

# 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 for free when ML binders represent object logic ones; 2) proof construction, and even proof search, are greatly simplified by leveraging the unification procedure provided by the ML. Notable examples of ML are Elf [14], Twelf [16],  $\lambda$ Prolog [9] and Isabelle [22] which have been utilized to implement various formal systems such as First Order Logic [4], Set Theory [13], Higher Order Logic [12], and even the Calculus of Constructions [3].

The object logic we are interested in is Coq's Calculus of Inductive Constructions (CIC)[20]. We aim to develop a higher-order unification-based proof search procedure for it using the ML Elpi [2], a dialect of  $\lambda$ Prolog. Elpi's equational theory includes  $\eta\beta$  equivalence and features a higher-order unification procedure  $\approx_\lambda$  restricted to the pattern fragment [8]. Elpi offers an encoding of CIC suitable for meta-programming [19, 18, 6, 5] but restricts  $\approx_\lambda$  to roughly first-order unification problems only. We refer to this basic encoding as  $\mathcal{F}_0$ .

In this paper we propose a more well-behaved encoding called  $\mathcal{H}_0$ , and show how to translate unification problems from  $\mathcal{F}_0$  to corresponding ones in  $\mathcal{H}_0$ . Consequently, we derive  $\approx_o$ , the higher-order unification procedure of  $\mathcal{F}_0$  that honours  $\eta\beta$ -equivalence (for CIC functions), addresses problems within the pattern fragment, and allows for the use of heuristics to deal with problems outside the pattern fragment. Moreover, as  $\approx_o$  delegates most of the work to  $\approx_\lambda$ , it can be used to efficiently simulate a logic program in  $\mathcal{F}_0$  by taking advantage of unification-related optimizations of the ML, such as clause indexing.

## KEYWORDS

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

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/nnnnnnn.nnnnnnn>

## ACM Reference Format:

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

## 1 INTRODUCTION

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

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

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

**Instance** `fin_fin`:  $\forall n, \text{Finite } (\text{fin } n)$ . (\* r1 \*)

**Instance** `nfact_dec`:  $\forall n \text{ nf}, \text{Decision } (\text{nfact } n \text{ nf})$ . (\* r2 \*)

**Instance** `forall_dec`:  $\forall A \text{ P}, \text{Finite } A \rightarrow$  (\* r3 \*)

$\forall x:A, \text{Decision } (\text{P } x) \rightarrow \text{Decision } (\forall x:A, \text{P } x)$ .

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

`Decision ( $\forall x: \text{fin } 7, \text{nfact } x \ 3$ )` (\* 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 represents a function of type  $A \rightarrow \text{Prop}$  (a predicate over  $A$ ). The solver has to infer a value for  $P$  by unifying the conclusion of

rule 3 with the goal, and in particular, it has to solve the unification problem  $P \ x = \text{nfact} \ x \ 3$ . This higher-order problem falls in the so-called pattern-fragment  $\mathcal{L}$  [8] and admits a unique solution  $\rho$  that assigns the term  $\lambda x. \text{nfact} \ x \ 3$  to  $P$ .

In order to implement such a search in Elpi, we shall describe the encoding of CIC terms and then the encoding of instances as rules. Elpi comes equipped with an Higher Order Abstract Syntax (HOAS [15]) datatype of CIC terms, called `tm`, that includes (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 the standard syntax of  $\lambda$ Prolog [9], the meta-level binding of a variable  $x$  in an expression  $e$  is written as  $\langle x \rangle e$ , while square brackets delimit a list of terms separated by comma. For example, the term  $\langle \forall y:t, \text{nfact} \ y \ 3 \rangle$  is encoded as follows:

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

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

```
finite (app [con "fin", NJ]). (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"]) x\
  app [con "nfact", x, con "3"]). (g)
```

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

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

In this paper we study a more sophisticated encoding of CIC terms and rules that, on a first approximation, would reshape (r3) as follows:

```
decision (all A x\ Pm x) :- link Pm P A, finite A, (r3')
pi x\ decision (app [P, x]).
```

Since  $Pm$  is a higher-order unification variable of type `tm -> tm`, with  $x$  in its scope, the unification problem (p') admits one solution:

```
app [con "nfact", x, con "3"] = Pm x (p')
Pm = x\ app [con "nfact", x, con "3"] (σ)
```

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

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

This simple example is sufficient to show that the encoding we seek is not trivial and does not only concern the head of rules, but the entire sequence of unification problems that constitute the execution of a logic program. In fact the solution for  $P$  above

generates a (Coq)  $\beta$ -redex in the second premise (the predicate under the `pi w\`).

In turn, this redex prevents rule (r2) from backchaining 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 outlined in this example is a subtle mismatch between the equational theories of the meta language and the object language, which in turn makes the unification procedures of the meta language weak. The equational theory of the meta language Elpi encompasses  $\eta\beta$ -equivalence and its unification procedure can solve higher-order problems in the pattern fragment. Although the equational theory of CIC is much richer, for efficiency and predictability reasons, automatic proof search procedures typically employ a unification procedure that only captures a  $\eta\beta$ -equivalence and only operates in  $\mathcal{L}$ . The similarity is striking, but one needs to exercise some caution in order to simulate a logic program in CIC using the unification of Elpi.

*Contributions.* In this paper we identify a minimal language  $\mathcal{F}_0$  in which the problems sketched in the introduction can be formally described. We detail an encoding of a logic program in  $\mathcal{F}_0$  to a strongly related logic program in  $\mathcal{H}_0$  (the language of the meta-language) and we show that the higher-order unification procedure of the meta language  $\approx_\lambda$  can be efficiently used to simulate a higher-order unification procedure  $\approx_o$  for the object language that features  $\eta\beta$ -conversion. We show how  $\approx_o$  can be extended with heuristics to deal with problems outside the pattern fragment. Section 2 formally states the problem and gives the intuition behind our solution; section 3 sets up a basic simulation of first-order logic programs, section 4 and section 5 extend it to higher-order logic programs in the pattern fragment while section 7 goes beyond the pattern fragment. Section 8 discusses the implementation in Elpi. The  $\lambda$ Prolog code discussed in the paper can be accessed at the address <https://github.com/FissoreD/ho-unif-for-free>.

## 2 PROBLEM STATEMENT AND SOLUTION

Even if we encountered the problem working on CIC, we devise a minimal setting to ease its study. In this setting, we have a  $\mathcal{F}_0$  language (for first order) with a rich equational theory and a  $\mathcal{H}_0$  meta language with a simpler one.

