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 to facilitate this task in two ways: 1) variable binding and substitution are for free when meta language binders represent object logic ones; 2) proof construction, and proof search, are greatly simplified by leveraging the unification procedure provided by the meta language. Notable examples of meta languages are Elf [21], Twelf [23], λProlog [16], Beluga [24], Abella [8] and Isabelle [30] which have been used to implement or specify many formal systems such as First Order Logic [5], Set Theory [20], Higher Order Logic [19], and the Calculus of Constructions [4].

The object logic we are interested in is Coq's type theory [28]. We aim to develop a higher-order unification-based proof search procedure using the meta language Elpi [3], a dialect of λProlog. Elpi's equational theory includes $\beta\eta$-equivalence and features a higher-order unification procedure $\simeq_m$ for the pattern fragment [15]. Elpi offers an encoding of Coq terms that is suitable for meta programming [27, 26, 9, 6] but that restricts $\simeq_m$ to first-order unification problems only. We refer to this basic encoding as $\mathcal{O}$.

In this paper we translate unification problems in $\mathcal{O}$ to an alternative encoding called $\mathcal{M}$, from which we derive $\simeq_o$, the higher-order unification procedure of $\mathcal{O}$. $\simeq_o$ honours $\beta\eta$-equivalence for terms within the pattern fragment, and allows for the use of heuristics when the terms fall outside the pattern fragment. Moreover, as $\simeq_o$ delegates most of the work to $\simeq_m$, it can be used to efficiently simulate a logic program in $\mathcal{O}$ by taking advantage of unification-related optimizations of the meta language, such as clause indexing.

KEYWORDS

Logic Programming, Meta-Programming, Higher-Order Unification

ACM Reference Format:
Davide Fissore and Enrico Tassi. 2024. Higher-Order unification for free!: Reusing the meta-language unification for the object language. In *26th International Symposium on Principles and Practice of Declarative Programming (PPDP 2024), September 10–11, 2024, Milano, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3678232.3678233

PPDP 2024, September 10–11, 2024, Milano, Italy

ACM ISBN 979-8-4007-0969-2/24/09
https://doi.org/10.1145/3678232.3678233

1 INTRODUCTION

We aim to implement a form of proof search known as type-class resolution [29, 25] for Coq's type system [28]. Type-class solvers are unification based 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 meta programming language [27], a dialect of λProlog, already used to extend Coq in various ways [27, 26, 9, 6].

The use of a meta language facilitates the implementation of a proof system 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. In this paper we focus on this last aspect of the implementation of a Prolog-like proof system, 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 library [12]. 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: ∀n, Finite (fin n).                    (* r1 *)
Instance nfact_dec: ∀n nf, Decision (nfact n nf). (* r2 *)
Instance forall_dec: ∀A P, Finite A →                    (* r3 *)
  (∀x:A, Decision (P x)) → Decision (∀x:A, P x).
```

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

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

The proof found by the solver back-chains on rule 3 (the only rule about the ∀ quantifier), and then solves the premises with rules 1 and 2 respectively. Note that rule 3 features a second-order parameter `P` that represents a function of type `A → Prop` (a 1-argument predicate over `A`). The solver infers a value for `P` by unifying the conclusion of rule 3 with the goal, and in particular, it has to solve the unification problem `P x = nfact x 3`. This higher-order problem falls in the so-called pattern fragment and admits a unique solution:

```
P = λx.nfact x 3                                         (ρ)
```

In the rest of this paper, we refer to the pattern fragment as $\mathcal{L}$ [14].

In order to implement such a search in Elpi, we shall describe the encoding of Coq terms and then the encoding of instances as rules. Elpi comes equipped with a Higher-Order Abstract Syntax

(HOAS [22]) datatype of Coq 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 λProlog [16], the meta-level binding of a variable `x` in an expression `e` is written as «`x\ e`», while square brackets delimit a list of terms separated by comma. For example, the term «`∀y:t, nfact y 3`» is encoded as follows:

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

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

```
finite   (app [con"fin", N]).                              (r1)
decision (app [con"nfact", N, NF]).                        (r2)
decision (all A x\ app [P, x]) :- finite A,                (r3)
  pi w\ decision (app [P, w]).
```

Unfortunately this 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 here

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

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)
```

The problem above has no solution in a first-order language where `P` can only stand for a predicate symbol, while in a higher-order language `P` can stand for any 1-argument predicate.

In this paper we study a more sophisticated encoding of Coq 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 «`link Pm A P`» brings the assignment (σ) back to the domain `tm` of Coq terms, obtaining the expected solution (ρ):

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

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

In fact the solution for `P` above generates a (Coq) β-redex in the second premise (the predicate under the `pi w\`). 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 $\beta\eta$-equivalence and its unification procedure can solve higher-order problems in the pattern fragment. Although the equational theory of Coq is much richer, for efficiency and predictability reasons automatic proof search procedures typically employ a unification procedure that only captures a $\beta\eta$-equivalence and only operates in $\mathcal{L}$. The similarity is striking, but one needs to exercise some caution in order to simulate a higher-order logic program in Coq using the unification of Elpi.

Contributions. In this paper we identify a minimal language $\mathcal{O}$ in which the problems sketched in the introduction can be formally described. We detail an encoding of a logic program in $\mathcal{O}$ to a strongly related logic program in $\mathcal{M}$, the language of the meta language, and we show that the higher-order unification procedure of the meta language $\simeq_m$ can be used to simulate a higher-order unification procedure $\simeq_o$ for the object language that features $\beta\eta$-conversion. We show how $\simeq_o$ can be extended with heuristics to deal with problems outside the pattern fragment.
Section 2 states the problem formally 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 λProlog 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 Coq's Calculus of Inductive Constructions, we devise a minimal setting to ease its study. In this setting, we have a (first order) language $\mathcal{O}$ with a rich equational theory and a $\mathcal{M}$ language with a simpler one.

2.1 Preliminaries: $\mathcal{O}$ and $\mathcal{M}$

In order to reason about unification we provide a description of the $\mathcal{O}$ and $\mathcal{M}$ languages where unification variables are first-class terms, i.e. they have a concrete syntax as shown in fig. 1. Unification variables in $\mathcal{O}$ (`o-uva` term constructor) have no explicit scope: the "arguments" of a higher-order variable are given via the `o-app` constructor. For example the term «`P x`» is represented as «`o-app [o-uva N, x]`», where `N` is the memory address of `P` and `x` is a bound variable. In $\mathcal{M}$ the representation of «`P x`» is instead «`uva N [x]`», since unification variables are higher-order and come equipped with an explicit scope.

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

Figure 1: The $\mathcal{O}$ and $\mathcal{M}$ languages

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

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

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

2.2 Equational theories and unification

2.2.1 Term equality: $=_o$ and $=_m$. For both languages, we extend the equational theory over ground terms to the full language by adding reflexivity for unification variables, i.e. a unification variable is equal to itself but different to any other term.

