Higher-Order unification for free!
Davide Fissore, Enrico Tassi

To cite this version:
Davide Fissore, Enrico Tassi. Higher-Order unification for free!: Reusing the meta-language unification for the object language. 2024. hal-04547069v4

HAL Id: hal-04547069
https://inria.hal.science/hal-04547069v4
Preprint submitted on 30 Jul 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
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 βη-equivalence and features a higher-order unification procedure ≃_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 ≃_m to first-order unification problems only. We refer to this basic encoding as O.

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

KEYWORDS

Logic Programming, Meta-Programming, Higher-Order Unification

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-known 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.

\[
\text{Instance } \text{fin} \text{fin}: \forall n, \text{Finite} (\text{fin } n). \\
\text{Instance } \text{nfact} \text{dec}: \forall n, \text{Decision} (\text{nfact } n \text{ nf}). \\
\text{Instance } \text{forall} \text{dec}: \forall A \text{, } \text{Finite } A \rightarrow \text{Decision } (\forall x : A, \text{Decision } (P x)) \\
\rightarrow \text{Decision } (\forall x : A, P x).
\]

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

\[
\text{Decision } (\forall x : \text{fin } 7, \text{nfact } x \ 3)
\]

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 \( P x = \text{nfact } x \ 3 \). This higher-order problem falls in the so-called pattern fragment and admits a unique solution:

\[
P = \lambda x. \text{nfact } x \ 3
\]

In the rest of this paper, we refer to the pattern fragment as \( L \) [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 \( \mathsf{tm} \), that includes (among others) the following constructors:

\begin{itemize}
\item type \( \mathsf{lam} \ \mathsf{tm} \rightarrow (\mathsf{tm} \rightarrow \mathsf{tm}) \rightarrow \mathsf{tm} \). % lambda abstraction
\item type \( \mathsf{app} \ \mathsf{list} \ \mathsf{tm} \rightarrow \mathsf{tm} \). % n-ary application
\item type \( \mathsf{all} \ \mathsf{tm} \rightarrow (\mathsf{tm} \rightarrow \mathsf{tm}) \rightarrow \mathsf{tm} \). % forall quantifier
\item type \( \mathsf{con} \ \mathsf{string} \rightarrow \mathsf{tm} \). % constants
\end{itemize}

Following the standard syntax of \( \lambda \)Prolog [16], the meta-level binding of a variable \( x \) in an expression \( e \) is written as \( \langle x \ \mathsf{\lambda} \ e \rangle \), while square brackets delimit a list of terms separated by comma. For example, the term \( \langle \mathsf{con} \ \mathsf{nfact}\mathsf{nfact} \mathsf{nfact} \mathsf{nfact} \mathsf{nfact} \mathsf{nfact} \mathsf{nfact} \rangle \) is encoded as follows:

\[
\text{all} \ (\mathsf{con} \mathsf{nfact}), \ y \ \mathsf{app} \ (\mathsf{con} \mathsf{nfact}), \ y, \ \mathsf{con} \mathsf{nfact}
\]

We now illustrate a naïve 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 \( \Rightarrow \) introduces a fresh nominal constant \( w \) for the premise \( p \).

\[
\begin{align*}
\text{finite} \ &\ (\mathsf{app} \ [\mathsf{fin}], \ N). \quad (r1) \\
\text{decision} \ &\ (\mathsf{app} \ [\mathsf{nfact}], \ N, \ NF). \quad (r2) \\
\text{decision} \ &\ (\mathsf{all} \ A \ x \ \mathsf{\lambda} \ \mathsf{app} [P, x]) \ : \ \text{finite} \ A, \ (r3) \\
\text{pi} \ &\ w \ \mathsf{\lambda} \ (\mathsf{app} [P, w]).
\end{align*}
\]

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 \( \mathsf{tm} \). If we try to back-chain the rule \( (r3) \) on the encoding of the goal \( g \) given here:

\[
\begin{align*}
\text{decision} \ &\ (\mathsf{all} \ (\mathsf{app} \ [\mathsf{fin}], \ \mathsf{con} \mathsf{nfact})), \ x \ \mathsf{\lambda} \ \mathsf{app} [P, x]. \quad (g) \\
\text{app} \ &\ [\mathsf{nfact}], \ x, \ \mathsf{con} \mathsf{nfact}].
\end{align*}
\]

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

\[
\begin{align*}
\text{app} \ &\ [\mathsf{nfact}], \ x, \ \mathsf{con} \mathsf{nfact}]. = \text{app} [P, x]. \quad (p)
\end{align*}
\]

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:

\[
\begin{align*}
\text{decision} \ &\ (\mathsf{all} \ A \ x \ \mathsf{\lambda} \ \mathsf{app} \ (P \ A), \ \text{finite} \ A, \ (r3’), \ \pi) \ x \ \mathsf{\lambda} \ (\mathsf{app} [P, w]).
\end{align*}
\]

Since \( P \) is a higher-order unification variable, this encoding \( p \) introduces one solution:

\[
\begin{align*}
\text{app} \ &\ [\mathsf{nfact}], \ x, \ \mathsf{con} \mathsf{nfact}]. = \text{app} [P, x]. \quad (p’)
\end{align*}
\]

Once the head of rule \( (r3’) \) unifies with the goal \( g \), the premise \( \langle \mathsf{link} \ P \ A \ P \rangle \) brings the assignment \( (\pi) \) back to the domain \( \mathsf{tm} \) of Coq terms, obtaining the expected solution \( \pi) \) :

\[
\begin{align*}
P = \mathsf{lam} \ A \ x \ \mathsf{app} [\mathsf{nfact}], \ x, \ \mathsf{con} \mathsf{nfact}].
\end{align*}
\]

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 \( \pi \) \). In turn, this redex prevents rule \( (r2) \) from backchaining properly since the following unification problem has no solution:

\[
\begin{align*}
\text{app} \ &\ [\mathsf{lam} \ A \ (\mathsf{\lambda} \ \mathsf{app} [\mathsf{nfact}], \ a, \ \mathsf{con} \mathsf{nfact}]), \ x] = \\
\text{app} \ &\ [\mathsf{con} \mathsf{nfact}], \ x, \ N, \ NF]
\end{align*}
\]

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 \( \eta \)-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 \( L \). 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.

\section{Contributions}

In this paper we identify a minimal language \( O \) in which the problems sketched in the introduction can be formally described. We detail an encoding of a logic program in \( O \) to a strongly related logic program in \( M \), 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 \( \lambda \)Prolog code discussed in the paper can be accessed at the address \url{https://github.com/FissoreD/ho-unif-for-free}.

\section{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 \( O \) with a rich equational theory and a \( M \) language with a simpler one.