### 2.1 Preliminaries: $\mathcal{F}_0$ and $\mathcal{H}_0$

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 as shown in fig. 1. Unification variables in  $\mathcal{F}_0$  (fuva term constructor) have no explicit scope: the arguments of a higher-order variable are given via the `fapp` constructor. For example the term  $\langle P \ x \rangle$  is represented as  $\langle \text{fapp} \ [\text{fuva} \ N, \ x] \rangle$ ,

```
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

where  $N$  is the memory address of  $P$  and  $x$  is a bound variable. In  $\mathcal{H}_o$ , the representation of  $\langle P \ x \rangle$  is instead  $\langle \text{uva } N \ [x] \rangle$ , since unification variables are higher-order and come equipped with an explicit scope.

*Notational conventions.* When we write  $\mathcal{H}_o$  terms outside code blocks we follow the usual  $\lambda$ -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 are 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 it is clear from the context, we shall use the same syntax for  $\mathcal{F}_o$  terms (although we never subscripts unification variables). We use  $s, s_1, \dots$  for terms in  $\mathcal{F}_o$  and  $t, t_1, \dots$  for terms in  $\mathcal{H}_o$ .

## 2.2 Equational theories an unification

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

**2.2.1 Term equality:  $=_o$  and  $=_\lambda$ .** For both languages, we extend the equational theory over ground terms to the full language by adding the reflexivity for unification variables (a variable is equal to itself).

The first four rules are common to both equalities and define the usual congruence over terms. Since we use an HOAS encoding, they also capture  $\alpha$ -equivalence. In addition to that,  $=_o$  has rules for  $\eta$  and  $\beta$ -equivalence.

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

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

The main point in showing these equality tests is to remark how weaker  $=_\lambda$  is, and to identify the four rules that need special treatment in the implementation of  $=_o$ . For brevity, we omit the code of beta: it is sufficient to know that  $\langle \text{beta } F \ L \ R \rangle$  computes in  $R$  the weak head normal form of  $\langle \text{app } [F] \ L \rangle$ . Note that the symbol

**explain** | separates the head of a list from the tail.  
**forall2**

*Substitution:  $\rho s$  and  $\sigma t$ .* We write  $\sigma = \{ X \mapsto t \}$  for the substitution that assigns the term  $t$  to the variable  $X$ . We write  $\sigma t$  for the application of the substitution to a term  $t$ , and  $\sigma X = \{ \sigma t \mid t \in X \}$  when  $X$  is a set of terms. We write  $\sigma \subseteq \sigma'$  when  $\sigma$  is more general than  $\sigma'$ . We shall use  $\rho$  for  $\mathcal{F}_o$  substitutions, and  $\sigma$  for the  $\mathcal{H}_o$  ones. For brevity, in this section, we consider the substitution for  $\mathcal{F}_o$  and  $\mathcal{H}_o$  identical. We defer to section 3.1 a more precise description pointing out their differences.

*Term unification:  $\approx_o$  vs.  $\approx_\lambda$ .*  $\mathcal{H}_o$ 's unification signature is:

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

We write  $\sigma t_1 \approx_\lambda \sigma t_2 \mapsto \sigma'$  when  $\sigma t_1$  and  $\sigma t_2$  unify with substitution  $\sigma'$ . Note that  $\sigma'$  is a refined (i.e. extended) version of  $\sigma$ ; this is reflected by the signature above that relates two substitutions. We write  $t_1 \approx_\lambda t_2 \mapsto \sigma'$  when the initial substitution  $\sigma$  is empty. We write  $\mathcal{L}$  as the set of terms that are in the pattern-fragment, i.e. every higher-order variable is applied to a list of distinct names.

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

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

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

Even if we provide an implementation of the object-language unification  $\approx_o$  in section 3.6, our real goal is the simulation of an entire logic program.

## 2.3 The problem: logic-program simulation

We represent a logic program *run* in  $\mathcal{F}_o$  as a sequence of *steps* of length  $N$ . At each step,  $p$  we unify two terms,  $\mathbb{P}_{p_l}$  and  $\mathbb{P}_{p_r}$ , taken from the list of all unification problems  $\mathbb{P}$ . The composition of these steps starting from the empty substitution  $\rho_0$  produces the final substitution  $\rho_N$ , which is the result of the logic program execution.

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

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

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

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

By analogy with  $\mathbb{P}$ , we write  $\mathbb{T}_{p_l}$  and  $\mathbb{T}_{p_r}$  for the two  $\mathcal{H}_0$  terms being unified at step  $p$ , and we write  $\mathbb{T}_p$  for the set  $\{\mathbb{T}_{p_l}, \mathbb{T}_{p_r}\}$ . hstep is made of two sub-steps: a call to the meta language unification and a check for progress on the set of links, that intuitively will compensate for the weaker equational theory honored by  $\approx_\lambda$ . hrnun compiles all terms in  $\mathbb{P}$ , then executes each step, and finally decompiles the solution. We claim:

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

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

That is, the two executions give the same result if all terms in  $\mathbb{P}$  are in the pattern fragment. Moreover:

PROPOSITION 2.2 (SIMULATION FIDELITY). *In the context of hrnun, if  $\mathbb{P} \subseteq \mathcal{L}$  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  $\mathcal{F}_0$  run is matched by a failure in  $\mathcal{H}_0$  at the same step. We consider this property very important from a practical point of view since it guarantees that the execution traces are strongly related, and in turn, this enables a user to debug a logic program in  $\mathcal{F}_0$  by looking at its execution trace in  $\mathcal{H}_0$ .

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

PROPOSITION 2.3 (FIDELITY RECOVERY). *In the context of hrnun, if  $\rho_{p-1} \mathbb{P}_p \in \mathcal{L}$  (even if  $\mathbb{P}_p \notin \mathcal{L}$ ) then*

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

In other words, if the two terms involved in a step re-enter  $\mathcal{L}$ , then hstep and fstep are again related, even if  $\mathbb{P} \not\subseteq \mathcal{L}$  and hence proposition 2.2 does not apply. Indeed, the main difference between proposition 2.2 and proposition 2.3 is that the assumption of the former is purely static, it can be checked upfront. When this assumption is not satisfied, one can still simulate a logic program and have guarantees of fidelity if, at run time, decidability of higher-order unification is restored.

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

*Backtracking.* We omit it from our model of a logic program's execution since it plays a very minor role, orthogonal to higher-order unification. We point out that each *run* corresponds to a (proof search) branch in the logic program that either fails at some point, or succeeds. A computation that succeeds by backtracking, exploring multiple branches, could be modeled as a set of runs with (possibly non-empty) common prefixes.

## 2.4 The solution (in a nutshell)

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

$\approx_\lambda$  is “well behaved” on  $t$ , in the sense that it does not contradict  $\approx_o$  as it would otherwise do on the “problematic” sub-terms. We now define “problematic” and “well behaved” more formally. We use the  $\diamond$  symbol since it stands for “possibly” in modal logic and all problematic terms are characterized by some “uncertainty”.

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

An example of a  $\diamond\beta$  term 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  $\lambda$ , or a constant, or remain an application.

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

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

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

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

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

*Definition 2.7 (Well behaved set).* Given a set of terms  $X \subseteq \mathcal{H}_0$ ,

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

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

PROPOSITION 2.8 ( $\mathcal{W}$ -ENFORCING). *Given two terms  $s_1$  and  $s_2$ , if  $\exists \rho, \rho s_1 \approx_o \rho s_2$ , then*

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

In other words the compiler outputs terms in  $\mathcal{W}$ , even if its input is not. Note that the property holds for any substitution.  $\rho$  could be given by an oracle and/or not necessarily be a most general one: in  $\mathcal{W} \approx_\lambda$  simply does not contradict  $\approx_o$ .

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

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

Proposition 2.9 is key to proving propositions 2.1 and 2.2. Informally, it says that the problematic terms moved on the side by the compiler are not reintroduced by hstep, hence  $\approx_\lambda$  can continue to operate properly. In sections 3, 5 and 7 we describe how the compiler recognizes terms in  $\diamond\beta$ ,  $\diamond\eta$  and  $\diamond\mathcal{L}$  and how progress takes care of them preserving  $\mathcal{W}$  and ensuring propositions 2.1 to 2.3.



### 3 BASIC COMPILATION AND SIMULATION

#### 3.1 Memory map ( $\mathbb{M}$ ) and substitution ( $\rho$ and $\sigma$ )

Unification variables are identified by a (unique) memory address. The memory and its associated operations are described below:

```

typeabbrev (mem A) (list (option A)).
type set? addr -> mem A -> A -> o.
type unset? addr -> mem A -> o.
type assign addr -> mem A -> A -> mem A -> o.
type new mem A -> addr -> mem A -> o.

```

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

Since each  $\mathcal{H}_0$  unification variables occurs together with a scope, its assignment needs to be abstracted over it to enable the instantiation of the same assignment to different scopes. This is expressed by the `inctx` container, and in particular its `abs` binding constructor.

```

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

```

A solution to a  $\mathcal{F}_0$  variable is a plain term, that is `fsubst` is an abbreviation for `mem fm`.

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

```

kind fvariable type.
type fv addr -> fvariable.
kind hvariable type.
type hv addr -> arity -> hvariable.
kind mapping type.
type (<->) fvariable -> hvariable -> mapping.
typeabbrev mmap (list mapping).

```

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

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

```

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

```

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

It is worth looking at the code of `deref` that applies the substitution to a  $\mathcal{H}_0$  term. Remark how assignments are moved to the current scope, i.e. renaming the `abs`-bound variables with the names in the scope of the unification variable occurrence.

```

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

```

```

deref S (lam F) (lam G) :-
  pi x\ deref S x x => deref S (F x) (G x).
deref S (uva N L) R :- set? N S A,
  move A L T, deref S T R.
deref S (uva N A) (uva N B) :- unset? N S,
  map (deref S) A B.

```

Note that `move` strongly relies on invariant 1: the length of the arguments of all occurrences of a unification are the same. Hence they have the same simple type for the meta-level and hence the number of `abs` nodes in the assignment matches that length. 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.

```

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

#### 3.2 Links ( $\mathbb{L}$ )

As we mentioned in section 2.4 the compiler replaces terms in  $\diamond\eta$ ,  $\diamond\beta$  and  $\diamond\mathcal{L}$  with fresh variables linked to the problematic terms. Terms in  $\diamond\beta$  do not need a link since  $\mathcal{H}_0$  variables faithfully represent the problematic term thanks to their scope.

```

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

```

The right hand side of a link, the problematic term, can occur under binders. To accommodate this situation the compiler wraps `baselink` using the `inctx` container (see `· ⊢ ·` also used for `subst`).

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 is assigned during a step, then the link is considered for progress and possibly eliminated. This is discussed in section 5 and section 7.

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

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

#### 3.3 Compilation

The simple compiler described in this section serves as a base for the extensions in sections 4, 5 and 7. Its main task is to beta normalize the term and map one syntax tree to the other. In order to bring back the substitution from  $\mathcal{H}_0$  to  $\mathcal{F}_0$  the compiler builds a “memory map” connecting the the kind of variables using routine (`malloc`).

The signature of the `comp` predicate below allows for the generation of links (suspended unification problems) that play no role in this section but play a major role in sections 4, 5 and 7. With respect to section 2 the signature also allows for updates to the substitution.

```

type comp fm -> tm -> mmap -> mmap -> links -> links ->
  subst -> subst -> o.

```

```

581 comp (fcon C) (con C) M M L L S S.
582 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :- (cλ)
583   comp-lam F F1 M1 M2 L1 L2 S1 S2.
584 comp (fuva A) (uva B []) M1 M2 L L S1 S2 :-
585   m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
586 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :- (c@)
587   fold6 comp A A1 M1 M2 L1 L2 S1 S2.
588
589 type compile fm -> tm -> mmap -> mmap -> links -> links ->
590   subst -> subst -> o.
591 compile F G M1 M2 L1 L2 S1 S2 :-
592   beta-normal F F', comp F' G M1 M2 L1 L2 S1 S2.

```

The code above uses that possibility in order to allocate space for the variables, i.e. sets their memory address to none (a details not worth mentioning in the previous sections).

explain  
fold6

```

593 type comp-lam (fm -> fm) -> (tm -> tm) ->
594   mmap -> mmap -> links -> links -> subst -> subst -> o.
595 comp-lam F G M1 M2 L1 L3 S1 S2 :-
600   pi x y\ (pi M L S\ comp x y M M L L S S) => (Hλ)
601   comp (F x) (G y) M1 M2 L1 (L2 y) S1 S2,
602   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.

```

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

```

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

### 3.4 Execution

A step in  $\mathcal{H}_0$  consists in unifying two terms and reconsidering all links for progress. If any of the two tasks fails we say that the entire step fails, and it is at this granularity that we can relate steps in the two languages.

```

624 type hstep tm -> tm -> links -> links -> subst -> subst -> o.
625 hstep T1 T2 L1 L2 S1 S3 :-
626   (T1 ≈λ T2) S1 S2,
627   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.

```

633 type progress links -> links -> subst -> subst -> o.
634 progress L L2 S1 S3 :-
635   progress1 L L1 S1 S2,
636   occur-check-links L1,
637   if (L = L1, S1 = S2)

```

```

(L2 = L1, S3 = S1)
(progress L1 L2 S2 S3).

```

**3.4.1 Progress.** In the base compilation scheme `progress1` is the identity on both the links and the substitution, so the fixpoint trivially terminates. Sections 5 and 7 add rules to `progress1` and justify why they don't hinder termination.

**3.4.2 Occur check.** Since compilation moves problematic terms out of the sight of  $\approx_\lambda$ , that procedure can only perform a partial occur check. For example the unification problem  $X \approx_\lambda f Y$  cannot generate a cyclic substitution alone, but should be disallowed if a  $\mathbb{L}$  contains a link like  $\vdash Y =_\eta \lambda z.X_z$ : we don't know yet if  $Y$  will feature a lambda in head position, but we surely know it contains  $X$ , hence  $f Y$  and that fails the occur check. The procedure `occur-check-links` is in charge of performing this check that is needed in order to guarantee proposition 2.2 (SIMULATION FIDELITY).