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

```
type (=o) to -> to -> o.                                        (=o)
o-con X =o o-con X.
o-app A =o o-app B :- forall2 (=o) A B.
o-lam F =o o-lam G :- pi x\ x =o x => F x =o G x.              (λλ)
o-uva N =o o-uva N.
o-lam F =o T :-                                                 (ηl)
  pi x\ beta T [x] (T' x), x =o x => F x =o T' x.
T =o o-lam F :-                                                 (ηr)
  pi x\ beta T [x] (T' x), x =o x => T' x =o F x.
o-app [o-lam X|L] =o T :- beta (o-lam X) L R, R =o T.          (βl)
T =o o-app [o-lam X|L] :- beta (o-lam X) L R, T =o R.          (βr)

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

Note that the symbol `|` separates the head of a list from the tail; that `forall2` applies a binary predicate to two lists (of equal length); and that `=>` is λProlog syntax for implication and is used to augment the current program with an extra rule. Both procedures use implication to assume that the fresh nominal constants introduces by `pi` are equal to themselves.

The main point in showing these equality tests is to remark how weaker $=_m$ is, and to identify the four rules that need special treatment in the implementation of $\simeq_o$. For brevity, we omit the code of `beta`: it is sufficient to know that «`beta F L R`» computes in `R` the weak-head normal form of «`o-app [F|L]`».

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

Term unification: $\simeq_o$ vs. $\simeq_m$. $\mathcal{M}$'s unification signature is:

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

We write $\sigma t_1 \simeq_m \sigma t_2 \mapsto \sigma'$ when σt_1 and σt_2 unify with substitution σ'. Note that σ' is a refined (i.e. extended) version of σ; this is reflected by the signature above that relates two substitutions. We write $t_1 \simeq_m t_2 \mapsto \sigma'$ when the initial substitution σ is empty. We write $\mathcal{L}$ as the set of terms that are in the pattern-fragment, i.e. every unification variable is applied to a list of distinct names.

The specification of a "good" unification procedure restricted to a domain $\mathcal{D}$ is the following:

Proposition 2.1 (Good unification). *A good unification $\simeq$ for equality $=$ in the domain $\mathcal{D}$ satisfies the following properties:*

$$\{\ t_1,\ t_2\ \} \subseteq \mathcal{D} \Rightarrow t_1 \simeq t_2 \mapsto \rho \Rightarrow \rho t_1 = \rho t_2 \tag{1}$$

$$\{\ t_1,\ t_2\ \} \subseteq \mathcal{D} \Rightarrow \rho t_1 = \rho t_2 \Rightarrow \exists \rho', t_1 \simeq t_2 \mapsto \rho' \wedge \rho' \subseteq \rho \tag{2}$$

The meta language of choice is expected to provide an implementation of $\simeq_m$ that is a good unification for $=_m$ in $\mathcal{L}$.

Even if we provide an implementation of the object-language unification $\simeq_o$ in section 3.6, our final goal is to simulate the execution of an entire logic-program.

2.3 The problem: logic-program simulation

We represent a logic program *run* in $\mathcal{O}$ as a sequence of n *steps*. Each step p consists in unifying two terms, $\mathbb{P}_{p_l}$ and $\mathbb{P}_{p_r}$, taken from the list of all unification problems $\mathbb{P}$. We write $\mathbb{P}_p = \{\ \mathbb{P}_{p_l}, \mathbb{P}_{p_r}\ \}$ and $s \in \mathbb{P} \Leftrightarrow \exists p, s \in \mathbb{P}_p$. The composition of these steps starting from the empty substitution ρ_0 produces the final substitution ρ_n, which is the result of the logic program execution.

$$\text{step}_o(\mathbb{P}, p, \rho) \mapsto \rho' \stackrel{def}{=\!=} \rho\mathbb{P}_{p_l} \simeq_o \rho\mathbb{P}_{p_r} \mapsto \rho'$$

$$\text{run}_o(\mathbb{P}, n) \mapsto \rho_n \stackrel{def}{=\!=} \bigwedge_{p=1}^{n} \text{step}_o(\mathbb{P}, p, \rho_{p-1}) \mapsto \rho_p$$

In order to simulate a logic program we start by compiling $\mathbb{P}$ to $\mathbb{Q}$, i.e. each $\mathcal{O}$-term $s \in \mathbb{P}$ is translated to a $\mathcal{M}$-term $t \in \mathbb{Q}$. 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 map m and a list of links l. The variable map connects unification variables in $\mathcal{M}$ to variables in $\mathcal{O}$ and is used to "decompile" the substitution, that is written $\langle \sigma, m, l \rangle^{-1} \mapsto \rho$. Links are an accessory piece of information whose description is deferred to section 2.4.

Then we simulate each run in $\mathcal{O}$ with a run in $\mathcal{M}$ as follows:

$$\begin{aligned}
&\text{step}_m(\mathbb{Q}, p, \sigma, \mathbb{L}) \mapsto (\sigma'', \mathbb{L}') \stackrel{def}{=\!=} \\
&\quad \sigma\mathbb{Q}_{p_l} \simeq_m \sigma\mathbb{Q}_{p_r} \mapsto \sigma' \wedge \text{progress}(\mathbb{L}, \sigma') \mapsto (\mathbb{L}', \sigma'') \\
&\text{run}_m(\mathbb{P}, n) \mapsto \rho_n \stackrel{def}{=\!=} \\
&\quad \mathbb{Q} \times \mathbb{M} \times \mathbb{L}_0 = \{(t, m, l) | s \in \mathbb{P}, \langle s \rangle \mapsto (t, m, l)\} \\
&\quad \bigwedge_{p=1}^{n} \text{step}_m(\mathbb{Q}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p) \\
&\quad \langle \sigma_n, \mathbb{M}, \mathbb{L}_n \rangle^{-1} \mapsto \rho_n
\end{aligned}$$

By analogy with $\mathbb{P}$, we write $\mathbb{Q}_{p_l}$ and $\mathbb{Q}_{p_r}$ for the two $\mathcal{M}$ terms being unified at step p, and we write $\mathbb{Q}_p$ for the set $\{ \mathbb{Q}_{p_l}, \mathbb{Q}_{p_r} \}$. The step_m procedure 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 $\simeq_m$. Procedure run_m compiles all terms in $\mathbb{P}$, then executes each step, and finally decompiles the solution. We claim:

Proposition 2.2 (Run equivalence). $\forall \mathbb{P}, \forall n$, *if* $\mathbb{P} \subseteq \mathcal{L}$

$$\text{run}_o(\mathbb{P}, n) \mapsto \rho \wedge \text{run}_m(\mathbb{P}, n) \mapsto \rho' \Rightarrow \forall s \in \mathbb{P}, \rho s =_o \rho' s$$

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

Proposition 2.3 (Simulation fidelity). *In the context of* run_o *and* run_m*, if* $\mathbb{P} \subseteq \mathcal{L}$ *we have that* $\forall p \in 1 \ldots n$,

$$\text{step}_o(\mathbb{P}, p, \rho_{p-1}) \mapsto \rho_p \Leftrightarrow \text{step}_m(\mathbb{Q}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p)$$

In particular, this property guarantees that a *failure* in the O run is matched by a failure in $\mathcal{M}$ *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 O by looking at its execution trace in $\mathcal{M}$.

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

Proposition 2.4 (Fidelity recovery). *In the context of* run_o *and* run_m*, if* $\rho_{p-1}\mathbb{P}_p \subseteq \mathcal{L}$ *(even if* $\mathbb{P}_p \not\subseteq \mathcal{L}$*) then*

$$\text{step}_o(\mathbb{P}, p, \rho_{p-1}) \mapsto \rho_p \Leftrightarrow \text{step}_m(\mathbb{Q}, p, \sigma_{p-1}, \mathbb{L}_{p-1}) \mapsto (\sigma_p, \mathbb{L}_p)$$

In other words, if the two terms involved in a step re-enter $\mathcal{L}$, then step_m and step_o are again related, even if $\mathbb{P} \not\subseteq \mathcal{L}$ and hence proposition 2.3 does not apply. Indeed, the main difference between proposition 2.3 and proposition 2.4 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 $F \simeq \lambda x.a$ and $F\ a \simeq a$: 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 σ_2, making the second problem re-enter $\mathcal{L}$.

Backtracking. We omit it from our model of logic-program 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 a (possibly non-empty) common prefix.

2.4 The solution (in a nutshell)

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

Definition 2.5 ($\diamond\beta$). $\diamond\beta$ is the set of terms of the form $F\ x_1 \ldots x_n$ such that $x_1 \ldots x_n$ 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 λ, or a constant, or a name, or remain an application.

Definition 2.6 ($\diamond\eta$). $\diamond\eta$ is the set of terms $\lambda x.s$ such that $\exists \rho, \rho(\lambda x.s)$ is an η-redex.

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

Definition 2.7 ($\diamond\mathcal{L}$). $\diamond\mathcal{L}$ is the set of terms of the form $F\ t_1 \ldots t_n$ such that $t_1 \ldots t_n$ 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, given $s \in \diamond\mathcal{L}$ there may exist a substitution ρ such that $\rho s \in \mathcal{L}$.

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

Definition 2.8 (Well behaved set). Given a set of terms X,

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

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

Proposition 2.9 ($\mathcal{W}$-enforcing). *Given a term* s *in* O*,*

$$\langle s \rangle \mapsto (t, m, l) \Rightarrow \mathcal{W}(t)$$

Proposition 2.10 (compiler-useful). *Given two terms* s_1 *and* s_2*, if* $\exists \rho, \rho s_1 =_o \rho s_2$*, then*

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

In other words the compiler outputs terms in $\mathcal{W}$ even if its input is not. And if two terms can be unified in the source language then their images can also be unified in the target language. Note that the property holds for any substitution, not necessarily a most general one: when applied to terms in $\mathcal{W}$ the meta-language unification $\simeq_m$ agrees with the object-language equality $=_o$. Note that a compiler that translate all terms to unification variables would satisfy this property, but wound certainly fail to verify proposition 2.3.

Proposition 2.11 ($\mathcal{W}$-preservation). $\forall \mathbb{Q}, \forall \mathbb{L}, \forall p, \forall \sigma, \forall \sigma'$

$$\mathcal{W}(\sigma\mathbb{Q}) \wedge \sigma\mathbb{Q}_{p_l} \simeq_m \sigma\mathbb{Q}_{p_r} \mapsto \sigma' \Rightarrow \mathcal{W}(\sigma'\mathbb{Q})$$
$$\mathcal{W}(\sigma\mathbb{Q}) \wedge \text{progress}(\mathbb{L}, \sigma) \mapsto (\_, \sigma') \Rightarrow \mathcal{W}(\sigma'\mathbb{Q})$$

Proposition 2.11 is key to proving propositions 2.2 and 2.3. Informally, it says that the problematic terms moved on the side by the compiler are not reintroduced by step_m, hence $\simeq_m$ can continue

to operate properly. In sections 4, 5 and 7 we describe how to recognize terms in $\diamond\beta$, $\diamond\eta$ and $\diamond\mathcal{L}$ and how progress takes care of links and how it preserves $\mathcal{W}$ and ensures propositions 2.2 to 2.4.

We prove proposition 2.3 (Simulation fidelity) in sections 5 to 7 and proposition 2.2 (Run equivalence) only in section 4, where links are not present yet. In order to prove proposition 2.2 in the presence of links one needs to refine proposition 2.3 by relating the two substitutions ρ_p and σ_p: they are equivalent only by using decompilation to take into account terms in $\mathbb{L}$. Then, by induction on the number of steps, proposition 2.3 implies proposition 2.2. We discuss proposition 2.4 (Fidelity recovery) in section 7.

3 BASIC COMPILATION AND SIMULATION

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

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 $\mathcal{M}$ unification variable 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 $\mathcal{O}$ variable is, instead, a plain term, that is, `fsubst` is an abbreviation for `mem to`. The compiler establishes a mapping between variables of the two languages.