\subsection{Preliminaries: \( O \) and \( M \)}

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

\begin{figure*}[h]
\centering
\begin{itemize}
\item kind to type.
\item kind tm type.
\item type o-app list to -> to.
\item type app list tm -> tm.
\item type o-lam (to -> to) -> to.
\item type lam (tm -> tm) -> tm.
\item type o-con string to.
\item type con string to.
\item type o-uva addr to.
\item type uva addr to list tm -> tm.
\end{itemize}
\caption{The \( O \) and \( M \) languages}
\end{figure*}

Notational conventions. When we write \( M \) 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 applications. Here are few examples:

\[
f a \quad \text{app ["f", } \text{con } \text{"a"]}
\]

\[
\lambda x.\lambda y. F_{xy} \quad \text{lam } \lambda \text{} l a m \ y \ \lambda \text{} y u v a \ F \ [x, y]
\]

\[
\lambda x. F_{x} \ a \quad \text{lam } \lambda \text{} l a m \ a \ \text{app } [\text{uva } F \ [x], \ \text{con } \text{"a"]}
\]

\[
\lambda x. F_{x} \ x \quad \text{lam } \lambda \text{} l a m \ x \ \text{app } [\text{uva } F \ [x], \ x]
\]

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

### 2.2 Equational theories and unification

#### 2.2.1 Term equality: \( \simeq_o \) and \( \simeq_m \)

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 \( \sigma \)-equivalence. In addition to that, \( \simeq_m \) has rules for \( \eta \) and \( \beta \)-equivalence.

- **type \( (\simeq_o) \) to \( \rightarrow \) to \( \rightarrow \) o.**
  - \( \circ-con \ X = \circ-con \ X. \)
  - \( \circ-app \ A = \circ-app \ B \quad \text{for all2 \( ( \simeq_m) \) A B.} \)
  - \( \circ-lam \ F = \circ-lam \ G \quad \text{pi } x \ \lambda \ x = \circ \ \pi \ F = \circ \ G \ x. \)
  - \( \circ-uva \ N = \circ-uva \ N. \)
  - \( \circ-lam \ F = \circ-lam \ T \quad \ (\eta_1) \)
  - \( \pi \ x \ \lambda \ beta \ T \ [x] \ (T' \ x), \ x = \circ \ x \ \Rightarrow \ F = \circ \ T' \ x. \)
  - \( T = \circ \ o-lam \ F \quad \ (\eta_2) \)
  - \( \pi \ x \ \lambda \ beta \ T \ [x] \ (T' \ x), \ x = \circ \ x \ \Rightarrow \ T' = \circ \ F \ x. \)
  - \( o-app \ o-lam \ X[I] = \circ \ T \quad \text{beta (o-lam \ X) L R, R = \circ} T. \)
  - \( T = \circ \ o-app \ o-lam \ X[I] \quad \text{beta (o-lam \ X) L R, T = \circ} R. \)

- **type \( (\simeq_m) \) tm \( \rightarrow \) tm \( \rightarrow \) o.**
  - \( \text{con C = \circ-con C.} \)
  - \( \text{app A = \circ-app B \quad \text{for all2 \( ( \simeq_m) \) A B.} \)
  - \( \text{lam F = \circ-lam G \quad \text{pi } x \ \lambda \ x = \circ \ F = \circ} \ G \ x. \)
  - \( \text{uva N = \circ-uva N.} \)

Note that the symbol \( \mid \) separates the head of a list from the tail; that for all2 applies a binary predicate to two lists (of equal length); and that \( \Rightarrow \) is \( \beta \)Prolog 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 \( \simeq_m \) is, and to identify the four rules that need special treatment in the implementation of \( \simeq_m \). For brevity, we omit the code of beta: it is sufficient to know that \( \beta \)-in \( F.L \). \( \Rightarrow \) computes in it the weak-head normal form of \( \circ-app \ [P/L]. \)

Substitution: \( ps \) and \( et \)

We write \( \sigma = \{ X \Rightarrow t \} \) for the substitution that assigns the term \( t \) to the variable \( X \). We write \( et \) for the application of the substitution to a term \( t \), and \( \sigma X = \{ t \mid t \in X \} \) when \( X \) is a set of terms. We write \( \sigma \subseteq \sigma' \) when \( \sigma \) is more general than \( \sigma' \). We shall use \( \rho \) for \( O \) substitutions, and \( \sigma \) for the \( M \) ones. For brevity, in this section, we consider the substitution for \( O \) and \( M \) identical. We defer to section 3.1 a more precise description pointing out their differences.

**Term unification:** \( \simeq_o \) vs. \( \simeq_m \)

We write \( \sigma_1 \simeq_m \sigma_2 \Rightarrow \sigma' \) when \( \sigma_1 \) and \( \sigma_2 \) 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 \( t_1 \simeq_m t_2 \Rightarrow \sigma' \) when the initial substitution \( \sigma \) is empty. We write \( L \) 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 \( D \) is the following:

**Proposition 2.1 (Good unification).** A good unification \( \sim \) for equality \( = \) in the domain \( D \) satisfies the following properties:

\[
\begin{align*}
(1) \quad & t_1, t_2 \subseteq D \Rightarrow t_1 \equiv t_2 \Rightarrow \rho \Rightarrow \rho t_1 \equiv \rho t_2 \\
(2) \quad & t_1, t_2 \subseteq D \Rightarrow \rho t_1 \equiv \rho t_2 \Rightarrow \exists \rho', t_1 \equiv t_2 \Rightarrow \rho' \wedge \rho' \subseteq \rho
\end{align*}
\]

The meta language of choice is expected to provide an implementation of \( \simeq_m \) that is a good unification for \( \simeq_m \) in \( L \).

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 \( O \) as a sequence of \( n \) steps. Each step \( p \) consists in unifying two terms, \( \mathbb{P}_p \) and \( \mathbb{P}_p' \), taken from the list of all unification problems \( \mathbb{P} \). We write \( \mathbb{P} = \{ \mathbb{P}_1, \mathbb{P}_2, \ldots \} \) and \( s \in \mathbb{P} \Rightarrow \exists \mathbb{P}_s, s \in \mathbb{P}_p \). 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.

\[
\begin{align*}
\text{step}_1(\mathbb{P}, p, \rho, \rho) & \mapsto \rho_0 \text{ def } = \mathbb{P}_p \rho_0 \mathbb{P}_p \Rightarrow \rho' \\
\text{run}_n(\mathbb{P}, n) & \text{ def } = \land^n_{p=1} \text{step}_p(\mathbb{P}, p, \rho_{p-1}) \Rightarrow \rho_p
\end{align*}
\]

In order to simulate a logic program we start by compiling \( \mathbb{P} \) to \( Q \), i.e. each \( O \)-term \( s \in \mathbb{P} \) is translated to a \( M \)-term \( t \in Q \). We write this translation as \( (s) \mapsto (t, m, l) \). 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 \( M \) to variables in \( O \) and is used to “decode” the substitution, that is written \( \langle \sigma, m, l \rangle \mapsto m \). Links are an accessory piece of information whose description is deferred to section 2.4.

