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-04547069v2

HAL Id: hal-04547069
https://inria.hal.science/hal-04547069v2
Preprint submitted on 27 Apr 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
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 [19], Twelf [21], λProlog [14], Beluga [22], Abella [8] and Isabelle [28] which have been used to implement or specify many formal systems such as First Order Logic [5], Set Theory [18], Higher Order Logic [17], and the Calculus of Constructions [4].

The object logic we are interested in is Coq’s type theory [26]. 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 \(\eta\beta\)-equivalence and features a higher-order unification procedure \(\simeq_m\) for the pattern fragment [13]. Elpi offers an encoding of Coq terms suitable for meta programming [25, 24, 9, 6] but that restricts \(\simeq_m\) to first-order unification problems only. We refer to this basic encoding as \(O\).

In this paper we propose a more well-behaved encoding called \(M\), and show how to translate unification problems from \(O\) to corresponding ones in \(M\). Consequently, we derive \(\simeq_o\), the higher-order unification procedure of \(O\) that honours \(\eta\beta\)-equivalence (for Coq functions), addresses problems within the pattern fragment, and allows for the use of heuristics to deal with problems 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 \(O\) 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 [27, 23] for the Coq’s system [26]. 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 [25], a dialect of λProlog, already used to extend Coq in various ways [25, 24, 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 \(\text{fin}\ n\), is finite; 2) the predicate \(\text{nfact}\ n\ n\), relating a natural number \(n\) to the number of its prime factors \(n_f\), is decidable; 3) the universal closure of a predicate has a decision procedure if its domain is finite and if the predicate is decidable.

\[
\begin{align*}
\text{Instance } \text{fin\_fin} & \quad : \forall n, \text{Finite (fin } n) \quad . \quad (+ ~ r1) \\
\text{Instance } \text{nfact\_dec} & \quad : \forall n, \text{Decision (nfact } n\ n) \quad . \quad (+ ~ r2) \\
\text{Instance } \text{forall\_dec} & \quad : \forall A, \text{Finite } A \rightarrow \text{Decision } (\forall x : A, \text{P } x) \quad . \quad (+ ~ r3) \\
\end{align*}
\]

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

\[
\text{Decision } (\forall x : \text{fin } 7, \text{nfact } x \ 3)\quad . \quad (+ ~ g)
\]

The proof found by the solver back-chains on rule 3 (the only rule about the \(\forall\) quantifier), and then solves the premises with rules 1 and 2 respectively. Note that rule 3 features a second-order parameter \(P\) that represents a function of type \(\lambda A \rightarrow \text{Prop}\) (a 1-argument predicate over \(A\)). The solver has to infer a value for \(P\) by unifying the conclusion of rule 3 with the goal, and in particular, it has to solve the unification problem \(P \ x = \text{nfact } x \ 3\). This higher-order problem falls in the so-called pattern fragment and admits a unique solution \(\rho\) that assigns the term \(\lambda x. \text{nfact } x \ 3\) to \(P\). In the rest of this paper, we refer to the pattern fragment as \(L\) [13].

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 an Higher-Order Abstract Syntax
We now illustrate a naive encoding of the three instances above as

\[
\text{(The problem above has no solution in a first-order language where)}
\]

\[
\text{\beta}\text{-unification variables are higher-order and come equipped with an}
\]

\[
\text{\eta}\beta\text{-conversion. We show how \eta\beta\text{-conversion can be extended with heuristics}}
\]

\[
\text{to deal with problems outside the pattern fragment.}
\]

\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 \(O\) 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 \(\equiv_m\) can be efficiently used to simulate a higher-order unification procedure \(\equiv_o\) for the object language that features \(\eta\beta\)-conversion. We show how \(\equiv_o\) can be extended with heuristics to deal with problems outside the pattern fragment.

The root cause of the problems we outlined in this example is a subtle mismatch between the equational theories of the meta-language and the object language, which in turn makes the unification procedures of the meta-language weak. The equational theory of the meta-language Elpi encompasses \(\eta\beta\)-equivalence and its unification procedure can solve higher-order problems in the pattern fragment. Although the equational theory of Coq is much richer, for efficiency and predictability reasons, automatic proof search procedures typically employ a unification procedure that only captures a \(\eta\beta\)-equivalence and only operates in \(L\). 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 \(\equiv_m\) can be efficiently used to simulate a higher-order unification procedure \(\equiv_o\) for the object language that features \(\eta\beta\)-conversion. We show how \(\equiv_o\) 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 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 \(M\)}

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 \(O\) (fuva term constructor) have no explicit scope: the arguments of a higher-order variable are given via the \(f\)app constructor. For example the term \(\text{\&}\ P \ x\ \ x\) is represented as \(f\text{\app}\ \text{fuva}\ N\ x\ x\), where \(N\) is the memory address of \(P\) and \(x\) is a bound variable.