```
kind ovariable type.
type ov addr -> ovariable.
kind mvariable type.
type mv addr -> arity -> mvariable.
kind mapping type.
type (<->) ovariable -> mvariable -> mapping.
typeabbrev mmap (list mapping).
```

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

Invariant 1 (Unification-variable arity). *Each variable* `A` *in* $\mathcal{M}$ *has a (unique) arity* `N` *and each occurrence* `(uva A L)` *is such that* `L` *has length* `N`.

```
type m-alloc ovariable -> mvariable -> mmap -> mmap ->
  subst -> subst -> o.                                     (malloc)
m-alloc Ov Mv M M S S :- mem M (Ov <-> Mv), !.
m-alloc Ov Mv M [Ov <-> Mv|M] S S1 :- Mv = mv N _, new S N S1.
```

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

It is worth examining the code of `deref`, that applies the substitution to a $\mathcal{M}$ term. Notice how assignments are moved to the current scope, i.e. the `abs`-bound variables are renamed with the names in the scope of the unification variable occurrence.

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

Note that `move` strongly relies on invariant 1: the length of the arguments of all occurrences of a unification variable is the same. Hence, they have the same simple type for the meta-level, and therefore the number of `abs` nodes in the assignment matches that length. This guarantees 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 «`abs x\abs y\y` » and $\sigma = \{\, A \mapsto \lambda x.\lambda y.y \,\}$ for «`lam x\lam y\y` ».

3.2 Links ($\mathbb{L}$)

As 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 $\mathcal{M}$ 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 links (list (inctx baselink)).
```

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

In the examples we represent links as equations under a context. The equality sign is subscripted with the 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 $\mathcal{M}$ to $\mathcal{O}$ the compiler builds a "memory map" connecting the $\mathcal{O}$ variables to the $\mathcal{M}$ ones.

The signature of the `comp` predicate below allows for the generation of links (suspended unification problems), which plays no role in this section but plays a major role in sections 4, 5 and 7.

```
type comp to -> tm -> mmap -> mmap -> links -> links ->
  subst -> subst -> o.
comp (o-con C) (con C) M M L L S S.
comp (o-lam F) (lam F1) M1 M2 L1 L2 S1 S2 :-                (c_λ)
  comp-lam F F1 M1 M2 L1 L2 S1 S2.
comp (o-uva A) (uva B []) M1 M2 L L S1 S2 :-
  m-alloc (ov A) (mv B (arity z)) M1 M2 S1 S2.
comp (o-app A) (app A1) M1 M2 L1 L2 S1 S2 :-                (c_@)
  fold6 comp A A1 M1 M2 L1 L2 S1 S2.

type compile to -> tm -> mmap -> mmap ->  links -> links ->
  subst -> subst -> o.
compile F G M1 M2 L1 L2 S1 S2 :-
  beta-normal F F', comp F' G M1 M2 L1 L2 S1 S2.
```

With respect to section 2, the signature also allows for updates to the substitution. The code above uses that possibility in order to allocate space for the variables, i.e. it sets their memory address to `none` (a detail not worth mentioning in the previous sections).

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

In the code above, the syntax `pi x y\..` is syntactic sugar for iterated `pi` abstraction, as in `pi x\ pi y\...` The auxiliary predicate `fold6` folds a predicate with three accumulators (the memory map, the links and the substitution) over a lists to obtain a new list and the final values of the accumulators.

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

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

3.4 Execution

A step in $\mathcal{M}$ consists in unifying two terms and reconsidering all links for progress. If either of these tasks fails, we consider the entire step to fail. It is at this granularity that we can relate steps in the two languages.

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

Note that the infix notation `((A ≃_m B) C D)` is syntactic sugar for `((≃_m) A B C D)`.

Reconsidering links is a fixpoint process because the progress of a link can update the substitution, which may then enable another link to progress.

```
type progress links -> links -> subst -> subst -> o.
progress L L3 S S3 :-
  progress1 L L1 S S1,
  occur-check-links L1,
  scope-check L1 L2 S1 S2
  if (L = L2, S = S2) (L3 = L2, S3 = S2)
     (progress L2 L3 S2 S3).
```

3.4.1 Progress. In the base compilation scheme, `progress1` is the identity function on both the links and the substitution, so the fixpoint trivially terminates. Sections 5 and 7 add rules to `progress1` and explain why the do not hinder termination.

3.4.2 Scope check. The predicate `scope-check` replaces each link of the form $\Gamma \vdash X_{x_1 \ldots x_n} =_\eta t$ with a link $x_1 \ldots x_n \vdash X_{x_1 \ldots x_n} =_\eta t'$ whenever $\{x_1 \ldots x_n\} \subset \Gamma$. The term $t' = \lambda x.Y_{x_1 \ldots x_n}\ x$ is obtained after executing $\lambda x.Y_{x_1 \ldots x_n}\ x \simeq_m t$ using a fresh Y. This unification ensures that t' cannot contain variables in $\Gamma / \{x_1 \ldots x_n\}$, and if it fails then progress hence step_m fails. In turn this grants fidelity in the case where the lhs of an η-`link` is pruned of a name that occurs (rigidly) in t: links represent suspended unification problems *that may succeed*.

3.4.3 Occur check. Since compilation moves problematic terms out of the sight of $\simeq_m$, that procedure can only perform a partial occur check. For example, the unification problem $X \simeq_m f\ Y$ cannot generate a cyclic substitution alone, but should be disallowed if $\mathbb{L}$ 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. The procedure `occur-check-links` is in charge of performing this check that is needed in order to guarantee proposition 2.3 (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), and that no link causes an occur-check (3.4.3) and the fact that $\mathbb{L}$ is duplicate-free (see section 6).

The second step allocates in the memory of $\mathcal{O}$ new variables used to implement the pruning of higher-order variables. For example, $F\ x\ y = F\ x\ z$ requires allocating a variable G in order to express the assignment $F_{ab} \mapsto G_a$.