Then we simulate each run in \( O \) with a run in \( M \) as follows:

\[
\begin{align*}
\text{step}_m(Q, p, \sigma, L) & \mapsto (\sigma', L') \text{ def } = \sigma'_{Q_{L_p}} = \sigma'_{Q_{L_p}} \sigma' \wedge \text{progress}(L, \sigma') \Rightarrow (L', \sigma') \\
\text{run}_n(Q, n) & \text{ def } = \land^n_{p=1} \text{step}_m(Q, p, \rho_{p-1}, L_{p-1}) \Rightarrow (\sigma_p, L_p) \\
(\sigma_m, M, L_n) & \mapsto \rho_n
\end{align*}
\]
By analogy with \( P \), we write \( Q_p \) and \( Q_p^\prime \) for the two \( M \) terms being unified at step \( p \), and we write \( Q_p \) for the set \( \{ Q_p, Q_p^\prime \} \). The step \( p \) 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 \( \text{run}_m \) compiles all terms in \( P \), then executes each step, and finally decompiles the solution. We claim:

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

\[
\text{run}_o(\bar{P}, n) \rightarrow p \land \text{run}_m(\bar{P}, n) \rightarrow p^\prime \Rightarrow \forall s \in P, ps \simeq_o p^\prime s
\]

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

**Proposition 2.3 (Simulation fidelity).** In the context of \( \text{run}_o \) and \( \text{run}_m \), if \( P \subseteq L \) we have that \( \forall p \in 1 \ldots n, \)

\[
\text{step}_o(\bar{P}, p, \rho_p) \rightarrow p \Leftrightarrow \text{step}_m(\bar{P}, p, \rho_p) \rightarrow (p, \rho_p)
\]

In particular, this property guarantees that a failure in the \( O \) run is matched by a failure in \( M \) 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 \( M \).

We also claim that \( \text{run}_m \) handles terms outside \( L \) in the following sense:

**Proposition 2.4 (Fidelity recovery).** In the context of \( \text{run}_o \) and \( \text{run}_m \), if \( P \subseteq L \) (even if \( P \not\subseteq L \)) then

\[
\text{step}_o(\bar{P}, p, \rho_p) \rightarrow p \Leftrightarrow \text{step}_m(\bar{P}, p, \rho_p) \rightarrow (p, \rho_p)
\]

In other words, if the two terms involved in a step re-enter \( L \), then \( \text{step}_o \) and \( \text{step}_m \) are again related, even if \( P \not\subseteq L \) 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 \( F \equiv \lambda x. a \) and \( F \equiv \lambda a \): the second problem is not in \( L \) and has two unifiers, namely \( \sigma_1 = \{ F \mapsto \lambda x. a \} \) and \( \sigma_2 = \{ F \mapsto \lambda a \} \). The first problem picks \( \sigma_2 \), making the second problem re-enter \( L \).

**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 \( \text{run} \) 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 accessor 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 \( \simeq_o \) 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 \( F \equiv x_1 \ldots x_n \) 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 \( \lambda \), or a constant, or a name, or remain an application.

**Definition 2.6 (\( \Box \eta \)).** \( \Box \eta \) is the set of terms \( \lambda x. s \) such that \( \exists p, \rho (\lambda x. s) \) is an \( \eta \)-redex.

An example of a term \( s \) in \( \Box \eta \) is \( \lambda x. \lambda y. F \) where \( F \) is a term whose substitution \( \rho = \{ F \mapsto \lambda x. \lambda y. F, y, x \} \) makes \( ps =_o \lambda x. \lambda y. F \) for the two \( \eta \)-long form of \( F \). This term is problematic since its leading \( \lambda \) abstraction cannot justify a unification failure against a constant \( f \).

**Definition 2.7 (\( \Box L \)).** \( \Box L \) is the set of terms of the form \( F \equiv t_1 \ldots t_n \) 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 \( L \). Still, given \( s \in \Box L \) there may exist a substitution \( \rho \) such that \( ps \in L \).

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

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

\[
W(X) \equiv \forall t \in P(X), t \notin (\Diamond \beta \cup \Box \eta \cup \Box L)
\]

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

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

\[
(s) \mapsto (t, m, l) \Rightarrow W(t)
\]

**Proposition 2.10 (Compiler-useful).** Given two terms \( s_1 \) and \( s_2 \), if \( \exists p, ps_1 =_o ps_2 , then \)

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

In other words the compiler outputs terms in \( W \) 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 \( W \) the meta-language unification \( \simeq_m \) agrees with the object-language equality \( \simeq_o \). Note that a compiler that translates all terms to unification variables would satisfy this property, but would certainly fail to verify proposition 2.3.

**Proposition 2.11 (W-preservation).** \( \forall Q, \forall L, \forall p, \forall \sigma, \forall \sigma' \)

\[
W(\sigma Q) \land \sigma Q_p \simeq_m Q \sigma_p \Rightarrow \sigma' \Rightarrow W(\sigma' Q)
\]

\[
W(\sigma Q) \land \text{progress}(L, \sigma) \Rightarrow (\sigma, \sigma') \Rightarrow W(\sigma' Q)
\]

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 \( \text{step}_m \), hence \( \simeq_m \) can continue...
to operate properly. In sections 4, 5 and 7 we describe how to recognize terms in $\Diamond \eta$, $\Diamond \beta$ and $\Diamond L$ and how progress takes care of links and how it preserves ‘$W$’ 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 $\rho_L$ and $\rho_P$; they are equivalent only by using decompilation to take into account terms in $L$. 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 ($M$) and substitution ($\rho$ and $\sigma$)

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

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem A</td>
<td>list (option A)</td>
</tr>
<tr>
<td>addr -&gt; mem A</td>
<td>A -&gt; o</td>
</tr>
<tr>
<td>unset? addr -&gt; mem A</td>
<td>o</td>
</tr>
<tr>
<td>assign addr -&gt; mem A</td>
<td>A -&gt; mem A</td>
</tr>
<tr>
<td>new mem A</td>
<td>addr -&gt; mem A</td>
</tr>
</tbody>
</table>
```

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 $M$ 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.