### 3.5 Substitution decompilation

Decompiling the substitution involves three steps.

First and foremost problematic terms stored in  $\mathbb{L}$  have to be moved back into the game: a suspended link must be turned into a valid assignment. This operation is possible thanks to invariant 2 (LINK LEFT HAND SIDE), thanks to the fact that no link causes an occur-check (3.4.2) and the fact that  $\mathbb{L}$  is duplicate free (??).

The second step amounts at allocating new variables in the memory of  $\mathcal{F}_0$ . This technicality is required because some higher-order unification may need to prune a variable. For example  $F x \cdot y = F x \cdot z$  requires to allocate a variable  $G$  in order to express the assignment  $F_{ab} \mapsto G_a$ .

The last step amounts at decompiling each assignment. Decompiling a term is trivial since  $\mathbb{M}$  is a bijection. The only tricky part concerns the `abs` node. In our simple setting the `flam` node carries no extra info (other than the function body), so each `abs` node can be trivially converted to a `flam` one. In the case of CIC, where lambdas carry the type of the bound variable, one has to store it somewhere. Note how this piece of information is akin to the arity of variables, that is CIC's unification variables have a (function) type, and that type can be used to annotate the lambdas needed in order to express their assignment.

LEMMA 3.1 (COMPILATION ROUND TRIP). *If `compile S T [] M [] _ []` then `decompile M T S`*

### 3.6 Definition of $\approx_o$ and its properties

We already have all the pieces to show the code of  $\approx_\lambda$ .

```

type (≈o) fm -> fm -> fsubst -> o.
(A ≈o B) F :-
  compile A A' [] M1 [] L1 [] S1,
  compile B B' M1 M2 L1 L2 S1 S2,
  hstep A' B' L2 L3 S2 S3,
  decompile M2 L3 S3 [] F.

```

So far the compiler is very basic, it does not really enforce the terms passed to `hstep` are in  $\mathcal{W}$ , and indeed makes no use of the higher-order capabilities of the meta language (all generated variables have an empty scope). Still, we can prove that  $\approx_o$  is a good "first order" unification algorithm if the input already happens to be in  $\mathcal{W}$ . Later,

when the compiler will enforce proposition 2.8 and the proof will be adjusted to cover for these cases.

LEMMA 3.2 (PROPERTIES OF  $\approx_o$ ). *The following properties hold for  $\approx_o$ :*

$$\mathcal{W}(\{t_1, t_2\}) \Rightarrow t_1 \approx_\lambda t_2 \mapsto \rho \Rightarrow \rho t_1 =_\lambda \rho t_2 \quad (3)$$

$$\mathcal{W}(\{t_1, t_2\}) \Rightarrow \rho t_1 =_\lambda \rho t_2 \Rightarrow \exists \rho', t_1 \approx_\lambda t_2 \mapsto \rho' \wedge \rho' \subseteq \rho \quad (4)$$

PROOF SKETCH. In this setting  $\approx_\lambda$  is as strong as  $\approx_o$  on ground terms. What we have to show is that whenever two different  $\mathcal{F}_o$  terms can be made equal by a substitution  $\rho$  (plus the  $\beta_l$  and  $\beta_r$  if needed) we can produce this  $\rho$  by finding a  $\sigma$  via  $\approx_\lambda$  on the corresponding  $\mathcal{H}_o$  terms and by decompiling it. If we look at the syntax of  $\mathcal{F}_o$  terms the only interesting case is  $\text{fuva } x \approx_o s$ . In this case after compilation we have  $Y \approx_\lambda t$  that succeeds with  $\sigma = \{Y \mapsto t\}$  and  $\sigma$  is decompiled to  $\rho = \{Y \mapsto s\}$  by lemma 3.1.  $\square$

THEOREM 3.3 (FIDELITY IN  $\mathcal{W}$ ). *Proposition 2.1 (SIMULATION) and proposition 2.2 (SIMULATION FIDELITY) hold if  $\mathcal{W}(\mathbb{P})$ .*

PROOF SKETCH. Trivial since  $\text{progress}_1$  is a no-op and  $\text{fstep}$  and  $\text{hstep}$  are the same, and by lemma 3.2  $\approx_\lambda$  is equivalent to  $\approx_o$ .  $\square$

### 3.7 Notational conventions

In the following sections we adopt this notation to talk about the compiler's output:  $\mathbb{P}$  is the input set of problems that compiled to  $\mathbb{T}$  with memory mapping  $\mathbb{M}$  and links  $\mathbb{L}$ . For example:

$$\begin{aligned} \mathbb{P} &= \{ p_1 \approx_o p_2 \quad p_3 \approx_o p_4 \} \\ \mathbb{T} &= \{ t_1 \approx_\lambda t_2 \quad t_3 \approx_\lambda t_4 \} \\ \mathbb{M} &= \{ X_1 \mapsto A_1^x \quad X_2 \mapsto A_2^y \} \\ \mathbb{L} &= \{ \Gamma \vdash a =_\eta b \} \end{aligned}$$

We index each sub-problem, sub-mapping, sub-link with its position starting from 1 and counting from left to right, top to bottom. For example,  $\mathbb{T}_2$  corresponds to the  $\mathcal{H}_o$  problem  $t_3 \approx_\lambda t_4$ .

## 4 HANDLING OF $\diamond\beta$

In order to make  $\approx_o$  higher-order we need to take care of terms in  $\diamond\beta$ . In the example below, we can see that the basic compilation given in the previous section is not able to make the  $\mathcal{H}_o$  unification problem succeeds.

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

The unification problem  $\mathbb{T}_1$  fails while trying to unify  $A x$  and  $x$ , equivalent to  $\text{app } [\text{uva } A \text{ [x]}, x]$  versus  $x$ . In order to exploit the higher-order unification algorithm of the meta language, we need to compile the  $\mathcal{F}_o$  term  $X x$  into the  $\mathcal{H}_o$  term  $A_x$  that is  $\text{uva } A \text{ [x]}.$

### 4.1 Compilation and decompilation

We add the following rule before rule  $(c@)$ , where pattern-fragment is a predicate checking if a list of terms is a list of distinct names.

```
comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
  pattern-fragment Ag, !,
  fold6 comp Ag Ag1 M1 M1 L L S1 S1,
  len Ag Arity,
  m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
```

Note that compiling  $\text{Ag}$  cannot create new mappings nor links, since  $\text{Ag}$  is made of bound variables and the hypothetical rule  $??$  loaded by  $\text{comp-lam}$  grants this property.

*Decompilation.* Since no link is created by the compilation of  $\diamond\beta$  terms, no modification to the  $\text{commit-link}$  is needed.

*Progress.* Similarly to decompilation, since no link is produced, no modification to the progress predicate is needed.

Definition 4.1 ( $\mathcal{W}/\beta$ ).  $\mathcal{W}/\beta(X) \Leftrightarrow \forall t \in \mathcal{P}(X), t \notin (\diamond\eta \cup \diamond\mathcal{L})$

LEMMA 4.2 (PROPERTIES OF  $\approx_o$ ). *The following properties hold for  $\approx_o$  where*

$$\mathcal{W}/\beta(\{t_1, t_2\}) \Rightarrow t_1 \approx_\lambda t_2 \mapsto \rho \Rightarrow \rho t_1 =_\lambda \rho t_2 \quad (5)$$

$$\mathcal{W}/\beta(\{t_1, t_2\}) \Rightarrow \rho t_1 =_\lambda \rho t_2 \Rightarrow \exists \rho', t_1 \approx_\lambda t_2 \mapsto \rho' \wedge \rho' \subseteq \rho \quad (6)$$

PROOF SKETCH. If we look at the  $\mathcal{F}_o$  terms there is one more interesting case, namely  $\text{fapp}[\text{fuva } x|\text{W}] \approx_o s$  when  $\text{W}$  are distinct names compiled to  $\tilde{w}$ . In this case the  $\mathcal{H}_o$  problem is  $Y_{\tilde{w}} \approx_\lambda t$  that succeeds with  $\sigma = \{Y_{\tilde{w}} \mapsto t[\tilde{w}/\tilde{y}]\}$  that in turn is decompiled to  $\rho = \{Y \mapsto \lambda \tilde{y}. s[\tilde{w}/\tilde{y}]\}$ . Thanks to  $\beta_l$   $(\lambda \tilde{y}. s[\tilde{w}/\tilde{y}]) \tilde{w} =_o s$ .  $\square$

LEMMA 4.3 ( $\mathcal{W}/\beta$ -ENFORCEMENT). *Given two terms  $s_1$  and  $s_2$  in  $\diamond\beta$ , if  $\exists \rho, \rho s_1 =_o \rho s_2$ , then*

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

PROOF SKETCH. problematic terms are mapped to  $\text{uva}$  by  $\text{comp}$ , the problematic  $\text{fapp}$  node is gone. if  $F x = t$  then  $F = \text{lam } x. t[]$   $\square$

THEOREM 4.4 (FIDELITY IN  $\mathcal{W}/\beta$ ). *Proposition 2.1 (SIMULATION) and proposition 2.2 (SIMULATION FIDELITY) hold if  $\mathcal{W}/\beta(\mathbb{P})$*

PROOF SKETCH. thanks to lemma 4.3  $\approx_\lambda$  is as powerful as  $\approx_o$  in  $\diamond\beta$ , as well as in  $\mathcal{W}$  by lemma 4.2.  $\square$

## 5 HANDLING OF $\diamond\eta$

$\eta$ -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  $\eta$ -contraction of  $\lambda x. t x$ .

Following the compilation scheme of section 3.3 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:  $\text{lam } x \backslash \text{uva } A \text{ [x]}$  and  $\text{con "f"}$  start with different, rigid, term constructors hence  $\approx_\lambda$  fails.

In order to guarantee proposition 2.1 we detect lambdas that can disappear by eta contraction (section 5.1) and we modify the compiled terms by putting fresh unification variables in their place: the problematic term is moved from  $\mathbb{T}$  to  $\mathbb{L}$  (section 5.2). The compilation of the problem  $\mathbb{P}$  above is refined to:

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

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:

fix  
in-  
den-  
ta-  
tion

finish/  
may  
W  
en-  
force-  
ment  
is  
not  
needed  
here

INVARIANT 3 ( $\eta$ -link RHS). *The rhs of any  $\eta$ -link has the shape  $\lambda x.t$  and  $t$  is not a lambda.*

$\eta$ -link are kept in the link store  $\mathbb{L}$  during execution and activated when some conditions hold on lhs or rhs. Link activation is implemented by extending the `progress1` predicate (defined in section 3.4).