The final step is to decompile each assignment. The `val` node carries a term that is easy to decompile since $\mathbb{M}$ is a bijection. Since the `o-lam` node carries no additional information (other than the function body), each `abs` node can be decompiled to a `o-lam` one.

However, if the `o-lam` node was carrying more information, as is the case for Coq where it holds the type of the bound variable, one would need to store this piece of information in the memory map, were we store the arity (that is a very simple function type).

Proposition 3.1 (Compilation round trip). *If* $\langle s\rangle \mapsto (t, m, l)$ *and* $l \in \mathbb{L}$ *and* $m \in \mathbb{M}$ *and* $\sigma = \{A \mapsto t\}$ *and* $X \mapsto A^n \in \mathbb{M}$ *then* $\langle \sigma, \mathbb{M}, \mathbb{L}\rangle^{-1} \mapsto \rho$ *and* $\rho X =_o \rho s$.

We omit to sketch the proof of this property for brevity.

3.6 Definition of $\simeq_o$ and its properties

We already have all the ingredients to show the code of $\simeq_o$.

```
type (≃o) to -> to -> 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 that the terms passed to step_m 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 $\simeq_o$ is a good "first-order" unification algorithm if the input already happens to be in $\mathcal{W}$. Later, when the compiler will ensure proposition 2.9 ($\mathcal{W}$-enforcing), the proof will be refined to cover for the new cases.

Theorem 3.2 (Properties of $\simeq_o$ in $\mathcal{W}$). *The implementation of* $\simeq_o$ *above is a good unification for* $=_o$ *(as per proposition 2.1) in the domain* $\mathcal{W}$.

Proof sketch. In this setting, $=_m$ is as strong as $=_o$ on ground terms. What we have to show is that whenever two different $\mathcal{O}$ terms can be made equal by a substitution ρ, we can produce this ρ by finding a σ via $\simeq_m$ on the corresponding $\mathcal{M}$ terms and by decompiling it. If we look at the syntax of $\mathcal{O}$, the only interesting case is «o-uva X ≃o s». In this case, after compilation, we have «uva Y [] ≃m t» that succeeds with $\sigma = \{Y \mapsto t\}$ and σ is decompiled to $\rho = \{X \mapsto s\}$ by proposition 3.1. □

Theorem 3.3 (Fidelity in $\mathcal{W}$). *Proposition 2.2 (Run equivalence) and proposition 2.3 (Simulation fidelity) hold if* $\mathcal{W}(\mathbb{P})$.

Proof sketch. Since progress1 is a no-op then step_o and step_m are the same. By theorem 3.2, $\simeq_m$ is equivalent to $\simeq_o$ in $\mathcal{W}$. □

3.7 Notational conventions

In the following sections we use the following notation for input and output of the compiler. $\mathbb{P}$ represents the input unification problems in $\mathcal{O}$ while $\mathbb{Q}$ their corresponding compiled version with memory mapping $\mathbb{M}$ and links $\mathbb{L}$. For example:

$$\begin{aligned}\mathbb{P} &= \{\ p_1 \simeq_o p_2 \quad p_3 \simeq_o p_4\ \}\\ \mathbb{Q} &= \{\ t_1 \simeq_m t_2 \quad t_3 \simeq_m t_4\ \}\\ \mathbb{M} &= \{\ X_1 \mapsto A_1 \quad X_2 \mapsto A_2\ \}\\ \mathbb{L} &= \{\ \Gamma \vdash a =_\eta b\ \}\end{aligned}$$

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

As we deal with each category of problematic terms we weaken the assumptions of theorem 3.3 using the following definitions:

Definition 3.4 ($\mathcal{W}_\beta$). $\mathcal{W}_\beta(X) \Leftrightarrow \forall t \in \mathcal{P}(X), t \notin (\diamond\eta \cup \diamond\mathcal{L})$

Definition 3.5 ($\mathcal{W}_{\beta\eta}$). $\mathcal{W}_{\beta\eta}(X) \Leftrightarrow \forall t \in \mathcal{P}(X), t \notin \diamond\mathcal{L}$

4 HANDLING OF $\diamond\beta$

The basic compiler given in the previous section is unable to make the following higher-order unification problem succeed.

$$\begin{aligned}\mathbb{P} &= \{\ \lambda x.(f\ (X\ x)\ a) \simeq_o \lambda x.(f\ x\ a)\ \}\\ \mathbb{Q} &= \{\ \lambda x.(f\ (A\ x)\ a) \simeq_m \lambda x.(f\ x\ a)\ \}\\ \mathbb{M} &= \{\ X \mapsto A\ \}\end{aligned}$$

The unification problem $\mathbb{Q}_1$ fails while trying to unify $A\ x$ and x, which is equivalent to «app [uva A [], x]» versus x. In order to exploit the higher-order unification algorithm of the meta language, we compile the $\mathcal{O}$ term $X\ x$ into the $\mathcal{M}$ term A_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 (o-app [o-uva 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 (ov A) (mv B (arity Arity)) M1 M2 S1 S2.
```

Note that compiling Ag cannot create new mappings nor links, since Ag is made of bound variables, and the hypothetical rule (H_λ) loaded by comp-lam grants this property.

Decompilation. No change to commit-link since no link is added.

Progress. No change to progress since no link is added.

Lemma 4.1 (Properties of $\simeq_o$ in $\mathcal{W}_\beta$). *Given the updated compilation scheme, the* $\simeq_o$ *of section 3.6 is a good unification (as per proposition 2.1) for* $=_o$ *in the domain* $\mathcal{W}_\beta$.

Proof sketch. If we look at the $\mathcal{O}$ terms, there is one more interesting case, namely o-app[o-uva X|W] ≃o s when W are distinct names compiled to $\vec{w}$. In this case the $\mathcal{M}$ problem is $Y_{\vec{w}} \simeq_m t$ that succeeds with $\sigma = \{Y_{\vec{y}} \mapsto t[\vec{w}/\vec{y}]\}$, which in turn is decompiled to $\rho = \{Y \mapsto \lambda\vec{y}.s[\vec{w}/\vec{y}]\}$. Thanks to β_l we have $(\lambda\vec{y}.s[\vec{w}/\vec{y}])\ \vec{w} =_o s$. □

Lemma 4.2 ($\mathcal{W}$-partial-enforcement). $\forall s, \langle s\rangle \mapsto (t, m, l)$, *if* $\mathcal{W}_\beta(s)$ *then* $\mathcal{W}(t)$.

In other words the compiler eliminates all subterms that are in $\diamond\beta$.

Theorem 4.3 (Fidelity in $\mathcal{W}_\beta$). *Proposition 2.2 (Run equivalence) and proposition 2.3 (Simulation fidelity) hold if* $\mathcal{W}_\beta(\mathbb{P})$.

Proof sketch. Thanks to lemma 4.2, $\simeq_m$ is as powerful as $\simeq_o$ in $\mathcal{W}_\beta$, as well as in $\mathcal{W}$ by theorem 3.3. □

5 HANDLING OF $\diamond\eta$

A term $\lambda x.t\ x$ is said to be the η-redex of t if x does not occur free in t, and conversely we say that t is the η-contraction of $\lambda x.t\ x$. The equational theory of $\mathcal{O}$ identifies these terms, but the current compilation scheme does not, as shown by the following example:

while $\lambda x.X\ x \simeq_o f$ does admit the solution $\rho = \{ X \mapsto f \}$, the corresponding problem in $\mathbb{Q}$ does not.

$$\begin{aligned}\mathbb{P} &= \{\ \lambda x.X\ x \simeq_o f\ \}\\ \mathbb{Q} &= \{\ \lambda x.A_x \simeq_m f\ \}\\ \mathbb{M} &= \{\ X \mapsto A\ \}\end{aligned}$$

The reason is that «`lam x\ uva A [x]`» and «`con"f"`» start with different term constructors.