```plaintext
<table>
<thead>
<tr>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inctx type</td>
<td>type</td>
</tr>
<tr>
<td>abs (tm</td>
<td>inctx A)</td>
</tr>
<tr>
<td>val A</td>
<td>A</td>
</tr>
<tr>
<td>subst (mem assignment)</td>
<td>mem assignment</td>
</tr>
</tbody>
</table>
```

A solution to a $O$ variable is, instead, a plain term, that is, $fsubst$ is an abbreviation for `mem`. The compiler establishes a mapping between variables of the two languages.

```plaintext
<table>
<thead>
<tr>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inctx type</td>
<td>type</td>
</tr>
<tr>
<td>ov addr</td>
<td>ovariable</td>
</tr>
<tr>
<td>mvariable type</td>
<td></td>
</tr>
<tr>
<td>mv addr</td>
<td>arity</td>
</tr>
<tr>
<td>mapping type</td>
<td></td>
</tr>
<tr>
<td>(&lt;&gt; ) ovariable</td>
<td>mvariable</td>
</tr>
<tr>
<td>mmap (list mapping)</td>
<td></td>
</tr>
</tbody>
</table>
```

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 $x$ in $M$ has a (unique) arity $N$ and each occurrence (uva A L) is such that L has length $N$.

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-alloc ovariable</td>
<td>mvariable</td>
</tr>
<tr>
<td>m-alloc Ov Mv M M M S</td>
<td>:=</td>
</tr>
<tr>
<td>m-alloc Ov Mv M [Ov &lt;&gt; Mv]</td>
<td>M</td>
</tr>
</tbody>
</table>
```

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 $\eta$-link; this detail is discussed in section 6.

It is worth examining the code of $deref$, that applies the substitution to a $M$ 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.

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deref subst -&gt; tm -&gt; tm -&gt; o</td>
<td></td>
</tr>
<tr>
<td>deref _ (con C) (con C)</td>
<td></td>
</tr>
<tr>
<td>deref S (app A) (app B) := map (deref S) A B</td>
<td></td>
</tr>
<tr>
<td>deref S (lam F) (lam G) := pi x\ deref S x x \Rightarrow deref S (F x) (G x)</td>
<td></td>
</tr>
<tr>
<td>deref S (uva N L) R := set? N S A</td>
<td></td>
</tr>
<tr>
<td>move A L T, deref S T R</td>
<td></td>
</tr>
<tr>
<td>deref S (uva N A) (uva N B) := unset? N S</td>
<td></td>
</tr>
<tr>
<td>map (deref S) A B</td>
<td></td>
</tr>
</tbody>
</table>
```

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.

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>move assignment -&gt; list tm -&gt; tm -&gt; o</td>
<td></td>
</tr>
<tr>
<td>move (abs Bo) [H</td>
<td>L] R := move (Bo H) R</td>
</tr>
<tr>
<td>move (val A) [] A := !</td>
<td></td>
</tr>
</tbody>
</table>
```

We write $\sigma = \{A_x \mapsto y\}$ for the assignment «abs x\abs y\y» and $\rho = \{A \mapsto \lambda x.\lambda y. y\}$ for «\lam x\lam y\y».

3.2 Links ($L$)

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

```plaintext
<table>
<thead>
<tr>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>baselink type</td>
<td></td>
</tr>
<tr>
<td>link-eta tm -&gt; tm -&gt; baselink</td>
<td></td>
</tr>
<tr>
<td>link-llam tm -&gt; tm -&gt; baselink</td>
<td></td>
</tr>
<tr>
<td>mmap (list baselink)</td>
<td></td>
</tr>
</tbody>
</table>
```

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 `inctx` 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 $x \mapsto A_x = L F x a$ corresponds to:

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs x\val (link-llam (uva A [x]) (app[uva F [x], con &quot;a&quot;]))</td>
<td></td>
</tr>
</tbody>
</table>
```

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 $M$ to $O$ the compiler builds a “memory map” connecting the $O$ variables to the $M$ ones.

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

\[
\text{type comp to -> tm -> mmap -> mmap -> links -> links -> subst -> subst -> o.}
\]

\[
\text{comp (o-con C) (con C) M M L L S S}.
\]

\[
\text{comp (o-lam F) (lam F1) M1 M2 L1 L2 S1 S2} : (c_j)
\]

\[
\text{comp-lam F F1 M1 M2 L1 L2 S1 S2.}
\]

\[
\text{comp (o-app A) (app A1) M1 M2 L1 L2 S1 S2}.
\]

\[
\text{fold6 comp A A1 M1 M2 L1 L2 S1 S2.}
\]

\[
\text{type compile to -> tm -> mmap -> links -> links -> subst -> subst -> o.}
\]

\[
\text{compile F G M1 M2 L1 L2 S1 S2}.
\]

\[
\text{beta-normal F F'}.
\]

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).

\[
Proposition 3.1 (Compilation round trip). If $s \mapsto (t, m, l)$ and $l \in L$ and $m \in M$, and $\sigma = [A \mapsto t]$ and $X \mapsto A^n \in M$ then $(\sigma, M, L)^{-1} \mapsto \rho$ and $pX \approx_\rho ps$.

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

3.6 Definition of $\approx_o$ and its properties

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

**type** $\approx_o$ to $\mapsto$ to $\mapsto$ fsusb $\rightarrow$ o.

$\approx_o$ is defined as follows:

- compile $A \approx_o B$ $F$ $:=$
  - compile $A$ $A'$ $\square$ $M_1$ $\square$ $L_1$ $\square$ $S_1$,
  - compile $B$ $B'$ $M_2$ $L_2$ $S_2$ $S_2$,
  - hstep $A'$ $B'$ $L_3$ $L_3$ $S_3$ $S_3$,
  - decompile $M_2$ $L_3$ $S_3$ $\square$ $F$.

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 $\approx_o$ 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 $\approx_o$ in $\mathcal{W}$).** The implementation of $\approx_o$ above is a good unification for $\approx_\sigma$ (as per proposition 2.1) in the domain $\mathcal{W}$.

**Proof sketch.** In this setting, $\approx_m$ is as strong as $\approx_o$ on ground terms. What we have to show is that whenever two different $O$ terms can be made equal by a substitution $\rho$, we can produce this $\rho$ by finding a $\sigma$ via $\approx_o$ on the corresponding $M$ terms and by decompiling them. If we look at the syntax of $O$, the only interesting case is $\approx-o$-uva $X \approx_o s$. In this case, after compilation, we have $\approx-o$-uva $Y \square \approx_m t$ that succeeds with $\sigma = \{ Y \mapsto t \}$ and $\sigma$ is decompiled to $\rho = \{ X \mapsto s \}$ by proposition 3.1. \hfill $\Box$

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

**Proof sketch.** Since progress$_1$ is a no-op then step$_m$ and step$_m$ are the same. By theorem 3.2, $\approx_m$ is equivalent to $\approx_o$ in $\mathcal{W}$. \hfill $\Box$

3.7 Notational conventions

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

- $\mathcal{P} = \{ p_1 \approx_o p_2 \}$
- $\mathcal{Q} = \{ t_1 \approx_m t_2 \}$
- $\mathcal{M} = \{ X_1 \mapsto A_1^n \}$
- $L = \{ \Gamma \vdash a \approx_\eta b \}$

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, $Q_2$ corresponds to the $M$ problem $t_4 \approx_m t_4$.

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

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

Definition 3.5 ($\mathcal{W}_\eta$). $\mathcal{W}_\eta(X) \equiv \forall t \in \mathcal{P}(X), t \notin \eta \triangleleft$

4 Handling of $\triangleleft\beta$

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

- $\mathcal{P} = \{ \lambda x. f(\lambda x.a) \}$
- $\mathcal{Q} = \{ \lambda x. f(\lambda a.x) \}$
- $\mathcal{M} = \{ x \mapsto A \}

The unification problem $Q_1$ fails while trying to unify $A x$ and $x$, which is equivalent to $\langle \text{app} [\text{uva} A \square, x] \rangle$ versus $x$. In order to exploit the higher-order unification algorithm of the meta language, we compile the $O$ term $x$ into the $M$ term $A_x$.

4.1 Compilation and decompilation

We add the following rule before rule $\langle \text{capp} \rangle$, where pattern-fragment $Ag$ is a predicate checking if a list of terms is a list of distinct names.

- $\text{comp} \langle \text{capp} \rangle = \langle \text{uva} A Ag1 \rangle = \langle M1 M2 L L S1 S2 \rangle$
- $\text{comp} \langle \text{capp} \rangle = \langle \text{app} A Ag1 \rangle = \langle M1 M2 L L S1 S2 \rangle$
- $\text{comp} \langle \text{capp} \rangle = \langle \text{alloc} \rangle = \langle \text{mv} B (\text{arity Arity}) \rangle$

Note that compiling $Ag$ cannot create new mappings nor links, since $Ag$ is made of bound variables, and the hypothetical rule $(H_s)$ 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 $\approx_o$ in $\mathcal{W}_\eta$).** Given the updated compilation scheme, the $\approx_o$ of section 3.6 is a good unification (as per proposition 2.1) for $\approx_o$ in the domain $\mathcal{W}_\eta$.

**Proof sketch.** If we look at the $O$ terms, there is one more interesting case, namely $\approx-o$-app $\approx-o$-uva $X \approx_o s$ when $\mathcal{W}$ are distinct names compiled to $\mathcal{W}$. In this case the $M$ problem is $Y_\mathcal{O} \approx_m t$ that succeeds with $\sigma = \{ Y \mapsto t[\mathcal{W}/\mathcal{Y}] \}$, which in turn is compiled to $\rho = \{ Y \mapsto \lambda \mathcal{Y}. s[\mathcal{W}/\mathcal{Y}] \}$. Thanks to $\beta_1$ we have $(\lambda \mathcal{Y}. s[\mathcal{W}/\mathcal{Y}]) \mathcal{W} \approx_o s$. \hfill $\Box$

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

In other words the compiler eliminates all subterms that are in $\mathcal{O}$.

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

**Proof sketch.** Thanks to lemma 4.2, $\approx_m$ is as powerful as $\approx_o$ in $\mathcal{W}_\eta$, as well as in $\mathcal{W}$ by theorem 3.3. \hfill $\Box$

5 Handling of $\eta$

A term $\lambda x.t$ is said to be the $\eta$-redex of $t$ if $x$ does not occur free in $t$, and conversely we say that $t$ is the $\eta$-contraction of $\lambda x.t$.

The equational theory of $O$ identifies these terms, but the current compilation scheme does not, as shown by the following example:
while $\lambda x. X \ x \equiv_0 f$ does admit the solution $\rho = \{ X \mapsto f \}$, the corresponding problem in $Q$ does not.

$$\begin{align*}
\mathcal{P} &= \{ \lambda x. X \ x \ \equiv_0 f \} \\
\mathcal{Q} &= \{ \lambda x. A \ \equiv_m f \} \\
\mathcal{M} &= \{ X \mapsto B^1 \} \\
\mathcal{L} &= \{ + A \mapsto \lambda x. B \}
\end{align*}$$

The reason is that «$\lambda x^\prime A \ x^\prime x$» and «$\text{conv}^\prime$» start with different term constructors.

In order to guarantee proposition 2.2, we detect lambda abstractions that can disappear by $\eta$-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 $Q$ to $L$ (section 5.2). The compilation of the problem $\mathcal{P}$ above is refined to:

$$\begin{align*}
\mathcal{P} &= \{ \lambda x. X \ x \ \equiv_0 f \} \\
\mathcal{Q} &= \{ \lambda x. A \ \equiv_m f \} \\
\mathcal{M} &= \{ X \mapsto B^1 \} \\
\mathcal{L} &= \{ + A \mapsto \lambda x. B \}
\end{align*}$$

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

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

Each $\eta$-link is kept in the link store $\mathcal{L}$ 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 $\eta$-redex, i.e. if there exists a substitution $\rho$ such that $\rho(\lambda x.r) \equiv_0 s$. The detection of lambda abstractions that can "disappear" is not as trivial as it may seem. Here are a few examples:

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

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 $\eta$-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 $\eta$-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 $\eta$-contract to $f \ (A \ x)$ making $\lambda x.f \ (A \ x)$ a potential $\eta$-redex.

We can now detail how $\diamond \eta$ terms are detected.

**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 $\rho s \equiv_0 x$.

**Lemma 5.2.** A $\beta$-normal term $s = \lambda x_1 \ldots x_n.t$ 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. $\lambda x_1 \ldots x_n.x \ x_1 \ldots x_n \equiv_0 x$);

3. $t$ is a unification variable with scope $W$, and for any $\nu \in \{x, x_1 \ldots x_n\}$, there exists a $w \in W$, such that $w \ may$-contract-to $\nu$ (if $n = 0$ this is equivalent to $x \in W$).

**Proof sketch.** Since our terms are in $\beta$-normal form the only rule that can play a role is $\eta_\nu$, 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 $\eta$-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 $\eta$ contraction. In that case, the term $t$ under the spine of binders $x_1 \ldots x_n$ 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 $x \ x_1 \ldots x_n$ (case 3).

$$\square$$

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

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 $\eta$-contracting $t$ cannot make $x$ disappear, since $x$ is not a locally bound variable inside $t$.

We can now describe the implementation of $\mathcal{O}\eta$ detection:

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

1. $t$ is a constant $c$ or a name $y$ applied to the arguments $l_1 \ldots l_m$ 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 $l_1 \ldots l_{m-n} y$;

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 ($\mathcal{O} \eta$ detection).** If $t$ is a $\beta$-normal term and $t \in \diamond \eta$ then $\text{may-eta} t$ 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 $\rho$ such that $pt \in \mathcal{P}$ is an $\eta$-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 $(c_j)$ from the code in section 3.3.