## 5.1 Detection of $\diamond\eta$

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

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

The first two examples are easy, and show how a unification variable can expose or erase a variable in their scope and turn the resulting term in an  $\eta$ -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  $\eta$ -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  $\eta$ -contract to  $f(A\ x)$  making  $\lambda x.f(A\ x)$  a potential  $\eta$ -expansion.

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

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

**Lemma 5.2.** A  $\beta$ -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  $\beta$ -normal form there is only one rule that can play a role (namely  $\eta_l$ ), hence if the term  $s$  is not exactly  $x$  (case 1) it can only be an  $\eta$ -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  $\eta$  contraction. In that case the term  $t$  is under the spine of binders  $x_1 \dots x_n$ ,  $t$  can either be  $x$  applied to terms that can *may-contract-to* these variables (case 2), or a unification variable that can be assigned to that application (case 3).  $\square$

**Definition 5.3** (occurs-rigidly). A name  $x$  *occurs-rigidly* in a  $\beta$ -normal term  $t$ , if  $\forall \rho, x \in \mathcal{P}(\rho t)$

In other words  $x$  *occurs-rigidly* in  $t$  if it occurs in  $t$  outside of the scope of a unification variable  $X$ , otherwise an instantiation of  $X$  can make  $x$  disappear from  $t$ . Moreover, note that  $\eta$ -contracting  $t$  cannot make  $x$  disappear, since  $x$  is not a locally bound variable inside  $t$ .

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

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

- (1)  $t$  is a constant or a name applied to the arguments  $l_1 \dots l_m$  such that  $m \geq n$  and for every  $i$  such that  $m - n < i \leq m$  the term  $l_i$  *may-contract-to*  $x_i$ , and no  $x_i$  *occurs-rigidly* in  $l_1 \dots l_{m-n}$ ;
- (2)  $t$  is a unification variable with scope  $W$  and for each  $x_i$  there exists a  $w_j \in W$  such that  $w_j$  *may-contract-to*  $x_i$ .

**Lemma 5.5** ( $\diamond\eta$  DETECTION). If  $t$  is a  $\beta$ -normal term and *maybe-eta*  $t$  holds, then  $t \in \diamond\eta$ .

**PROOF SKETCH.** Follows from definition 5.3 and lemma 5.2  $\square$

Remark that the converse of lemma 5.5 does not hold: there exists a term  $t$  satisfying the criteria (1) of definition 5.4 that is not in  $\diamond\eta$ , i.e. there exists no substitution  $\rho$  such that  $\rho t$  is an  $\eta$ -expansion. A simple counter example is  $\lambda x.f(A\ x)(A\ x)$  since  $x$  does not *occurs-rigidly* in the first argument of  $f$ , and the second argument of  $f$  *may-contract-to*  $x$ . In other words  $A\ x$  may either use or discard  $x$ , but our analysis does not take into account that *the same term* cannot have two contrasting behaviors.

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

## 5.2 Compilation and decompilation

**Compilation.** The following rule is inserted just before rule ( $c_\lambda$ ) from the code in section 3.3.

```
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  $\eta$ -link, which relates the variable `A` with `lam F1`. This link clearly validates invariant 2.

**COROLLARY 5.6.** *The rhs of any  $\eta$ -link has exactly one lambda abstraction, hence the rule above respects invariant 3.*

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



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

### 5.3 Progress

$\eta$ -link are meant to delay the unification of “problematic” terms until we know for sure if the term has to be  $\eta$ -contracted or not.

*Definition 5.7 ( $\eta$ -progress-lhs).* 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  $\eta$ -expansion of  $X$  with  $T$ , that is we run  $\lambda x.X x \approx_{\lambda} T$
- (2) if  $X = \lambda x.t$  we run  $X \approx_{\lambda} T$ .

*Definition 5.8 ( $\eta$ -progress-rhs).* 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  $\eta$ -contracting  $T$  to  $T'$ ,  $T'$  is a term not starting with the  $\text{lam}$  constructor. In the first case,  $X$  is unified with  $T$  and in the second one,  $X$  is unified with  $T'$  (under the context  $\Gamma$ ).

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  $\eta$ -link.

*Definition 5.9 ( $\eta$ -progress-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  $\Delta$  to  $\Gamma$  by renaming its bound variables, i.e.  $T'' = T'[\vec{r}/\vec{s}]$ . We then run  $T \approx_{\lambda} T''$  (under the context  $\Gamma$ ).

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

PROOF SKETCH. By construction, every “problematic” term in  $\mathcal{F}_0$  is replaced with a variable in the corresponding  $\mathcal{H}_0$  term. Therefore,  $t$  is  $\mathcal{W}$ .  $\square$

LEMMA 5.11. *Given a  $\eta$ -link  $l$ , the unification done by  $\eta$ -progress-lhs is between terms in  $\mathcal{W}$*

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

LEMMA 5.12. *Given a  $\eta$ -link  $l$ , the unification done by  $\eta$ -progress-rhs is between terms in  $\mathcal{W}$ .*

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

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

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

LEMMA 5.14. *The introduction of  $\eta$ -link guarantees proposition 2.9 ( $\mathcal{W}$ -PRESERVATION)*

PROOF SKETCH. By lemmas 5.11 to 5.13, every unification performed by the activation of a  $\eta$ -link is done between terms in  $\mathcal{W}$ , therefore, the substitution remains  $\mathcal{W}$ .  $\square$

LEMMA 5.15. *progress terminates.*

PROOF SKETCH. Rules definitions 5.7 and 5.8 and definition 5.9 remove one link from  $\mathbb{L}$ , hence they cannot be applied indefinitely. Moreover each rule only relies on terminating operations such as  $\approx_{\lambda}$ ,  $\eta$ -contraction,  $\eta$ -expansion, relocation (a recursive copy of a finite term).  $\square$

THEOREM 5.16 (FIDELITY IN  $\diamond\eta$ ). *Given a list of unification problems  $\mathbb{P}$ , such that  $\forall t, t \in \mathcal{P}(\mathbb{P}) \wedge t \notin \diamond\mathcal{L}$ , the introduction of  $\eta$ -link guarantees proposition 2.2 (SIMULATION FIDELITY).<sup>1</sup>*

PROOF SKETCH.  $\eta$ -progress-lhs and  $\eta$ -progress-deduplicate activate a  $\eta$ -link when, in the original unification problem, a  $\diamond\eta$  term is unified with respectively a well-behaved term or another  $\diamond\eta$  term. In both cases, the links trigger a unification which succeeds iff the same unification in  $\mathcal{F}_0$  succeeds, guaranteeing proposition 2.2.  $\eta$ -progress-rhs never fails, in fact, this progression refines a variable to a rigid term and plays no role in proposition 2.2.  $\square$

*Example of  $\eta$ -progress-lhs.* The example at the beginning of section 5, once  $\sigma = \{ A \mapsto f \}$ , triggers  $\eta$ -progress-lhs 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  $\sigma = \{ A \mapsto f ; B_x \mapsto f \}$ . Decompilation then generates  $\rho = \{ X \mapsto f \}$ , since  $X$  is mapped to  $B$  and  $f$  is the  $\eta$ -contracted version of  $\lambda x.f x$ .

*Example of  $\eta$ -progress-deduplicate.* A very basic example of  $\eta$ -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  $\eta$ -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  $\eta$ -link progression due to  $\eta$ -progress-rhs

<sup>1</sup>We also suppose that any higher-order variable is always applied with the same number of arguments. This problem is addressed in section 6

## 6 MAKING $\mathbb{M}$ A BIJECTION

In section 3.1, we introduced the definition of “memory map” ( $\mathbb{M}$ ). This memory allows to decompile the  $\mathcal{H}_0$  terms back to the object language. It is the case that, while solving unification problems, a same unification variable  $X$  is used multiple times with different arities.

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

In the unification problems  $\mathbb{P}$  above, we see that  $X$  is used with arity 2 in  $\mathbb{P}_1$  and with arity 1 in  $\mathbb{P}_2$ . By invariant 1 (UNIFICATION-VARIABLE ARITY), we are not allowed to use a same  $\mathcal{H}_0$  variable to represent the two occurrences of  $X$ . If we execute `hrun`, we remark that the unification fails. There is in fact a major problem: `hstep` is not conscious of the connection between the variables  $C$  and  $E$  (both corresponding to  $X$ ), since no link in  $\mathbb{L}$  puts  $C$  and  $E$  in relation and decompilation does not work properly if a  $\mathcal{F}_0$  variable is mapped to two distinct  $\mathcal{H}_0$  variables. The two main drawbacks connected to this situation are firstly the lost of proposition 2.2 (SIMULATION FIDELITY) and secondly, if we want to guarantee at least proposition 2.1 (SIMULATION), we should overcomplicate the decompilation phase. In order to ease the second drawback, we pose the following property:

**PROPOSITION 6.1 ( $\mathbb{M}$  IS A BIJECTION).** *Given a list of unification problems  $\mathbb{P}$ , then the memory map  $\mathbb{M}$  compiled from  $\mathbb{P}$  is a bijection relating the  $\mathcal{F}_0$  and the  $\mathcal{H}_0$  variables.*

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

**Definition 6.2 (align-arity).** Given two mappings  $m_1 : X \mapsto A^m$  and  $m_2 : X \mapsto C^n$  where  $m < n$  and  $d = n - m$ , *align-arity*  $m_1 m_2$  generates the following  $d$  links, one for each  $i$  such that  $0 \leq i < d$ ,

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

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  $\eta$ -expand the occurrence of the variable with lower arity to match the higher arity. Since each  $\eta$ -link can add exactly one lambda, we need as many links as the difference between the two arities.

**Definition 6.3 (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*  $m_1 m_2$ .

**THEOREM 6.4 (FIDELITY WITH MAP-DEDUPLICATION).** *Given a list of unification problems  $\mathbb{P}$ , such that  $\forall t, t \in \mathcal{P}(\mathbb{P}) \Rightarrow \mathcal{W}(t) \vee t \in \Diamond\eta$ , if  $\mathbb{P}$  contains two same  $\mathcal{F}_0$  variables with different arities, then map-deduplication guarantees proposition 2.2 (SIMULATION FIDELITY)*

**PROOF SKETCH.** By the definition of *map-deduplication*, any two occurrences of the same  $\mathcal{F}_0$  variables  $X_1, X_2$  with different arities are related with  $\eta$ -link. If one of the two variables is instantiated, the corresponding  $\eta$ -link is triggered instantiating the related variable. This allows to make unification fail if  $X_1$  and  $X_2$  are unified

with different terms. Finally, since  $\mathbb{P}$  contains only terms that are either  $\mathcal{W}$  or  $\Diamond\eta$ , by theorem 5.16, we can conclude the proof.  $\square$

If we look back the example given 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  $\eta$ -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  $\eta$ -progress-lhs.  $C_{yx}$  is therefore assigned to  $x$  (the second variable of its scope). We can finally see the  $\eta$ -progress-rhs of  $\mathbb{L}_1$ : its rhs is now  $\lambda y. y$  (the term  $C_{xy}$  reduces to  $y$ ). Since it is no more in  $\Diamond\eta$ ,  $\lambda y. y$  is unified with  $E_x$ . After the execution of the remaining `hstep`, we obtain the following  $\mathcal{F}_0$  substitution  $\rho = \{X := \lambda x. \lambda y. y, Y := (f \lambda x. x)\}$ .

## 7 HANDLING OF $\Diamond\mathcal{L}$

In this section we suppose the unification of the object language between two terms  $t_1$  and  $t_2$  to fail each time at least one of the between  $t_1$  or  $t_2$  is outside  $\mathcal{L}$ . This means for instance that  $X \neq_o Y$  and  $X Y \neq_o X Y$ .

In general, unification between  $\Diamond\mathcal{L}$  terms admits more than one solution and committing one of them in the substitution does not guarantee property (2). For instance,  $X a \approx_o a$  admits two different substitutions:  $\rho_1 = \{X \mapsto \lambda x. x\}$  and  $\rho_2 = \{X \mapsto \lambda_. a\}$ . Prefer one over the other may break future unifications.

Given a list of unification problems,  $\mathbb{P}_1 \dots \mathbb{P}_n$  with  $\mathbb{P}_n$  in  $\Diamond\mathcal{L}$ , it is often the case that the resolution of  $\bigwedge_{i=0}^{n-1} \mathbb{P}_i$  gives a partial substitution  $\rho$ , such that  $\rho\mathbb{P}_n$  falls again in  $\mathcal{L}$ .

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

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

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

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

**INVARIANT 4 ( $\mathcal{L}$ -link RHS).** *The rhs of any  $\mathcal{L}$ -link has the shape  $X_{s_1 \dots s_n} t_1 \dots t_m$  such that  $X$  is a unification variable with scope  $s_1 \dots s_n^2$  and  $t_1 \dots t_m$  is a list of terms. This is equivalent to `app[uvva X S | L]`, where  $S = s_1 \dots s_n$  and  $L = t_1 \dots t_m$ .*

### 7.1 Compilation and decompilation

Detection of  $\Diamond\mathcal{L}$  is quite simple to implement in the compiler, since it is sufficient to detect applications with flexible head and argument

<sup>2</sup>with  $s_1 \dots s_n$  that are distinct names

that are not in  $\mathcal{L}$ . The following rule for  $\diamond \mathcal{L}$  compilation is inserted just before rule (c@).

