add-new-map H (ho.lam B) Map NewMap F1 F2 :-
2292   pi x\ add-new-map H (B x) Map NewMap F1 F2.
2293 add-new-map H (ho.app L) Map NewMap F1 F3 :-
2294   add-new-map-aux H L Map NewMap F1 F3.
2295 add-new-map _ (ho.con _) _ [] F F :- !.
2296 add-new-map _ N _ [] F F :- name N.
2297
2298 type complete-mapping-under-ass ho.subst -> ho.assignment ->
2299   map -> map -> fo.fsubst -> fo.fsubst -> o.
2300 complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
2301   add-new-map H Val Map1 Map2 F1 F2.
2302 complete-mapping-under-ass H (ho.abs Abs) Map1 Map2 F1 F2 :-
2303   pi x\ complete-mapping-under-ass H (Abs x) Map1 Map2 F1 F2.
2304
2305 type complete-mapping ho.subst -> ho.subst ->
2306   map -> map -> fo.fsubst -> fo.fsubst -> o.
2307 complete-mapping _ [] L L F F.
2308 complete-mapping H [none | T1] L1 L2 F1 F2 :-
2309   complete-mapping H T1 L1 L2 F1 F2.
2310 complete-mapping H [some T0 | T1] L1 L3 F1 F3 :-
2311   ho.deref-assmt H T0 T,
2312   complete-mapping-under-ass H T L1 L2 F1 F2,
2313   append L1 L2 Lall,
2314   complete-mapping H T1 Lall L3 F2 F3.
2315
2316 type decompile map -> links -> ho.subst ->
2317   fo.fsubst -> fo.fsubst -> o.
2318 decompile Map1 L H0 F0 F02 :-
2319   commit-links L L1_ H0 H01, !,
2320   decompile Map1 L1_ H01 F01 F02.

```



```
complete-mapping H01 H01 Map1 Map2 F0 F01,
decompl-subst Map2 Map2 H01 F01 F02.
```

17 AUXILIARY FUNCTIONS

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

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