In order to guarantee proposition 2.2, we detect lambda abstractions that can disappear by η-contraction (section 5.1), and we modify the compiler so that it generates fresh unification variables in their place and moves the problematic term from $\mathbb{Q}$ to $\mathbb{L}$ (section 5.2). The compilation of the problem $\mathbb{P}$ above is refined to:

$$\begin{aligned}\mathbb{P} &= \{\ \lambda x.X\ x \simeq_o f\ \}\\ \mathbb{Q} &= \{\ A \simeq_m f\ \}\\ \mathbb{M} &= \{\ X \mapsto B\ \}\\ \mathbb{L} &= \{\ \vdash A =_\eta \lambda x.B_x\ \}\end{aligned}$$

As per invariant 2 the η-`link` left-hand side is a variable while the right-hand side is a term in $\diamond\eta$ that has the following property:

Invariant 3 (η-`link` rhs). *The rhs of any η-`link` has the shape $\lambda x.t$ and t is in $\mathcal{W}$.*

Each η-`link` is kept in the link store $\mathbb{L}$ during execution and is activated under some conditions. Activation is implemented in section 5.3 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 $\lambda x.r$ can be an η-redex, i.e. if there exists a substitution ρ such that $\rho(\lambda x.r) =_o s$. The detection of lambda abstractions that can "disappear" is not as trivial as it may seems. Here a few examples:

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

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

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

Lemma 5.2. *A β-normal term $s = \lambda x_1 \dots x_n.t$* may-contract-to *$x$ only if one of the following three conditions holds:*

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

Proof sketch. Since our terms are in β-normal form the only rule that can play a role is η_l, and note that that rule uses `beta` to add an argument to an application or to the scope of a variable. If the term s is not exactly x (case 1) it can only be an η-redex of x, or a unification variable that can be assigned to x, or a combination of both. If s begins with a lambda, then the lambda can only disappear by η contraction. In that case, the term t under the spine of binders $x_1 \dots x_n$ can either be x applied to terms that *may-contract-to* these variables (case 2), or a unification variable that can be assigned to the application $x\ x_1 \dots x_n$ (case 3). □

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

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

We can now describe the implementation of $\diamond\eta$ detection:

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

(1) t is a constant c or a name y 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 $l_1 \dots l_{m-n}\ y$;
(2) t is a unification variable with scope W and for each x_i there exists a $w_j \in W$ such that w_j *may-contract-to* x_i.

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

Proof sketch. Follows from definition 5.3 and lemma 5.2 □

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

As we will see in the rest of this section, this is not a problem since it does not break proposition 2.2 nor proposition 2.3.

5.2 Compilation and decompilation

Compilation. The following rule is inserted just before rule (c_λ) from the code in section 3.3.

```
comp (o-lam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
  maybe-eta (o-lam 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].
```

Whenever `o-lam F` is detected to be in $\diamond\eta$ it is compiled to `lam F1` and replaced by the fresh variable `A`. This variable sees all the names free in `lam F1` and is connected to `lam F1` via a η-`link`. Invariant 2 (Link left hand side) holds for this link. Moreover:

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

Proof sketch. By contradiction, suppose that the rule above is applied and that the rhs of the link is $\lambda x.\lambda y.t$, where x and y occur in t. If *maybe-eta* $\lambda y.t$ holds then 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* $\lambda y.t$ does not hold, then also *maybe-eta* $\lambda x.\lambda y.t$ does not hold either, contradicting the assumption that the rule was applied. □

Lemma 5.7 ($\mathcal{W}$-partial-enforcement). *$\forall s, \langle s\rangle \mapsto (t, m, l)$, if $\mathcal{W}_{\beta\eta}(s)$ then $\mathcal{W}(t)$.*

Proof sketch. By lemma 5.5 □

Decompilation. The decompilation of a η-`link` is performed by unifying the lhs with the rhs. Note that this unification never fails, since the lhs is a flexible term not appearing in any other η-`link` (by definition 5.9).

5.3 Progress

η-`links` are meant to delay the unification of "problematic" terms until we know for sure if their head lambdas can be η-contracted.

Definition 5.8 (η-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 η-redex of X with t, that is we run $\lambda x.X\ x \simeq_m t$
(2) if $X = \lambda x.s$ we run $X \simeq_m t$.

Definition 5.9 (η-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 Δ to Γ by renaming its bound variables, i.e. $T'' = T'[\vec{r}/\vec{s}]$. We then run $T \simeq_m T''$ (under the context Γ).

Lemma 5.10. *Let $\lambda x.t$ be the rhs of a η-`link`, then $\mathcal{W}(t)$, enforcing invariant 3.*

Proof sketch. By lemma 5.7. □

Lemma 5.11. *Given a η-`link` $\Gamma \vdash X =_\eta \lambda x.t$, the unification done by η-progress-lhs is between terms in $\mathcal{W}$.*

Proof sketch. Let σ be a substitution such that $\mathcal{W}(\sigma X)$ (since $\simeq_m$ preserves $\mathcal{W}$). By invariant 3, we have $\mathcal{W}(t)$. If $\sigma X = \lambda x.r$, then r is unified with t and both terms are in $\mathcal{W}$. Otherwise, $\lambda x.X\ x$ is unified with $\lambda x.t$, and $X\ x$ and t are both in $\mathcal{W}$. □

Lemma 5.12. *The unification done by η-progress-deduplicate is between terms in $\mathcal{W}$.*

Proof. Given two η-`links` $\Gamma_1 \vdash X =_\eta \lambda x.t$ and $\Gamma_2 \vdash Y =_\eta \lambda x.t'$ the unification is performed between t and t'. By lemma 5.7 both terms are in $\mathcal{W}$. □

Lemma 5.13. *The progress of η-`link` guarantees proposition 2.11 ($\mathcal{W}$-preservation)*

Proof sketch. By lemmas 5.11 and 5.12, every unification performed by the activation of a η-`link` is between terms in $\mathcal{W}$. □

Lemma 5.14. `progress` *terminates.*

Proof sketch. Rules 5.8 and 5.9 remove one link from $\mathbb{L}$, hence they cannot be applied indefinitely. Moreover each rule only relies on terminating operations such as $\simeq_m$, η-contraction, η-redex, relocation (a recursive copy of a finite term). □

Theorem 5.15 (Fidelity in $\mathcal{W}_{\beta\eta}$). *Given a list of unification problems $\mathbb{P}$ such that $\mathcal{W}_{\beta\eta}(\mathbb{P})$, if the memory map is bijective then the introduction of η-`link` guarantees proposition 2.3 (Simulation fidelity).*

Proof sketch. We want to prove that step_o succeeds iff step_m succeeds, where step_m is made by a unification step u and a link progression p. Without loss of generality we consider a O unification problem of the form $s_1 \simeq_o s_2$ where $s_1 \in \diamond\eta$ and is compiled to a variable X with the accessory link $\Gamma \vdash X =_\eta \lambda x.t_1$. The unification u always succeeds, since X is a fresh variable, we only have to consider the link progression p. Two cases should be analyzed:

- $s_2 \in \diamond\eta$, in this case, the unification problem becomes $X \simeq_m Y$ with the extra link $\vdash Y =_\eta \lambda x.t_2$. After the unification of X and Y, *η-progress-deduplicate* is fired and, once the lambdas are crossed, t_1 and t_2 are unified. This unification succeeds iff $s_1 \simeq_o s_2$ succeeds by theorem 4.3.
- $s_2 \notin \diamond\eta$, in this case s_2 is compiled to t_2 and the unification step u assigns t_2 to X and triggers *η-progress-lhs*. If s_1 and s_2 are equal thanks to η_l then progression p unifies $\lambda x.t_1$ with the η-redex of t_2 namly $\lambda x.t_2\ x$, otherwise they are equal thanks to rule $\lambda\lambda$ and the link progress p unifies $\lambda x.t_1$ with t_2. Alternatively, if s_1 are different s_2 it cannot be because of the λ constructor in the head of t_1, since rule $\lambda\lambda$ or rule η_l move under it, eventually finding two different subterms s_1' and s_2'. By theorem 4.3, $\simeq_m$ fails on terms t_1' and t_2' even if *η-progress-lhs* η-expands t_2.

□

Corollary 5.16 (Properties of $\simeq_o$ in $\mathcal{W}_{\beta\eta}$). *Given the updated compilation scheme, procedure $\simeq_o$ of section 3.6 is a good unification (as per proposition 2.1) for $=_o$ in the domain $\mathcal{W}_{\beta\eta}$.*

Example of η-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. This rule runs $\lambda x.f\ x \simeq_m \lambda x.B_x$, resulting in $\sigma = \{ A \mapsto f\ ;\ B_x \mapsto f \}$. Since X is mapped to B and B_x is decompiled into $\lambda x.f\ x$, by the definition of η-`link` decompilation σ is decompiled to $\rho = \{ X \mapsto \lambda x.f\ x \}$.