```
comp \ (o-lam F) \ (uva A Scope) M1 M2 L1 L3 S1 S3 ::=
may-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 ≥η 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 λx.λy.t, where x and y occur in t. If maybe-eta λy.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 λx.λy.t does not hold, then also maybe-eta λx.λy.t does not hold either, contradicting the assumption that the rule was applied.

**Lemma 5.7 (W-partial-enforcement).** ∀s, (s) ≣ (t, m, l), if Wη(s) then W(t).

**Proof sketch.** By lemma 5.5

**Decomposition.** The decomposition 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 Γ ⊢ X =η t is removed from L when X becomes rigid. Let y ∈ Γ. There are two cases:

1. if X = a or X = y or X = f₁ a₁ . . . aⁿ we unify the η-redex of X with t, that is we run λx.X x =m t.
2. if X = λx.s we run X =m t.

**Definition 5.9 (η-progress-deduplicate).** A link Γ ⊢ X₂ =η T is removed from L when another link Δ ⊢ X₁ =η T' is in L. By invariant 1 the length of ṡ and ṕ is the same; hence we can move the term T' from Δ to Γ by renaming its bound variables, i.e. T'' = T' [fühl]. We then run T =m T'' (under the context Γ).

**Lemma 5.10.** Let λx.t be the rhs of a η-link, then W(t), enforcing invariant 3.

**Proof sketch.** By lemma 5.7.

**Lemma 5.11.** Given a η-link Γ ⊢ X =η λx.t, the unification done by η-progress-lhs is between terms in W.

**Proof sketch.** Let σ be a substitution such that W(σX) (since =m preserves W). By invariant 3, we have W(t). If σX = λx.r, then r is unified with t and both terms are in W. Otherwise, λx.X x is unified with λx.t, and X x and t are both in W.

**Lemma 5.12.** The unification done by η-progress-deduplicate is between terms in W.

**Proof.** Given two η-links Γ₁ ⊢ X =η λx.t and Γ₂ ⊢ Y =η λx.t, the unification is performed between t and t'. By lemma 5.7 both terms are in W.

**Lemma 5.13.** The progress of η-link guarantees proposition 2.11 (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 W.

**Lemma 5.14.** η-progress terminates.

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

**Theorem 5.15 (Fidelity in Wη).** Given a list of unification problems F such that Wη(F), 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 step₀ succeeds iff step₂ succeeds, where step₂ is made by a unification step u and a link progression p. Without loss of generality we consider a Ω unification problem of the form s₁ ⊳ η s₂ where s₁ ∈ ≥η and is compiled to a variable X with the accessory link Γ ⊢ X =η λx.t₁. 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:

1. s₂ ∈ ≥η, in this case, the unification problem becomes X =m Y with the extra link Γ ⊢ Y =η λx.t₂. After the unification of X and Y, η-progress-deduplicate is fired and, once the lambdas are crossed, t₁ and t₂ are unified. This unification succeeds iff s₁ ⊳ η s₂ succeeds by theorem 4.3.
2. s₂ ∉ ≥η, in this case s₂ is compiled to t₂ and the unification step u assigns t₂ to X and triggers η-progress-lhs. If s₁ and s₂ are equal thanks to η then progression p unifies λx.t₁ with the η-redex of t₂ namely λx.t₂ x, otherwise they are equal thanks to rule λλ and the link progression p unifies λx.t₁ with t₂. Alternatively, if s₁ are different s₂ it cannot be because of the λ constructor in the head of t₁, since rule λλ or rule η move under it, eventually finding two different subterms s₂' and s₂''. By theorem 4.3, ≡m fails on terms t₁' and t₂'' even if η-progress-lhs η-expands t₂.

**Corollary 5.16 (Properties of ≡m in Wη).** Given the updated compilation scheme, procedure ≡m is Ω in the domain Wη.

**Example of η-progress-lhs.** The example at the beginning of section 5, once σ = { A → f }, triggers η-progress-lhs since the link becomes ⊢ f =η λx.Bₓ and the lhs is a constant. This rule runs λx.f x =m λx.Bₓ, resulting in σ = { A → f ; Bₓ → f }. Since X is mapped to B and Bₓ is compiled into λx.f x, by the definition of η-link decomposition σ is compiled to ρ = { X → λx.f x }.

**6 MAKING Μ 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 $M$ generated by the compiler is not a bijection. For example:

\[
\begin{align*}
\mathbb{P} &= \{ \lambda x . (X x) \approx_o f & X \approx_o \lambda x . a \} \\
\mathbb{Q} &= \{ A \approx_m f & C \approx_m \lambda x . a \} \\
M &= \{ X \mapsto \mathcal{C} & X \mapsto \mathcal{B}^1 \} \\
L &= \{ t \mapsto A \approx_n \lambda x . B_x \}
\end{align*}
\]

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 $M$ for $X$, namely $B$ and $C$. However, there is no connection between these two variables and any incompatible assignments to them would not be detected, in turn breaking proposition 2.3. To address this issue, we post-process $M$ to ensure the following property:

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

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

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

\[
x_0 \ldots x_{m+i} \mapsto B^{i}_{m+i}, \ldots_{m+i} \approx_n \lambda x_{m+i+1} \ldots_{m+i+1} B^{i_{m+i}}
\]

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 $m_1, m_2 \in M$ such that $m_1 : X \mapsto A^m$ and $m_2 : X \mapsto A^n$ and $m < n$ we remove $m_1$ from $M$ and add to $L$ 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 $W_{\theta_\eta}(\mathbb{P})$, if $\mathbb{P}$ contains the same $O$-variable used at different arities, then map-deduplication guarantees proposition 2.3 (SIMULATION FIDELITY)