```

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

```

The list  $Ag$  is split into the list  $Pf$  and  $Extra$  such that  $append\ Pf\ Extra\ Ag$  and  $Pf$  is the largest prefix of  $Ag$  such that  $Pf$  is in  $\mathcal{L}$ . The rhs of the  $\mathcal{L}$ -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  $Extra1$ . Note that this construction enforce invariant 4.

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

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

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

## 7.2 Progress

Given a  $\mathcal{L}$ -link  $l$  of the form  $\Gamma \vdash T =_{\mathcal{L}} X_{s_1 \dots s_n} t_1 \dots t_m$ , we provide 3 different activation rules:

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

**Definition 7.4 ( $\mathcal{L}$ -progress-rhs).**  $l$  is removed from  $\mathbb{L}$  if  $X_{s_1 \dots s_n}$  is instantiated to a term  $t$  and the  $\beta$ -reduced term  $t'$  obtained from the application of  $t$  to  $l_1 \dots l_m$  is in  $\mathcal{L}$ . Moreover,  $X$  is unified with  $t$ .

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

**LEMMA 7.6.** *progress terminates*

**PROOF SKETCH.** Let  $l$  a  $\mathcal{L}$ -link in the store  $\mathbb{L}$ . If  $l$  is activated by  $\mathcal{L}$ -progress-rhs, then it disappears from  $\mathbb{L}$  and progress terminates. Otherwise, the rhs of  $l$  is made by a variable applied to  $m$  arguments. At each activation of  $\mathcal{L}$ -progress-refine,  $l$  is replaced by a new  $\mathcal{L}$ -link  $l^1$  having  $m - 1$  arguments. At the  $m^{th}$  iteration, the  $\mathcal{L}$ -link  $l^m$  has no more arguments and is removed from  $\mathbb{L}$ . Note that at the  $m^{th}$  iteration,  $m$  new  $\eta$ -link have been added to  $\mathbb{L}$ , however, by lemma 5.15, the algorithm terminates. Finally

$\mathcal{L}$ -progress-fail also guarantees termination since it makes progress immediately fails.  $\square$

**THEOREM 7.7 (FIDELITY WITH  $\mathcal{L}$ -link).** *The introduction of  $\mathcal{L}$ -link guarantees proposition 2.3 (FIDELITY RECOVERY)*

**PROOF SKETCH.** Let  $\mathbb{T}$  a unification problem and  $\sigma$  a substitution such that  $\mathbb{T} \in \diamond \mathcal{L}$ . If  $\sigma \mathbb{T}$  is in  $\mathcal{L}$ , then by definitions 7.3 and 7.4, the  $\mathcal{L}$ -link associated to the subterm of  $\mathbb{T}$  have been solved and removed. The unification is done between terms in  $\mathcal{L}$  and by theorem 5.16 fidelity is guaranteed. If  $\sigma \mathbb{T}$  is in  $\diamond \mathcal{L}$ , then, by definition 7.5, the unification fails, as per the corresponding unification in  $\mathcal{F}_0$ .  $\square$

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

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

Initially the  $\mathcal{L}$ -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  $\eta$ -progress-lhs and  $E_x$  is assigned to  $x$ . Under this substitution the  $\mathcal{L}$ -link becomes  $x \vdash C_x =_{\mathcal{L}} (D\ x)$ , and by  $\mathcal{L}$ -progress-refine it is replaced with the link:  $\vdash E =_{\eta} \lambda x.D_x$ , while  $C_x$  is unified with  $D_x$ . The second unification problem assigns  $f$  to  $B$ , that in turn activates the second  $\eta$ -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  $\mathcal{L}$ -progress-rhs.** We can take the example provided in section 7. The problem is compiled into:

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

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

## 7.3 Relaxing definition 7.5 ( $\mathcal{L}$ -PROGRESS-FAIL)

Working with terms in  $\mathcal{L}$  is sometime too restrictive [1] and there exist a few strategies to go beyond  $\mathcal{L}$  without implementing Huet's algorithm ??.

Some implementations of  $\lambda$ Prolog [11] such as Teyjus [10] delay the resolution of  $\diamond \mathcal{L}$  unification problems until the substitution put them in  $\mathcal{L}$ . Other systems, for example of the unification algorithm of Coq used in its type-class solver [17], apply heuristics like preferring projection over mimic, and commit to that solution.

In this section we show how we can implement these strategies by simply adding (or removing) rules to the progress predicate. In

D:Si  
puo  
togliere  
se  
serve  
spazio

D:Si  
puo  
togliere  
se  
serve  
spazio

the example below  $\mathbb{P}_1$  is in  $\diamond \mathcal{L}$ .

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

If we want the object language unification to delay the first unification problem (waiting for  $X$  to be instantiated by a future unification), we can relax definition 7.5. Instead of failing because the lhs of the considered  $\mathcal{L}$ -link  $l$  becomes rigid, we keep it in  $\mathbb{L}$  until the head of its rhs also become rigid. In this case, since lhs and rhs have rigid heads, they can be unified just before removing  $l$  from  $\mathbb{L}$ . We can note that this rule trivially guarantees proposition 2.2 (SIMULATION FIDELITY). On the other hand, the occur check becomes partial since we break invariant 2 (LINK LEFT HAND SIDE).

If we want to follow the second strategy and pick an arbitrary solution. For instance, in  $X a b = Y b$ , the last argument of the two terms is the same and unification can succeed by assigning  $Xa$  to  $Y$ . Note that this is not the only solution, since  $\sigma = \{X \mapsto \lambda x. Y\}$  is another valid substitution. Our unification procedure can be modified to accommodate this behavior by adding a rule to progress that tries to align the rightmost arguments and unify the resulting heads whenever lhs of a  $\mathcal{L}$ -link is not in  $\mathcal{L}$ .

Note that delaying unification outside  $\mathcal{L}$  can leave  $\mathcal{L}$ -link for the decompilation phase. Therefore commit-links should be modified accordingly.

## 8 ACTUAL IMPLEMENTATION IN ELPI

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

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

about the progress of 1 link:

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

about the progress of 2 links:

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

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

## 9 RELATED WORK AND CONCLUSION

Different strategies can be used to unify terms of the object language. The first approach that comes to mind consist in implementing  $\approx_o$  as a regular routine in the ML as follows:

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

Opting for this method would result in a suboptimal utilization of the logic programming engine provided by the meta language, as it degrades indexing by eliminating all data from rule heads. Additionally, implementing an unification procedure in the meta language is likely to be significantly slower compared to the built-in one.

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