6 MAKING $\mathbb{M}$ A BIJECTION

We want to allow for partial application of functions. As a consequence the same unification variable X can be used multiple

times with different arities. When it is the case the memory map $\mathbb{M}$ generated by the compiler is not a bijection. For example:

$$\begin{array}{l}\mathbb{P} = \{\ \lambda x.(X\ x)\ \simeq_o\ f \qquad X\ \simeq_o\ \lambda x.a\ \}\\ \mathbb{Q} = \{\ \qquad A\ \simeq_m\ f \qquad C\ \simeq_m\ \lambda x.a\ \}\\ \mathbb{M} = \{\ X \mapsto C \qquad X \mapsto B\ \}\\ \mathbb{L} = \{\ \vdash A =_\eta\ \lambda x.B_x\ \}\end{array}$$

in the unification problems $\mathbb{P}$ above, X is used with arity 1 in $\mathbb{P}_1$ and with arity 0 in $\mathbb{P}_2$. In order to preserve invariant 1 (Unification-variable arity) the compiler did generate two entries in $\mathbb{M}$ for X, namely B and C. However, there is no connection between these two variables and any incompatible assignements to them would not be detected, in turn breaking proposition 2.3. To address this issue, we post-process $\mathbb{M}$ to ensure the following property:

Proposition 6.1 ($\mathbb{M}$ is a bijection). *After compilation, for each $\mathcal{O}$-variable X in $\mathbb{P}$ and for each $\mathcal{M}$-variable A in $\mathbb{Q}$ there is exactly one entry $X \mapsto A^n$ in $\mathbb{M}$.*

Note that running $\simeq_m$ may require allocating new variables in $\mathcal{M}$ as explained in section 3.5. The property above ignores these variables since they are fresh and have no corresponding variable in $\mathcal{O}$.

The core procedure of this post-processing step is *align-arity* that is iterated by *map-deduplication*:

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

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

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

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

*Definition 6.3 (*map-deduplication*).* Forall mappings $m_1, m_2 \in \mathbb{M}$ such that $m_1 : X \mapsto A^m$ and $m_2 : X \mapsto C^n$ and $m < n$ we remove m_1 from $\mathbb{M}$ and add to $\mathbb{L}$ the result of *align-arity* $m_1\ m_2$.

Theorem 6.4 (Fidelity with *map-deduplication*). *Given a list of unification problems $\mathbb{P}$, such that $\mathcal{W}_{\beta\eta}(\mathbb{P})$, if $\mathbb{P}$ contains the same $\mathcal{O}$-variable used at different arities, then* map-deduplication *guarantees proposition 2.3 (Simulation fidelity)*

Proof sketch. Let X be a $\mathcal{O}$-variable appearing in two different subterms s_1 and s_2. In s_1 variable X is applied to the list of terms $x_1 \dots x_m$ while, in s_2, it is applied to $y_1 \dots y_n$. Without loss of generality we take $m < n$. After *map-deduplication* we have two distinct $\mathcal{M}$-variables for X, say A and C, related by a chain $\vec{c}$ of η-`links` of length $n - m$. We prove that if A and C are assigned respectively to the $\mathcal{W}$ terms $\lambda x_1 \dots x_p.t_1$ and $\lambda x_1 \dots x_q.t_2$, then a failure occurs iff these terms do not unify. By *η-progress-lhs*, the assignment of A activates p links of $\vec{c}$. In particular, when $p \ge n$, then the entire chain collapses and unification is performed between $\lambda x_n \dots x_p.t_1$ and $\lambda x_1 \dots x_q.t_2$. By theorem 4.3 and since we work with $\mathcal{W}$ terms, this unification succeeds iff it does in $\mathcal{O}$. Otherwise if $p < n$ then we have to consider two cases. 1) If t_1 is a rigid term then t_2 must be the η-redex of t_1, implying $q = 0$. If it is not the case, then *η-progress-lhs* would eventually cause a failure. 2) If t_1 is a unification variable then `scope-check` guarantees that t_2 does not use $x_1 \dots x_p$ as free variables, causing otherwise a unification failure. After this check, $m - n - p$ links of $\vec{c}$ remain suspended, as we expect: decompilation is charged to wake all links in $\mathbb{L}$ and eventually relate t_1 with t_2. □

Example of map-deduplication. If we take back the example given at the beginning of this section, *map-deduplication* produces the following result:

$$\begin{array}{l}\mathbb{P} = \{\ \lambda x.(X\ x)\ \simeq_o\ f \qquad X\ \simeq_o\ \lambda x.a\ \}\\ \mathbb{Q} = \{\ \qquad A\ \simeq_m\ f \qquad C\ \simeq_m\ \lambda x.a\ \}\\ \mathbb{M} = \{\ X \mapsto B\ \}\\ \mathbb{L} = \{\ \vdash C =_\eta\ \lambda x.B_x \qquad \vdash A =_\eta\ \lambda x.B_x\ \}\end{array}$$

Note that $X \mapsto C^0$ disappears from $\mathbb{M}$ in favour of the auxiliary η-`link` $\vdash C =_\eta \lambda x.B_x$. The resolution of $\mathbb{Q}_1$ assignes a to A. This wakes up $\mathbb{L}_2$ by *η-progress-lhs*, assigning «$f\ x$» to B_x. The resolution of $\mathbb{Q}_2$ instantiates C to $\lambda x.a$, which, in turn triggers $\mathbb{L}_1$ by *η-progress-lhs*. In turn, this performs $\lambda x.a \simeq_m \lambda x.(f\ x)$ that fails as expected.

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

As observed in [17] it is worth handling terms in $\Diamond\mathcal{L}$ since, in practice, these terms often re-enter $\mathcal{L}$ at runtime.

In the following example problem $\mathbb{P}_2$, namely $X\ a \simeq_o a$, admits two different solutions: $\rho_1 = \{\ X \mapsto \lambda x.x\ \}$ and $\rho_2 = \{\ X \mapsto \lambda x.a\ \}$. The unification algorithm alone cannot chose the right one, since no solution is more general than the other, but for the first problem the choice is obvious, and in turn this choice makes $\mathbb{P}_2 \subseteq \mathcal{L}$:

$$\begin{array}{l}\mathbb{P} = \{\ X\ \simeq_o\ \lambda x.a \qquad (X\ a)\ \simeq_o\ a\ \}\\ \mathbb{Q} = \{\ A\ \simeq_m\ \lambda x.a \qquad (A\ a)\ \simeq_m\ a\ \}\\ \mathbb{M} = \{\ X \mapsto A\ \}\end{array}$$

In order to support this scenario we have to improve a little our compiler and progress routines. In particular $\mathbb{Q}_1$ generates $\sigma = \{\ A \mapsto \lambda x.a\ \}$ making $\sigma\mathbb{Q}_2$ equal to $(\lambda x.a)\ a \simeq_m a$ that $\simeq_m$ cannot solve since it lacks rules β_l and β_r.

To address this problem the compiler must recognize $\Diamond\mathcal{L}$ terms and replace them with fresh variables and generate a new kind of links that we call $\mathcal{L}$-`link`.

In addition to invariant 2 (Link left hand side), the term on the rhs of a $\mathcal{L}$-`link` has the following property:

Invariant 4 ($\mathcal{L}$-`link` rhs). *The rhs of any $\mathcal{L}$-`link` has the shape $X_{s_1 \dots s_n}\ t_1 \dots t_m$ where X is a unification variable in $\mathcal{L}$ (with scope $s_1 \dots s_n$) and $t_1 \dots t_m$ is a list of terms such that $m > 0$ and t_1 is either a variable occurring in $s_1 \dots s_n$ or a term other than a variable.*

Note that the shape of such as rhs is «`app [uva X S | L]`», where `S` is `[s1, ..., sn]` and `L` is `[t1, ..., tm]`.