**Proof Sketch.** Let $X$ be a $O$-variable appearing in two different subterms $s_1$ and $s_2$. In $s_1$ variable $X$ is applied to the list of terms $x_1 \ldots x_m$ while, in $s_2$, it is applied to $y_1 \ldots y_n$. Without loss of generality we take $m < n$. After map-deduplication we have two distinct $M$-variables for $X$, say $A$ and $C$, related by a chain $\eta$ of $\eta$-links of length $n - m$. We prove that if $A$ and $C$ are assigned respectively to the $W$ terms $\lambda x_1 \ldots x_p . t_1$ and $\lambda x_1 \ldots x_q . t_2$, then a failure occurs iff these terms do not unify. By $\eta$-progress-lhs, the assignment of $\lambda$ activates $p$ links of $\eta$. In particular, when $p > n$, then the entire chain collapses and unification is performed between $\lambda x_1 \ldots x_p . t_1$ and $\lambda x_1 \ldots x_q . t_2$. By theorem 4.3 and since we work with $W$ terms, this unification succeeds iff it does in $O$. 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 $\eta$-redex of $t_1$, implying $q = 0$. If it is not the case, then $\eta$-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 $x_1 \ldots x_p$ as free variables, causing otherwise a unification failure. After this check, $m - n - p$ links of $\eta$ remain suspended, as we expect: decompilation is charged to wake all links in $L$ 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:

\[
\begin{align*}
\mathbb{P} &= \{ \lambda x . (X x) \approx_o f & X \approx_o \lambda x . a \} \\
\mathbb{Q} &= \{ A \approx_m f & C \approx_m \lambda x . a \} \\
M &= \{ X \mapsto \mathcal{C} & X \mapsto \mathcal{B} \} \\
L &= \{ t \mapsto A \approx_n \lambda x . B_x \}
\end{align*}
\]

Note that running $\approx_m$ makes the compiler did generate two entries in $M$ for $X$, namely $B$ and $C$. The resolution of $\mathbb{Q}_1$ assigns $\approx_o$ to $A$. This wakes up $L_2$ by $\eta$-progress-lhs, assigning $\approx_o f x x$ to $B_x$. The resolution of $\mathbb{Q}_2$ instantiates $\lambda x . a$ to $B_x$, which, in turn triggers $L_3$ by $\eta$-progress-lhs. In turn, this performs $\lambda x . a \approx_m \lambda x . (f x)$ that fails as expected.

7 HANDLING \& \textit{O} \textit{L}

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

In the following example problem $\mathbb{P}_2$, namely $X a \approx_o a$, admits two different solutions: $p_1 = \{ X \mapsto \lambda x . x \}$ and $p_2 = \{ X \mapsto \lambda x . a \}$. The unification algorithm above, and since no solution is more general than the other, but for the first problem the choice is obvious, and in turn this choice makes $\mathbb{P}_2 \subseteq \mathcal{L}$:

\[
\begin{align*}
\mathbb{P} &= \{ X \approx_o \lambda x . a & (X a) \approx_o a \} \\
\mathbb{Q} &= \{ A \approx_m \lambda x . a & (A a) \approx_m a \} \\
M &= \{ X \mapsto \mathcal{A} \}
\end{align*}
\]

In order to support this scenario we have to improve a little our compiler and progress routines. In particular $\mathbb{Q}_1$ generates $\sigma = \{ A \approx_o \lambda x . a \}$ making $\sigma \mathbb{Q}_2$ equal to $\lambda x . a \approx_o a$ that $\approx_o$ cannot solve since it lacks $\beta_1$ and $\beta_f$.

To address this problem the compiler must recognize $\mathcal{O} \mathcal{L}$ 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 (L-link rhs).** The rhs of any $\mathcal{L}$-link has the shape $X_{s_1 \ldots s_n} t_1 \ldots t_m$ where $X$ is a unification variable in $\mathcal{L}$ (with scope $s_1 \ldots s_n$) and $t_1 \ldots t_m$ is a list of terms such that $m > 0$ and $t_1$ is either a variable occurring in $s_1 \ldots s_n$ or a term other than a variable.

Note that the shape of such rhs is «app [uva X S \mid L]» where $S$ is $[s_1, \ldots, s_n]$ and $L$ is $[t_1, \ldots, t_m]$.

7.1 Compilation and decompilation

We insert this rule just before rule (c@0):

comp (o-app [o-uva A Ag]) (uva B Sc) M1 M3 L1 L3 S1 S4 := 1, 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 M1 L1 L1 S3 S3,
We relax this condition and accommodate for heuristics in 7.3.

\[ l \text{ is split into two parts: } Pf \text{ is in } L, \text{ and can be empty; Extra cannot be empty and is such that } \text{app } Pf \text{ Extra age. The} \]
\[ \text{rhs of the } L \text{-link is the application of a fresh variable } c \text{ having in scope all names in } Pf \text{ (the compilation of } Pf). \text{ The variable } b, \text{ returned as the compiled term, is a fresh variable having in scope all the free variables occurring in } Pf \text{ and Extra (the compilation of Extra).} \text{ This construction enforces invariant } 4 \text{ (} L \text{-link rhs).} \]

**Lemma 7.1 (L-enforcement).** \( \forall s, (s) \mapsto (t, m, l), \text{ then } W(t). \)

**Proof sketch.** By lemmas 4.2 and 5.7 and the rule above. \( \Box \)

**Decomposition.** All \( L \text{-link} \) should be solved before decomposition. If any \( L \text{-link} \) remains in \( L \), decomposition fails.

### 7.2 Progress

**Definition 7.2 (L-progress-refine).** Let \( \sigma \) be a substitution such that \( \sigma t_1 \) is a name \( s \) not occurring in \( s_1 \ldots s_n \). If \( m = 1 \), then \( t \) is removed and the lhs is unified with \( X_{s_1 \ldots s_n} \). If \( m > 1 \), then \( t \) is replaced by the link \( \Gamma \vdash s_1 \ldots s_n t \ldots t_m \) (where \( Y \) is a fresh variable of arity \( n + 1 \)) and link \( \Gamma \vdash X_{s_1 \ldots s_n} = \eta \lambda x.Y_{s_1 \ldots s_n} x \) is added to \( L \).

**Definition 7.3 (L-progress-rhs).** Link \( t \) is removed from \( L \) if \( X_{s_1 \ldots s_n} \) is instantiated to a term \( f \) and \( t_1 \ldots t_m \) \( \beta \)-reduces to a \( t' \) in \( L \).

**Definition 7.4 (L-progress-fail).** Progress fails when either

- there exists another link \( t' \) in \( L \) with the same \( \eta \)-lhs as \( t \); or
- the \( \eta \)-lhs of \( t \) become rigid.

We relax this condition and accommodate for heuristics in 7.3. Finally we introduce the following \( \eta \)-link progress rule.

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

- \( \text{maybe-eta } T \) does not hold (anymore), in this case \( X \) is unified with \( T \); or
- \( T \eta \)-contracts to a term \( T' \) not starting with the \( \eta \)am constructor. In this case \( X \) is unified with \( T' \).

The idea behind this rule is to instantiate the \( \eta \)-lhs variable of a \( \eta \)-link whenever its rhs is for sure a \( \eta \)-W term.

**Lemma 7.6.** progress terminates

**Proof sketch.** Let \( l = \Gamma \vdash t =_L X_{s_1 \ldots s_n} t_1 \ldots t_m \) a \( L \)-link in the store \( L \). The progression made by \( L \)-progress-rhs terminates: \( l \) is removed from \( L \) after having performed terminating instructions. If \( l \) is not activated by \( L \)-progress-rhs, then \( X \) is a variables.