```
decision (all A x\ P x) :- finite A, pi x\ decision (P x).
```

However, this encoding has two big limitations. Firstly, in CIC, it is not always feasible to adopt it due to the meta language's type system being too limited to accommodate for that one the object language. E.g. CIC can typecheck variadic functions, see for example ??.

Secondly, the CIC encoding provided by Elpi is primarily utilized for meta programming, in order to extend the Coq system. Consequently, it must be able to manipulate terms that are not known in advance without relying on introspection primitives such as Prolog's `functor` and `arg`. In this context, constants need to live in an open world, akin to the string data type used in the preceding examples.

In the literature we could find a related encoding of the Calculus of Constructions (CC) [3]. The goal of that work was to exhibit a logic program performing proof checking for CC and hence relate the proof system of intuitionistic higher-order logic (that animates  $\lambda$ Prolog programs) with the one of CC. The encoding is hence tailored toward a different goal, for example it 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.

*The approach presented in this paper* provides a third option that addresses all the concerns mentioned earlier. It capitalizes on the benefit of not requiring to fully implement the unification algorithm of the object language. Instead, it employs the unification capabilities of the meta language, facilitated by the various links to manage "problematic" subterms. As a result of this choice our encoding takes advantage of indexing data structures and mode analysis for clause filtering. It worth mentioning that we only replace terms with variables only when it is strictly needed, leaving the rest of the term structure intact and hence indexable by the meta language logic programming engine. Moreover, the unification process we obtain is ready to take advantage of potential improvements to the programming engine, such as tabled search, and apply forms of static analysis for the meta language, such as determinacy, to the object language. Finally, our approach is flexible enough to accommodate different strategies and heuristics to handle terms outside the pattern fragment, and it is not tightly coupled with CIC.

## REFERENCES

- [1] Andreas Abel and Brigitte Pientka. "Extensions to Miller's Pattern Unification for Dependent Types and Records". In:



2018. URL: <https://api.semanticscholar.org/CorpusID:51885863>.
- [2] Cvetan Dunchev et al. “ELPI: Fast, Embeddable,  $\lambda$ 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](http://dx.doi.org/10.1007/978-3-662-48899-7_32).
- [3] Amy Felty. “Encoding the Calculus of Constructions in a Higher-Order Logic”. In: ed. by M. Vardi. IEEE, June 1993, pp. 233–244. doi: 10.1109/LICS.1993.287584.
- [4] Amy Felty and Dale Miller. “Specifying theorem provers in a higher-order logic programming language”. In: *Ninth International Conference on Automated Deduction*. Ed. by Ewing Lusk and Ross Overbeck. 310. Argonne, IL: Springer, May 1988, pp. 61–80. doi: 10.1007/BFb0012823.
- [5] Davide Fissore and Enrico Tassi. “A new Type-Class solver for Coq in Elpi”. In: *The Coq Workshop 2023*. Bialystok, Poland, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- [6] Benjamin Grégoire, Jean-Christophe 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. “The Metalanguage  $\lambda$ prolog and Its Implementation”. In: *Functional and Logic Programming*. Ed. by Herbert Kuchen and Kazunori Ueda. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–20. ISBN: 978-3-540-44716-0.
- [11] Gopalan Nadathur and Dale Miller. “An Overview of Lambda-Prolog”. In: June 1988, pp. 810–827.
- [12] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7.
- [13] Lawrence C. Paulson. “Set theory for verification. I: from foundations to functions”. In: *J. Autom. Reason.* 11.3 (Dec. 1993), pp. 353–389. ISSN: 0168-7433. doi: 10.1007/BF00881873. URL: <https://doi.org/10.1007/BF00881873>.
- [14] F. Pfenning. “Elf: a language for logic definition and verified metaprogramming”. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, California, USA: IEEE Press, 1989, pp. 313–322. ISBN: 0818619546.
- [15] F. Pfenning and C. Elliott. “Higher-order abstract syntax”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 199–208. ISBN: 0897912691. doi: 10.1145/53990.54010. URL: <https://doi.org/10.1145/53990.54010>.
- [16] Frank Pfenning and Carsten Sch  rmann. “System Description: Twelf – A Meta-Logical Framework for Deductive Systems”. In: *Automated Deduction – CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2.
- [17] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, C  sar Mu  oz, and Sof  ene Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-71067-7.
- [18] Enrico Tassi. “Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq”. In: *ITP 2019 - 10th International Conference on Interactive Theorem Proving*. Portland, United States, Sept. 2019. doi: 10.4230/LIPIcs.CVIT.2016.23. URL: <https://inria.hal.science/hal-01897468>.
- [19] Enrico Tassi. “Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi  $\lambda$ 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>.
- [20] The Coq Development Team. *The Coq Reference Manual – Release 8.18.0*. <https://coq.inria.fr/doc/V8.18.0/refman>. 2023.
- [21] P. Wadler and S. Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. doi: 10.1145/75277.75283. URL: <https://doi.org/10.1145/75277.75283>.
- [22] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. “The Isabelle Framework”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, C  sar Mu  oz, and Sof  ene Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7.

## APPENDIX

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

## 10 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.

```

## 11 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-normal fm -> fm -> o.
beta-normal (uvar _ _) _ :- halt "Passed uvar to beta-normal".
beta-normal A A :- name A.
beta-normal (fcon A) (fcon A).
beta-normal (fuva A) (fuva A).
beta-normal (flam A) (flam B) :-
  pi x\ beta-normal (A x) (B x).
beta-normal (fapp [flam B | L]) T2 :- !,
  beta (flam B) L T1, beta-normal T1 T2.

```

```

1625 beta-normal (fapp L) (fapp L1) :-
1626   map beta-normal L L1.
1627
1628 type mk-app fm -> list fm -> fm -> o.
1629 mk-app T L S :- beta T L S.
1630
1631 type eta-contract fm -> fm -> o.
1632 eta-contract (fcon X) (fcon X).
1633 eta-contract (fapp L) (fapp L1) :- map eta-contract L L1.
1634 eta-contract (flam F) T :- eta-contract-aux [] (flam F) T.
1635 eta-contract (flam F) (flam F1) :-
1636   pi x\ eta-contract x x => eta-contract (F x) (F1 x).
1637 eta-contract (fuva X) (fuva X).
1638 eta-contract X X :- name X.
1639
1640 type eta-contract-aux list fm -> fm -> fm -> o.
1641 eta-contract-aux L (flam F) T :-
1642   pi x\ eta-contract-aux [x|L] (F x) T. % also checks H Prefix does
1643 eta-contract-aux L (fapp [H|Args]) T :-
1644   rev L LRev, append Prefix LRev Args,
1645   if (Prefix = []) (T = H) (T = fapp [H|Prefix]).
1646
1647
1648
1649 kind inctx type -> type.
1650 type abs (tm -> inctx A) -> inctx A.
1651 type val A -> inctx A.
1652 typeabbrev assignment (inctx tm).
1653 typeabbrev subst (mem assignment).
1654
1655 kind tm type.
1656 type app list tm -> tm.
1657 type lam (tm -> tm) -> tm.
1658 type con string -> tm.
1659 type uva addr -> list tm -> tm.
1660
1661 type ( $\approx_\lambda$ ) tm -> tm -> subst -> subst -> o.
1662 (con C  $\approx_\lambda$  con C) S S.
1663 (app L1  $\approx_\lambda$  app L2) S S1 :- fold2 ( $\approx_\lambda$ ) L1 L2 S S1.
1664 (lam F1  $\approx_\lambda$  lam F2) S S1 :-
1665   pi x\ (pi S\ (x  $\approx_\lambda$  x) S S) => (F1 x  $\approx_\lambda$  F2 x) S S1.
1666 (uva N Args  $\approx_\lambda$  T) S S1 :-
1667   set? N S F,!, move F Args T1, (T1  $\approx_\lambda$  T) S S1.
1668 (T  $\approx_\lambda$  uva N Args) S S1 :-
1669   set? N S F,!, move F Args T1, (T  $\approx_\lambda$  T1) S S1.
1670 (uva M A1  $\approx_\lambda$  uva N A2) S1 S2 :- !,
1671   pattern-fragment A1, pattern-fragment A2,
1672   prune! M A1 N A2 S1 S2.
1673 (uva N Args  $\approx_\lambda$  T) S S1 :- not_occ N S T, pattern-fragment Args,
1674   bind T Args T1, assign N S T1 S1.
1675 (T  $\approx_\lambda$  uva N Args) S S1 :- not_occ N S T, pattern-fragment Args,
1676   bind T Args T1, assign N S T1 S1.
1677
1678 type prune! addr -> list tm -> addr ->
1679   list tm -> subst -> subst -> o.
1680 /* no pruning needed */
1681 prune! N A N A S S :- !.
1682

```

## 12 THE META LANGUAGE

( $\cdot \vdash \cdot$ )

```

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

```

```

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

```

### 13 THE COMPILER

```

kind arity type.
type arity nat -> arity.
kind fvariable type.
type fv addr -> fvariable.
kind hvariable type.
type hv addr -> arity -> hvariable.
kind mapping type.
type (<=>) fvariable -> hvariable -> mapping.
typeabbrev mmap (list mapping).

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

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

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

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

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

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

type maybe-eta fm -> list fm -> o. (◇η)
maybe-eta (fapp [fuva _|Args]) L :- !,
  forall1 (x\ exists (reducible-to [] x) Args) L, !.

```



```

1857 maybe-eta (flam B) L :- !, pi x\ maybe-eta (B x) [x | L].
1858 maybe-eta (fapp [T|Args]) L :- (name T; T = fcon _),
1859   split-last-n {len L} Args First Last,
1860   none (x\ exists (y\ occurs-rigidly x y) First) L,
1861   forall2 (reducible-to []) {rev L} Last.
1862
1863
1864 type locally-bound tm -> o.
1865 type get-scope-aux tm -> list tm -> o.
1866 get-scope-aux (con _) [].
1867 get-scope-aux (uva _ L) L1 :-
1868   forall2 get-scope-aux L R,
1869   flatten R L1.
1870 get-scope-aux (lam B) L1 :-
1871   pi x\ locally-bound x => get-scope-aux (B x) L1.
1872 get-scope-aux (app L) L1 :-
1873   forall2 get-scope-aux L R,
1874   flatten R L1.
1875 get-scope-aux X [X] :- name X, not (locally-bound X).
1876 get-scope-aux X [] :- name X, (locally-bound X).
1877
1878 type names1 list tm -> o.
1879 names1 L :-
1880   names L1,
1881   new_int N,
1882   if (1 is N mod 2) (L1 = L) (rev L1 L).
1883
1884 type get-scope tm -> list tm -> o.
1885 get-scope T Scope :-
1886   get-scope-aux T ScopeDuplicata,
1887   undup ScopeDuplicata Scope.
1888 type rigid fm -> o.
1889 rigid X :- not (X = fuva _).
1890
1891 type comp-lam (fm -> fm) -> (tm -> tm) ->
1892   mmap -> mmap -> links -> links -> subst -> subst -> o.
1893 comp-lam F G M1 M2 L1 L3 S1 S2 :-
1894   pi x y\ (pi M L S\ comp x y M M L L S S) => (Hλ)
1895   comp (F x) (G y) M1 M2 L1 (L2 y) S1 S2,
1896   close-links L2 L3.
1897
1898 type close-links (tm -> links) -> links -> o.
1899 close-links (v\ [X | L v]) [X|R] :- !, close-links L R.
1900 close-links (v\ [X v | L v]) [abs X|R] :- close-links L R.
1901 close-links (_\ []) [].
1902 type comp fm -> tm -> mmap -> mmap -> links -> links ->
1903   subst -> subst -> o.
1904 comp (fcon C) (con C) M M L L S S.
1905 comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
1906   maybe-eta (flam F) [], !,
1907   alloc S1 A S2,
1908   comp-lam F F1 M1 M2 L1 L2 S2 S3,
1909   get-scope (lam F1) Scope,
1910   L3 = [val (link-eta (uva A Scope) (lam F1)) | L2].
1911 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :- (cλ)
1912   comp-lam F F1 M1 M2 L1 L2 S1 S2.
1913 comp (fuva A) (uva B []) M1 M2 L L S1 S2 :-
1914

```