7.1 Compilation and decompilation

We insert this rule just before rule ($c_@$):

```
comp (o-app [o-uva A|Ag]) (uva B Sc) M1 M3 L1 L3 S1 S4 :- !,
  pattern-fragment-prefix Ag Pf Extra, alloc S1 B S2,
  len Pf Ar, m-alloc (ov A) (mv C (arity Ar)) 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 Sc,
L3 = [val (link-llam (uva B Sc) Beta) | L2].
```

The list `Ag` is split into two parts: `Pf` is in $\mathcal{L}$, and can be empty; `Extra` cannot be empty and is such that «`append Pf Extra Ag`». The rhs of the $\mathcal{L}$-`link` is the application of a fresh variable `C` having in scope all names in `Pf1` (the compilation of `Pf`). The variable `B`, returned as the compiled term, is a fresh variable having in scope all the free variables occurring in `Pf1` and `Extra1` (the compilation of `Extra`). This construction enforces invariant 4 ($\mathcal{L}$-`link` RHS).

Lemma 7.1 ($\mathcal{W}$-enforcement). *$\forall s, \langle s \rangle \mapsto (t, m, l)$, then $\mathcal{W}(t)$.*

Proof sketch. By lemmas 4.2 and 5.7 and the rule above. □

Decompilation. All $\mathcal{L}$-`link` should be solved before decompilation. If any $\mathcal{L}$-`link` remains in $\mathbb{L}$, decompilation fails.

7.2 Progress

Let l be a $\mathcal{L}$-`link` of the form $\Gamma \vdash T =_{\mathcal{L}} X_{s_1 \dots s_n}\, t_1 \dots t_m$

Definition 7.2 ($\mathcal{L}$-progress-refine). Let σ be a substitution such that σt_1 is a name s not occurring in $s_1 \dots s_n$. If $m = 1$, then l is removed and the lhs is unified with $X_{s_1 \dots s_n\, s}$. If $m > 1$, then l is replaced by the link $\Gamma \vdash T =_{\mathcal{L}} Y_{s_1 \dots s_n\, s}\, t_2 \dots t_m$ (where Y is a fresh variable of arity $n+1$) and 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 7.3 ($\mathcal{L}$-progress-rhs). Link l is removed from $\mathbb{L}$ if $X_{s_1 \dots s_n}$ is instantiated to a term t and $t\, t_1 \dots t_m$ β-reduces to a $t' \in \mathcal{L}$.

Definition 7.4 ($\mathcal{L}$-progress-fail). progress fails when either

- there exists another link $l' \in \mathbb{L}$ with the same lhs as l; or
- the lhs of l become rigid.

We relax this condition and accommodate for heuristics in 7.3.

Finally we introduce the following η-`link` progress rule.

Definition 7.5 (η-progress-rhs). A link $\Gamma \vdash X =_\eta T$ is removed from $\mathbb{L}$ when either

- *maybe-eta* T does not hold (anymore), in this case X is unified with T; or
- T η-contracts to a term T' not starting with the `lam` constructor. In this case X is unified with T'

The idea behind this rule is to instantiate the lhs variable of a η-`link` whenever its rhs is for sure a $\mathcal{W}$ term.

Lemma 7.6. `progress` *terminates*

Proof sketch. Let $l = \Gamma \vdash T =_{\mathcal{L}} X_{s_1 \dots s_n}\, t_1 \dots t_m$ a $\mathcal{L}$-`link` in the store $\mathbb{L}$. The progression made by *$\mathcal{L}$-progress-rhs* terminates: l is removed from $\mathbb{L}$ after having performed terminating instructions. If l is not activated by *$\mathcal{L}$-progress-rhs*, then X is a variables. The activation of *$\mathcal{L}$-progress-refine* replaces l with the new $\mathcal{L}$-`link` $\Gamma \vdash T =_{\mathcal{L}} Y_{s_1 \dots s_n\, s}\, t_2 \dots t_m$. This progression can be triggered at most m times, since at each time the number of terms applied to X decreases. In the end l_m will be removed from $\mathbb{L}$ after the unification between lhs with rhs. We also note that each *$\mathcal{L}$-progress-refine* generates a η-`link`, yet, according to lemma 5.14, this does not affect termination. *$\mathcal{L}$-progress-fail* terminates since it stop progression with failure. Finally, *η-progress-rhs* performs terminating operations and, if its permises succeed, the considered η-`link` is removed from $\mathbb{L}$, ensuring termination. □

Theorem 7.7 (Fidelity in $\mathcal{O}$). *The introduction of $\mathcal{L}$-`link` guarantees proposition 2.4 (Fidelity recovery) if $\mathbb{P} \not\subseteq \mathcal{L}$.*

Proof sketch. Let $\mathbb{P}_i$ be the first problem in $\mathbb{P}$ such that it is not in $\mathcal{L}$ and let σ be the substitution obtained solving $\mathbb{Q}_1 \dots \mathbb{Q}_{i-1}$. After the execution of step_m for the $i-1$ time, each variable X corresponding to a $\diamond\eta$ subterm in the original $\mathcal{O}$ unification problem, is instantiated iff it is known for sure that X is no more in $\diamond\eta$. If $\sigma\mathbb{Q}_i$ is in $\mathcal{L}$, then by definitions 7.2 and 7.3 the associated $\mathcal{L}$-`link` is solved and removed. In this case, all calls to $\simeq_m$ are between terms in $\mathcal{L}$ and by theorem 4.3 fidelity is guaranteed. If $\sigma\mathbb{Q}$ is still in $\diamond\mathcal{L}$, then by definition 7.4 unification fails as the corresponding unification in $\mathcal{O}$ would (it is called outside its domain). □

7.3 Relaxing definition 7.4 ($\mathcal{L}$-*progress-fail*)

Working with terms in $\mathcal{L}$ is sometime too restrictive [1] and we could find in literature a few strategies to go beyond $\mathcal{L}$ without implementing Huet's algorithm [11]. Some implementations of λProlog [18] such as Teyjus [17] delay the resolution of $\diamond\mathcal{L}$ unification problems until the substitution makes them reenter $\mathcal{L}$. Other systems apply heuristics like preferring projection over mimic, and commit to that solution. This is the case for the unification algorithm Coq uses in its type-class solver [25],

In this section we show how we can implement these strategies by simply adding (or removing) rules to the `progress` predicate. In the example below $\mathbb{P}_1$ is in $\diamond\mathcal{L}$.

$$\begin{aligned}
\mathbb{P} &= \{\ (X\ a) \simeq_o a \quad X \simeq_o \lambda x.Y\ \} \\
\mathbb{Q} &= \{\ \quad A \simeq_m a \quad B \simeq_m \lambda x.C\ \} \\
\mathbb{M} &= \{\ Y \mapsto C \quad X \mapsto B\ \} \\
\mathbb{L} &= \{\ \vdash A =_{\mathcal{L}} (B\ a)\ \}
\end{aligned}$$

If we want the object-language unification to delay the first unification problem (waiting for X to be be instantiated), we can relax definition 7.4. Instead of failing when the lhs of $\mathbb{L}_1$ becomes rigid (equal to a), we keep it in $\mathbb{L}$ until the head of its rhs also become rigid. In this case, since both the lhs and rhs have rigid heads, they can be unified. While this relaxed rule does not break proposition 2.3 (Simulation fidelity) per se, the `occur-check-links` procedure becomes incomplete since invariant 2 (Link left hand side) is broken. Also note that delaying unification outside $\mathcal{L}$ can leave $\mathcal{L}$-`link` for the decompilation phase. Therefore `commit-links` should be modified accordingly.

If instead we want $\simeq_o$ to follow the second strategy and pick an arbitrary solution we can modify progress by applying the desired heuristic instead of failing. For instance, in $X\ a\ b = Y\ b$, the last argument of the two terms is the same and unification can succeed by assigning $X\,a$ to Y. This heuristic is used by [25].

8 ACTUAL IMPLEMENTATION IN ELPI

In this paper we did study a compiler and a simulation loop on a minimal language. The actual implementation uses the Coq-Elpi meta language to both compile the sequence of problems (the rules) and execute them, that is Elpi plays the role of $\mathcal{M}$ as well.