The activation of \( L \)-progress-refine replaces \( l \) with the new \( L \)-link \( \Gamma \vdash s_1 \ldots s_n t \ldots t_m \). 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 \( L \) after the unification between \( \eta \)-lhs with rhs. We also note that each \( L \)-progress-refine generates a \( \eta \)-link, yet, according to lemma 5.14, this does not affect termination. \( L \)-progress-fail terminates since it stop progression with failure. Finally, \( \eta \)-progress-rhs performs terminating operations and, if its permises succeed, the considered \( \eta \)-link is removed from \( L \), ensuring termination. \( \Box \)

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

**Proof sketch.** Let \( P \) be the first problem in \( \mathbb{P} \) such that it is not in \( L \) and let \( \sigma \) be the substitution obtained solving \( Q_1 \ldots Q_{i-1} \). After the execution of \( step_m \) for the \( i \) - 1 time, each variable \( X \) corresponding to a \( \lambda \eta \) subterm in the original \( O \) unification problem, is instantiated iff it is known for sure that \( X \) is no more in \( \lambda \eta \). If \( \sigma Q_j \) is in \( L \), then by definitions 7.2 and 7.3 the associated \( L \)-link is solved and removed. In this case, all calls to \( \eta \)-m are between terms in \( L \) and by theorem 4.3 fidelity is guaranteed. If \( \sigma Q_j \) is still in \( \mathbb{O} \), then by definition 7.4 unification fails as the corresponding unification in \( O \) would (it is called outside its domain). \( \Box \)

### 7.3 Relaxing definition 7.4 (\( L \)-progress-fail)

Working with terms in \( L \) is sometime too restrictive [1] and we could find in literature a few strategies to go beyond \( L \) without implementing Huet’s algorithm [11]. Some implementations of \( \lambda \)Prolog [18] such as Teju [17] delay the resolution of \( \Diamond L \) unification problems until the substitution makes them reenter \( L \). 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 \( \mathbb{L} \).

\[
\begin{align*}
\mathbb{P} &= \set{(X \ a) =_\sigma a \ X =_\sigma \lambda x.Y} \\
\mathbb{Q} &= \set{A =_\sigma B \ B =_\sigma \lambda x.C} \\
\mathbb{M} &= \set{Y \mapsto C^0 \ X \mapsto B^0} \\
\mathbb{L} &= \set{\vdash A =_L B^0}
\end{align*}
\]

If we want the object-language unification to delay the first unification problem (waiting for \( X \) to be instantiated), we can relax definition 7.4. Instead of failing when the \( \eta \)-lhs of \( L_1 \) becomes rigid (equal to \( a \)), we keep it in \( L \) 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-\( \eta \)-links procedure becomes incomplete since invariant 2 (Link left hand side) is broken. Also note that delaying unification outside \( L \) can leave \( L \)-link for the decomposition phase. Therefore commit-\( \eta \)-links should be modified accordingly.

If instead we want \( \sigma \) 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 \( Xa \) 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 programs (the rules)
and execute them, that is Elpi plays the role of 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 run-
time iterates \( \simeq_m \) but \( \text{step}_m \) also needs to check links for progress.
Luckily Elpi extends \( \lambda \text{Prolog} \) 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 vari-
ables is resumed as \textit{as soon as one of these variable is assigned},
before any other existing goal is considered. In turn link activation
grants proposition 2.3 \textit{(Simulation Fidelity)}. We point out that
[13] delays \( \triangle \mathcal{L} \) unification problems while we also delay prob-
lems that are in \( \mathcal{O}_Y \). For brevity we only provide two pseudo-code
snippets. The first one depicts how \( \eta \text{-link} \) are suspended or make
progress.

\begin{verbatim}
  link-eta L R :- not (var L), 1, eta-progress-lhs L R.
  link-eta L R :- not (maybe-eta R), 1, eta-progress-rhs L R.
  link-eta L R :- declare_constraint (link-eta L R) [L,R].
\end{verbatim}

The second snippet illustrates the deduplication of \( \eta \text{-link} \). The syntax \( \langle H \Rightarrow G \ ? \ P \rangle \) denotes a \( \lambda \text{Prolog} \) sequent, that is a goal \( P \)
under a set \( G \) of hypothetical rules (introduced by \( \Rightarrow \)) and where
the program context binds (via the pi operator) eigenvariables in \( H \). Sequents are presented to CHR with their higher order unification
variables replaced by “frozen constants”, that is \( \lambda_{xyz} \) becomes
\( \text{uvar} \ f_A \ [x,y,z] \) for a fresh constant \( f_A \). Frozen constants are
“defrost” when they are part of a new goal (see [10, section 4.3]).

\begin{verbatim}
  rule (N1 -> G1 ?- link-eta (uvar X LX1) T1) % match
    / (N2 -> G2 ?- link-eta (uvar X LX2) T2) % remove
    I (relocate LX1 LX2 T2 T2') % condition
    \(\Rightarrow\) (N1 -> G1 ?- T1 = T2'). % new goal
\end{verbatim}

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
\( \simeq_m \) as a regular routine, i.e. write rules as follows

\begin{verbatim}
  decision X :- unify X (all A \( \backslash \) app [P, X]), finite A, 
    pi X \( \backslash \) decision (app [P, X]).
\end{verbatim}

where unify is a regular predicate. This method fails to take ad-
vantage 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:

\begin{verbatim}
  decision (all A \( \backslash \) P x) :- finite A, pi x \( \backslash \) decision (P x).
\end{verbatim}

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 cannot 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. Conse-
sequently, it must be able to manipulate terms that are not known
in advance without relying on introspection primitives such as
\( \lambda \text{Prolog’s functor/3} \) and \( \text{arg/3} \). 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 \( \text{con} \) 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
\( \lambda \text{Prolog} \) 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, 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 pres-
ence 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 lan-
guage. 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.

\begin{quote}
The approach presented in this paper addresses all the concerns
mentioned earlier. It takes advantage of the unification capabil-
ities 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
Stdp 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 tabed search.

We considered mechanizing our results with Abella [8], espe-
cially 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 \( \text{map} \) or \( \text{forall/2} \) is surely possible, but has
a high cost in verbosity given how pervasively they are used. More-
over, and more importantly, it creates a distance between the veri-
fied 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.
\end{quote}
ACKNOWLEDGMENTS
This work has been supported by the French government, through the France 2030 investment plan managed by the Agence Nationale de la Recherche, as part of the “UCA DS4H” project, reference ANR-17-EURE-0004.

REFERENCES