```

1915   m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
1916 comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
1917   pattern-fragment Ag, !,
1918   fold6 comp Ag Ag1 M1 M1 L L S1 S1,
1919   len Ag Arity,
1920   m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
1921 comp (fapp [fuva A|Ag]) (uva B Scope) M1 M3 L1 L3 S1 S4 :- !,
1922   pattern-fragment-prefix Ag Pf Extra,
1923   len Pf Arity,
1924   alloc S1 B S2,
1925   m-alloc (fv A) (hv C (arity Arity)) M1 M2 S2 S3,
1926   fold6 comp Pf Pf1 M2 M2 L1 L1 S3 S3,
1927   fold6 comp Extra Extra1 M2 M3 L1 L2 S3 S4,
1928   Beta = app [uva C Pf1 | Extra1],
1929   get-scope Beta Scope,
1930   L3 = [val (link-llam (uva B Scope) Beta) | L2].
1931 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :- (c@)
1932   fold6 comp A A1 M1 M2 L1 L2 S1 S2.
1933
1934 type alloc mem A -> addr -> mem A -> o.
1935 alloc S N S1 :- mem.new S N S1.
1936
1937 type compile-terms-diagnostic
1938   triple diagnostic fm fm ->
1939   triple diagnostic tm tm ->
1940   mmap -> mmap ->
1941   links -> links ->
1942   subst -> subst -> o.
1943 compile-terms-diagnostic (triple D F01 F02) (triple D H01 H02) M1 M3 L1
1944   beta-normal F01 F01',
1945   beta-normal F02 F02',
1946   comp F01' H01 M1 M2 L1 L2 S1 S2,
1947   comp F02' H02 M2 M3 L2 L3 S2 S3.
1948
1949 type compile-terms
1950   list (triple diagnostic fm fm) ->
1951   list (triple diagnostic tm tm) ->
1952   mmap -> links -> subst -> o.
1953 compile-terms T H M L S :-
1954   fold6 compile-terms-diagnostic T H [] M_ [] L_ [] S_,
1955   print-compil-result T H L_ M_,
1956   deduplicate-map M_ M S_ L_ L.
1957
1958 type make-eta-link-aux nat -> addr -> addr ->
1959   list tm -> links -> subst -> subst -> o.
1960 make-eta-link-aux z Ad1 Ad2 Scope1 L H1 H1 :-
1961   rev Scope1 Scope, eta-expand (uva Ad2 Scope) T1,
1962   L = [val (link-eta (uva Ad1 Scope) T1)].
1963 make-eta-link-aux (s N) Ad1 Ad2 Scope1 L H1 H3 :-
1964   rev Scope1 Scope, alloc H1 Ad H2,
1965   eta-expand (uva Ad Scope) T2,
1966   (pi x\ make-eta-link-aux N Ad Ad2 [x|Scope1] (L1 x) H2 H3),
1967   close-links L1 L2,
1968   L = [val (link-eta (uva Ad1 Scope) T2) | L2].
1969
1970 type make-eta-link nat -> nat -> addr -> addr ->
1971   list tm -> links -> subst -> subst -> o.
1972

```

```

1973 make-eta-link (s N) z Ad1 Ad2 Vars L H H1 :-
1974   make-eta-link-aux N Ad2 Ad1 Vars L H H1.
1975 make-eta-link z (s N) Ad1 Ad2 Vars L H H1 :-
1976   make-eta-link-aux N Ad1 Ad2 Vars L H H1.
1977 make-eta-link (s N) (s M) Ad1 Ad2 Vars Links H H1 :-
1978   (pi x\ make-eta-link N M Ad1 Ad2 [x|Vars] (L x) H H1),
1979   close-links L Links.
1980
1981 type deduplicate-map mmap -> mmap ->
1982   subst -> subst -> links -> links -> o.
1983 deduplicate-map [] [] H H L L.
1984 deduplicate-map (((fv 0 <-> hv M (arity LenM)) as X1) | Map1) Map2 H2 S1 S2 [] :- is-in-pf T2, !,
1985   take-list Map1 ((fv 0 <-> hv M' (arity LenM'))), !,
1986   std.assert! (not (LenM = LenM')) "Deduplicate map, there is a bug!"
1987   print "arity-fix links:" {ppmapping X1} "~!~" {ppmapping ((fv 0 <-> hv M' (arity LenM')))},
1988   make-eta-link LenM LenM' M M' [] New H1 H2,
1989   print "new eta link" {pplinks New},
1990   append New L1 L2,
1991   deduplicate-map Map1 Map2 H2 H3 L2 L3.
1992 deduplicate-map [A|As] [A|Bs] H1 H2 L1 L2 :-
1993   deduplicate-map As Bs H1 H2 L1 L2, !.
1994 deduplicate-map [A|_] _ H _ _ _ :-
1995   halt "deduplicating mapping error" {ppmapping A} {ho.ppsubst H}.
1996
1997 14 THE PROGRESS FUNCTION
1998
1999 macro @one :- s z.
2000
2001 type contract-rigid list ho.tm -> ho.tm -> ho.tm -> o.
2002 contract-rigid L (ho.lam F) T :-
2003   pi x\ contract-rigid [x|L] (F x) T. % also checks H Prefix does not occur
2004 contract-rigid L (ho.app [H|Args]) T :-
2005   rev L LRev, append Prefix LRev Args,
2006   if (Prefix = []) (T = H) (T = ho.app [H|Prefix]).
2007
2008 type progress-eta-link ho.tm -> ho.tm -> ho.subst -> ho.subst -> links -> links -> o.
2009 progress-eta-link (ho.app _ as T) (ho.lam x\ _ as T1) H H1 [] :- !, not (T1 = ho.uva _ _), !, fail.
2010   ({eta-expand T @one} ==1 T1) H H1.
2011 progress-eta-link (ho.con _ as T) (ho.lam x\ _ as T1) H H1 [] :- !,
2012   ({eta-expand T @one} ==1 T1) H H1.
2013 progress-eta-link (ho.lam _ as T) T1 H H1 [] :- !,
2014   (T ==1 T1) H H1.
2015 progress-eta-link (ho.uva _ _ as X) T H H1 [] :-
2016   contract-rigid [T T1, !, (X ==1 T1) H H1.
2017 progress-eta-link (ho.uva Ad _ as T1) T2 H H [eval-link-eta T1 T2] :- !,
2018   if (ho.not_occ Ad H T2) true fail.
2019
2020 type is-in-pf ho.tm -> o.
2021 is-in-pf (ho.app [ho.uva _ _ | _]) :- !, fail.
2022 is-in-pf (ho.lam B) :- !, pi x\ is-in-pf (B x).
2023 is-in-pf (ho.con _).
2024 is-in-pf (ho.app L) :- forall1 is-in-pf L.
2025 is-in-pf N :- name N.
2026 is-in-pf (ho.uva _ L) :- pattern-fragment L.
2027
2028 type arity ho.tm -> nat -> o.
2029 arity (ho.con _) z.
2030
2031 arity (ho.app L) A :- len L A.
2032
2033 type occur-check-err ho.tm -> ho.tm -> ho.subst -> o.
2034 occur-check-err (ho.con _) _ _ :- !.
2035 occur-check-err (ho.app _) _ _ :- !.
2036 occur-check-err (ho.lam _) _ _ :- !.
2037 occur-check-err (ho.uva Ad _) T S :-
2038   not (ho.not_occ Ad S T).
2039
2040 type progress-beta-link-aux ho.tm -> ho.tm ->
2041   ho.subst -> ho.subst -> links -> o.
2042 progress-beta-link-aux T1 T2 S1 S2 [] :- is-in-pf T2, !,
2043   (T1 ==1 T2) S1 S2.
2044 progress-beta-link-aux T1 T2 S S [eval-link-llam T1 T2] :- !.
2045
2046 type progress-beta-link ho.tm -> ho.tm -> ho.subst ->
2047   ho.subst -> links -> o.
2048 progress-beta-link T (ho.app [ho.uva V Scope | L] as T2) S S2 [eval-link-llam T1 T2] :- !,
2049   arity T Arity, len L ArgsNb, ArgsNb >n Arity, !,
2050   minus ArgsNb Arity Diff, mem.new S V1 S1,
2051   eta-expand (ho.uva V1 Scope) Diff T1,
2052   ((ho.uva V Scope) ==1 T1) S1 S2.
2053
2054 progress-beta-link (ho.uva _ _ as T) (ho.app [ho.uva Ad1 Scope1 | L1] as T2) S1 S2 [eval-link-llam T1 T2] :- !,
2055   append Scope1 L1 Scope1L,
2056   pattern-fragment-prefix Scope1L Scope2 L2,
2057   not (Scope1 = Scope2), !,
2058   mem.new S1 Ad2 S2,
2059   len Scope1 Scope1Len,
2060   len Scope2 Scope2Len,
2061   make-eta-link Scope1Len Scope2Len Ad1 Ad2 [] LinkEta S2 S3,
2062   if (L2 = []) (NewLinks = LinkEta, T2 = ho.uva Ad2 Scope2)
2063   (T2 = ho.app [ho.uva Ad2 Scope2 | L2],
2064    NewLinks = [eval-link-llam T T2 | LinkEta]).
2065
2066 progress-beta-link T1 (ho.app [ho.uva _ _ | _] as T2) _ _ _ :-
2067   progress-beta-link (ho.uva _ _ as T) (ho.app [ho.uva _ _ | _] as T2) S1 S2 [eval-link-llam T1 T2] :- !,
2068   occur-check-err T T2 S1, !, fail.
2069
2070 progress-beta-link T1 (ho.app [ho.uva _ _ | _] as T2) H H [eval-link-llam T1 T2] :- !,
2071   progress-beta-link T1 (ho.app [Hd | T1]) S1 S2 B :-
2072   ho.lam beta Hd T1 T3,
2073   progress-beta-link-aux T1 T3 S1 S2 B.
2074
2075 type solve-link-abs link -> links -> ho.subst -> ho.subst -> o.
2076 solve-link-abs (ho.abs X) R H H1 :-
2077   pi x\ ho.copy x x => (pi S\ ho.deref S x x) =>
2078   solve-link-abs (X x) (R' x) H H1,
2079   close-links R' R.
2080
2081 solve-link-abs (@eval-link-eta A B) NewLinks S S1 :- !,
2082   progress-eta-link A B S S1 NewLinks.
2083
2084 solve-link-abs (@eval-link-llam A B) NewLinks S S1 :- !,
2085   progress-eta-link A B S S1 NewLinks.
2086
2087 solve-link-abs (@eval-link-llam A B) NewLinks S S1 :- !,
2088   progress-eta-link A B S S1 NewLinks.

```