The main difference is that we cannot implement run_m since it is the runtime of the programming language. In particular the runtime iterates $\simeq_m$, but step_m also needs to check links for progress. Luckily Elpi extends λProlog with constraints (suspended goals) and Constraint Handling Rules (CHR) to operate on them [10, 7]. Similarly to [13] a constraint is a goal suspended on a list of variables that is resumed *as soon as one of these variable is assigned*, before any other existing goal is considered. In turn link activation grants proposition 2.3 (Simulation fidelity). We point out that [13] delays $\diamond\mathcal{L}$ unification problems while we also delay problems that are in $\diamond\eta$. For brevity we only provide two pseaudo-code snippets. The first one depicts how η-link are suspended or make progress.

```
link-eta L R :- not (var L), !, eta-progress-lhs L R.
link-eta L R :- not (maybe-eta R), !, eta-progress-rhs L R.
link-eta L R :- declare_constraint (link-eta L R) [L,R].
```

The second snippet illustrates the deduplication of η-link. The syntax «N ▹ G ?- P» denotes a λProlog sequent, that is a goal P under a set G of hypothetical rules (introduced by =>) and where the program context binds (via the pi operator) eigenvariables in N. Sequents are presented to CHR with their higher order unification variables replaced by "frozen constants", that is A_{xyz} becomes «uvar f_A [x,y,z]» for a fresh constant f_A. Frozen constants are "defrost" when they are part of a new goal (see [10, section 4.3]).

```
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
```

The first directive matches a constraint whose lhs is a variable X; the second matches and removes a constraint on the same variable; the third relocates T2 and the latter crafts the new goal that unifies T1 with T2' under G1 and the set of heigenvariables N1.

9 RELATED WORK AND CONCLUSION

Object-language terms can be unified using different strategies.

The first approach that comes to mind consist in implementing $\simeq_o$ as a regular routine, i.e. write rules as follows

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

where unify is a regular predicate. This method fails to take advantage of the logic programming engine provided by the meta language since it removes all the data from the rule's heads and makes indexing degrade. Additionally, implementing a unification procedure in the meta language is likely to be significantly slower compared to the built-in one on the common domain of first order terms.

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

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

However, this encoding has two big limitations. First it is not always feasible to adopt it for Coq due to the fact that the type system of the meta language is too limited to accommodate the one of the object language, e.g. Coq can typecheck variadic functions [2]. Second, the encoding of Coq terms provided by Elpi is primarily used for meta programming, i.e. 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/3 and arg/3. To avoid that constants cannot be symbols of the meta language, but must rather live in an open world, akin to the string data type used in the con constructor.

In the literature we could find a related encoding of the Calculus of Constructions (CC) [4]. The goal of that work is to exhibit a logic program performing proof checking for CC and hence relate the proof system of intuitionistic higher-order logic (that animates λProlog 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, and that choice alone makes things harder for us. 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 are not interested in exploiting a decidable fragment of higher-order unification but are rather leaning towards an interactive system where the user is given control over the search procedure.

Another work that is, somewhat surprisingly, only superficially related to ours is the type-class engine built in Isabelle's meta language. In [31] classes identify (simple) types, they are not higher order predicates as in Coq, hence the solver does not require a higher-order unification procedure.

The approach presented in this paper addresses all the concerns mentioned earlier. It takes advantage of the unification capabilities of the meta language at the price of handling problematic sub terms on the side. As a result our encoding takes advantage of indexing data structures and mode analysis for clause filtering. It is worth mentioning that we replace terms with variables only when it is strictly needed, leaving the rest of the term structure intact and hence indexable. Some preliminary benchmarks on the Stdpp library [12] show encouraging results: our type-class solver has some overhead on small goals but is consistently faster than the Coq one starting from goals made of 32 nodes. Moreover our approach is flexible enough to accommodate different strategies and heuristics to handle terms outside the pattern fragment. Finally, our encoding is sufficiently shallow to: 1) allow to lift forms of static analysis for the meta language, such as determinacy, also to the object language; 2) take advantage of future improvements to the logic-programming engine of Elpi, such as tabled search.

We considered mechanizing our results with Abella [8], especially in the early phase of this work, but unfortunately we could not. The main reason is that the logic of Abella does not let one quantify over predicates nor use the cut operator. Eliminating cut or higher order combinators like map or forall2 is surely possible, but has a high cost in verbosity given how pervasively they are used. Moreover, and more importantly, it creates a distance between the verified and the actual code, which we found to hinder our exploration. However, we did establish a test suite with about a hundred cases. It is available at https://github.com/FissoreD/ho-unif-for-free.

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] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN: 0262026651.

[3] Cvetan Dunchev et al. "ELPI: Fast, Embeddable, λProlog Interpreter". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis et al. Vol. 9450. 2015, pp. 460–468. DOI: 10.1007/978-3-662-48899-7_32. URL: http://dx.doi.org/10.1007/978-3-662-48899-7%5C_32.

[4] Amy Felty. "Encoding the Calculus of Constructions in a Higher-Order Logic". In: ed. by M. Vardi. IEEE, June 1993, pp. 233–244. DOI: 10.1109/LICS.1993.287584.

[5] Amy Felty and Dale Miller. "Specifying theorem provers in a higher-order logic programming language". In: *Ninth International Conference on Automated Deduction*. Ed. by Ewing Lusk and Ross Overbeck. 310. Argonne, IL: Springer, May 1988, pp. 61–80. DOI: 10.1007/BFb0012823.

[6] Davide Fissore and Enrico Tassi. "A new Type-Class solver for Coq in Elpi". In: *The Coq Workshop 2023*. Bialystok, Poland, July 2023. URL: https://inria.hal.science/hal-04467855.

[7] Thom Fruehwirth. *Constraint Handling Rules - What Else?* 2017. arXiv: 1701.02668 [cs.PL].

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

[9] Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. "Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin's data types with Coq-Elpi". In: *CPP '23: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. Boston, MA, USA: ACM, Jan. 2023, pp. 167–181. DOI: 10.1145/3573105.3575683. URL: https://inria.hal.science/hal-03800154.

[10] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. "Implementing type theory in higher order constraint logic programming". In: *Mathematical Structures in Computer Science* 29.8 (2019), pp. 1125–1150. DOI: 10.1017/S0960129518000427.

[11] Gérard P. Huet. "A Unification Algorithm for Typed lambda-Calculus". In: *Theor. Comput. Sci.* 1.1 (1975), pp. 27–57. DOI: 10.1016/0304-3975(75)90011-0. URL: https://doi.org/10.1016/0304-3975(75)90011-0.

[12] 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). DOI: 10.1017/S0956796818000151. URL: https://api.semanticscholar.org/CorpusID:2023423.

[13] Spiro Michaylov and Frank Pfenning. "Higher-Order Logic Programming as Constraint Logic Programming". In: *Principles and Practice of Constraint Programming*. 1993. URL: https://api.semanticscholar.org/CorpusID:9980455.

[14] Dale Miller. "A logic programming language with lambda-abstraction, function variables, and simple unification". In: *Extensions of Logic Programming*. Ed. by Peter Schroeder-Heister. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 253–281. ISBN: 978-3-540-46879-0.

[15] 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.

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] Frank Pfenning and Conal 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.

[23] Frank Pfenning and Carsten Schürmann. "System Description: Twelf — A Meta-Logical Framework for Deductive Systems". In: *Automated Deduction — CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2.

[24] Brigitte Pientka and Jana Dunfield. "Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)". In: *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. Ed. by Jürgen Giesl and Reiner Hähnle. Vol. 6173. Lecture Notes in Computer Science. Springer, 2010, pp. 15–21. DOI: 10.1007/978-3-642-14203-1_2. URL: https://doi.org/10.1007/978-3-642-14203-1%5C_2.

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

[26] 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, OR, United States, Sept. 2019. DOI: 10.4230/LIPIcs.CVIT.2016.23. URL: https://inria.hal.science/hal-01897468.

[27] Enrico Tassi. "Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)". In: *The Fourth International Workshop on Coq for Programming Languages*. Los Angeles, CA, United States, Jan. 2018. URL: https://inria.hal.science/hal-01637063.

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

[29] 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.

[30] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. "The Isabelle Framework". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7.

[31] Markus Wenzel. "Type classes and overloading in higher-order logic". In: *Theorem Proving in Higher Order Logics*. Ed. by Elsa L. Gunter and Amy Felty. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 307–322. ISBN: 978-3-540-69526-4.