```

2089     progress-beta-link A B S S1 NewLinks.
2090
2091     type take-link link -> links -> link -> links -> o.
2092     take-link A [B|XS] B XS :- link-abs-same-lhs A B, !.
2093     take-link A [L|XS] B [L|YS] :- take-link A XS B YS.
2094
2095     type link-abs-same-lhs link -> link -> o.
2096     link-abs-same-lhs (ho.abs F) B :-
2097       pi x\ link-abs-same-lhs (F x) B.
2098     link-abs-same-lhs A (ho.abs G) :-
2099       pi x\ link-abs-same-lhs A (G x).
2100     link-abs-same-lhs (@val-link-eta (ho.uva N _) _) (@val-link-eta (ho.uva N1 _) _) :-
2101       link-abs-same-lhs A (ho.abs G) :- !, pi x\ link-abs-same-lhs A (G x)
2102     type same-link-eta link -> link -> ho.subst -> ho.subst -> o.
2103     same-link-eta (ho.abs F) B H H1 :- !, pi x\ same-link-eta (F x) B H H1 :-
2104     same-link-eta A (ho.abs G) H H1 :- !, pi x\ same-link-eta A (G x) H H1 :-
2105     same-link-eta (@val-link-eta (ho.uva N S1) A)
2106       (@val-link-eta (ho.uva N S2) B) H H1 :-
2107       std.map2 S1 S2 (x\y\r\ r = ho.copy x y) Perm,
2108       Perm => ho.copy A A',
2109       (A' ==l B) H H1.
2110
2111     type progress1 links -> links -> ho.subst -> ho.subst -> o.
2112     progress1 [] [] X X.
2113     progress1 [A|L1] [A|L3] S S2 :- take-link A L1 B L2, !,
2114       same-link-eta A B S S1,
2115       progress1 L2 L3 S1 S2.
2116     progress1 [L0|L1] L3 S S2 :- deref-link S L0 L,
2117       solve-link-abs L R S S1, !,
2118       progress1 L1 L2 S1 S2, append R L2 L3.
2119
2120
2121
2122     type abs->lam ho.assignment -> ho.tm -> o.
2123     abs->lam (ho.abs T) (ho.lam R) :- !, pi x\ abs->lam (T x) (R x).
2124     abs->lam (ho.val A) A.
2125
2126     type commit-links-aux link -> ho.subst -> ho.subst -> o.
2127     commit-links-aux (@val-link-eta T1 T2) H1 H2 :-
2128       ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2129       (T1' ==l T2') H1 H2.
2130     commit-links-aux (@val-link-llam T1 T2) H1 H2 :-
2131       ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2132       (T1' ==l T2') H1 H2.
2133     commit-links-aux (ho.abs B) H H1 :-
2134       pi x\ commit-links-aux (B x) H H1.
2135
2136     type commit-links links -> links -> ho.subst -> ho.subst -> o.
2137     commit-links [] [] H H.
2138     commit-links [Abs | Links] L H H2 :-
2139       commit-links-aux Abs H H1, !, commit-links Links L H1 H2.
2140
2141     type decomp-subst map -> map -> ho.subst ->
2142       fo.fsubst -> fo.fsubst -> o.
2143     decomp-subst _ [A|_] _ _ _ :- fail.
2144     decomp-subst _ [] _ F.
2145     decomp-subst Map [mapping (fv V0) (hv VM _)|T1] H F F2 :-
2146
2147     mem.set? VM H T, !,
2148     ho.deref-assmt H T TTT,
2149     abs->lam TTT T', tm->fm Map T' T1,
2150     fo.eta-contract T1 T2, mem.assign V0 F T2 F1,
2151     decomp-subst Map T1 H F1 F2.
2152     decomp-subst Map [mapping _ (hv VM _)|T1] H F F2 :-
2153     mem.unset? VM H, decomp-subst Map T1 H F F2.
2154
2155     type tm->fm map -> ho.tm -> fo.fsubst -> o.
2156     tm->fm _ (ho.con C) (fo.fcon C).
2157     tm->fm L (ho.lam B1) (fo.flam B2) :-
2158     tm->fm L (ho.app L1) T :- map (tm->fm L) L1 [Hd|T1],
2159     fo.mk-app Hd T1 T.
2160     tm->fm L (ho.uva VM TL) T :- mem L (mapping (fv V0) (hv VM _)),
2161     map (tm->fm L) TL T1, fo.mk-app (fo.fuva V0) T1 T.
2162
2163     type add-new-map-aux ho.subst -> list ho.tm -> map ->
2164       map -> fo.fsubst -> fo.fsubst -> o.
2165     add-new-map-aux _ [] _ [] S S.
2166     add-new-map-aux H [T|Ts] L L2 S S2 :-
2167     add-new-map H T L L1 S S1,
2168     add-new-map-aux H Ts L1 L2 S1 S2.
2169
2170     type add-new-map ho.subst -> ho.tm -> map ->
2171       map -> fo.fsubst -> fo.fsubst -> o.
2172     add-new-map _ (ho.uva N _) Map [] F1 F1 :-
2173     mem Map (mapping _ (hv N _)), !.
2174     add-new-map H (ho.uva N L) Map [Map1 | MapL] F1 F3 :-
2175     mem.new F1 M F2,
2176     len L Arity, Map1 = mapping (fv M) (hv N (arity Arity)),
2177     add-new-map H (ho.app L) [Map1 | Map] MapL F2 F3.
2178     add-new-map H (ho.lam B) Map NewMap F1 F2 :-
2179     pi x\ add-new-map H (B x) Map NewMap F1 F2.
2180     add-new-map H (ho.app L) Map NewMap F1 F3 :-
2181     add-new-map-aux H L Map NewMap F1 F3.
2182     add-new-map _ (ho.con _) _ [] F F :- !.
2183     add-new-map _ N _ [] F F :- name N.
2184
2185     type complete-mapping-under-ass ho.subst -> ho.assignment ->
2186       map -> map -> fo.fsubst -> fo.fsubst -> o.
2187     complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
2188     add-new-map H Val Map1 Map2 F1 F2.
2189     complete-mapping-under-ass H (ho.abs Abs) Map1 Map2 F1 F2 :-
2190     pi x\ complete-mapping-under-ass H (Abs x) Map1 Map2 F1 F2.
2191
2192     type complete-mapping ho.subst -> ho.subst ->
2193       map -> map -> fo.fsubst -> fo.fsubst -> o.
2194     complete-mapping _ [] L L F F.
2195     complete-mapping H [none | T1] L1 L2 F1 F2 :-
2196     complete-mapping H T1 L1 L2 F1 F2.
2197     complete-mapping H [some T0 | T1] L1 L3 F1 F3 :-
2198     ho.deref-assmt H T0 T,
2199     complete-mapping-under-ass H T L1 L2 F1 F2,
2200     append L1 L2 LA11,
2201     complete-mapping H T1 LA11 L3 F2 F3.
2202
2203
2204

```

```

2205   type decompile map -> links -> ho.subst ->
2206       fo.fsubst -> fo.fsubst -> o.
2207   decompile Map1 L H0 F0 F02 :-
2208       commit-links L L1_ H0 H01, !,
2209       complete-mapping H01 H01 Map1 Map2 F0 F01,
2210       decomp1-subst Map2 Map2 H01 F01 F02.
2211

```

## 16 AUXILIARY FUNCTIONS

```

2213   type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A ->
2214       list A1 -> B -> B -> C -> C -> o.
2215   fold4 _ [] [] A A B B.
2216   fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0,
2217       fold4 F XS YS A0 A1 B0 B1.
2218
2219   type len list A -> nat -> o.
2220   len [] z.
2221   len [_|L] (s X) :- len L X.
2222

```

2223  
2224  
2225  
2226  
2227  
2228  
2229  
2230  
2231  
2232  
2233  
2234  
2235  
2236  
2237  
2238  
2239  
2240  
2241  
2242  
2243  
2244  
2245  
2246  
2247  
2248  
2249  
2250  
2251  
2252  
2253  
2254  
2255  
2256  
2257  
2258  
2259  
2260  
2261  
2262

2263  
2264  
2265  
2266  
2267  
2268  
2269  
2270  
2271  
2272  
2273  
2274  
2275  
2276  
2277  
2278  
2279  
2280  
2281  
2282  
2283  
2284  
2285  
2286  
2287  
2288  
2289  
2290  
2291  
2292  
2293  
2294  
2295  
2296  
2297  
2298  
2299  
2300  
2301  
2302  
2303  
2304  
2305  
2306  
2307  
2308  
2309  
2310  
2311  
2312  
2313  
2314  
2315  
2316  
2317  
2318  
2319  
2320