In \(M\), the representation of \(\text{\&}\ P\ x\ x\) is instead \(\text{\lf}v\text{\ua}\ N\ [x]\), since unification variables are higher-order and come equipped with an explicit scope.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\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 application. Here are few examples:

$$
\begin{align*}
(f & a) \text{ app } [\text{con } "f", \text{con } "a"] \\
\lambda x.\lambda y. F_{xy} & \text{ lam } x \ \text{ lam } y \ \text{ uva } F \ [x, \ y] \\
\lambda x.\lambda x. a & \text{ lam } x \ \text{ lam } \text{ app } [\text{uva } F \ [x], \ \text{con } "a"] \\
\lambda x. F_x x & \text{ lam } x \ \text{ lam } \text{ app } [\text{uva } F \ [x], \ x]
\end{align*}
$$

When it is clear from the context, we shall use the same syntax for $O$ terms (although we never subscript 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: $\approx_o$ and $\approx_m$. For both languages, we extend the equational theory over ground terms to the full language by adding the reflexivity for unification variables (a variable is equal to itself).

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

$$
\text{type } (\approx_o) \ 	ext{ fm } \rightarrow \text{ fm } \rightarrow \text{ o.} \\
\text{fcon X } \equiv_o \text{ fcon X.} \\
fapp \ A \equiv_o \ fapp \ B \ \Rightarrow \text{ forall2 } (\approx_o) \ A \ B. \\
\text{flam F } \equiv_o \ \text{flam G } \ \Rightarrow \pi \ x \ \text{x } \equiv_o \ x \ \Rightarrow \ F \ x \ \equiv_o \ G \ x. \\
fuvu N \equiv_o \ fuvu N. \\
\text{fapp \ F } \equiv_o \ T \ \Rightarrow \pi \ x \ \beta \ T \ [x]\ (T' \ x), \ x \equiv_o \ x \ \Rightarrow \ F \ x \ \equiv_o \ T' \ x. \\
T \equiv_o \ \text{flam F } \ \Rightarrow \pi \ x \ \beta \ T \ [x]\ (T' \ x), \ x \equiv_o \ x \ \Rightarrow \ T' \ x \ \equiv_o \ T. \\
fapp \ [\text{flam X } | \ L] \equiv_o T \ \Rightarrow \beta \ (\text{flam X} \ L \ R, \ T \equiv_o R). \\
fapp \ [\text{flam X } | \ L] \equiv_o T \ \Rightarrow \beta \ (\text{flam X} \ L \ R, \ T \equiv_o R). \ (\beta_o)
\text{type } (\approx_m) \ 	ext{ tm } \rightarrow \text{ tm } \rightarrow \text{ o.} \\
\text{con C } \equiv_m \ \text{con C.} \\
\text{app A } \equiv_m \ \text{app } B \ \Rightarrow \ \text{forall2 } (\approx_m) \ A \ B. \\
\text{lam F } \equiv_m \ \text{lam G } \ \Rightarrow \pi \ x \ \text{x } \equiv_m \ x \ \Rightarrow \ F \ x \ \equiv_m \ G \ x. \\
uva N \equiv_m \ uva N \ B \ \Rightarrow \ \text{forall2 } (\approx_m) \ A \ B.
\end{align*}
$$

Note that the symbol $|$ separates the head of a list from the tail, that forall2 applies a binary predicate to two lists (of same length). Finally $\Rightarrow$ is $\lambda$Prolog syntax to augment the current program with an extra rule, and both procedures use it to assume that the fresh nominal constants introduced by $\pi$ are equal to themselves.

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

Substitution: $rs$ and $st$. We write $\sigma = \{ X \rightarrow t \}$ for the substitution that assigns the term $t$ to the variable $X$. We write $\sigma t$ for the application of the substitution to a term $t$, and $\sigma X = \{ \sigma t \ | \ t \in X \}$ 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: $\approx_o$ vs. $\approx_m$. $M$’s unification signature is:

$$
\text{type } (\approx_m) \ tm \rightarrow \ tm \rightarrow \text{ subst } \rightarrow \text{ subst } \rightarrow \text{ o.}
$$

We write $\sigma_1 \approx_m \sigma_2 \iff \sigma'$. 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 $t_1 \approx_m t_2 \iff \sigma'$ when the initial substitution $\sigma$ is empty. We write $\mathcal{L}$ as the set of terms that are in the pattern-fragment, i.e. every higher-order variable is applied to a list of distinct names.

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

$$
\text{PROP} 2.1 \text{ (GOOD UNIFICATION). A good unification } \approx \text{ for equality in the domain } \mathcal{D} \text{ satisfies the following properties:}
$$

$$
\begin{align*}
(1) & \{ t_1, t_2 \} \subseteq \mathcal{D} \Rightarrow t_1 \equiv t_2 \iff \rho \Rightarrow \rho_1 = \rho_2 \\
(2) & \{ t_1, t_2 \} \subseteq \mathcal{D} \Rightarrow \rho_1 = \rho_2 \Rightarrow \exists \rho', t_1 \equiv t_2 \iff \rho' \land \rho' \subseteq \rho
\end{align*}
$$

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

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

2.3 The problem: logic-program simulation

We represent a logic program run in $O$ as a sequence of steps of length $n$. At each step $p$ we unify two terms, $\mathcal{P}_p$ and $\mathcal{P}_p$, taken from the list of all unification problems $\mathcal{P}$. 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.

$$
\text{ostep}(\mathcal{P}, \rho, \rho) \iff \rho' \iff \rho \equiv\ P_1 \rho_1 \equiv_o \rho \equiv_o \rho_p \Rightarrow \rho'
\text{orun}(\mathcal{P}, n) \iff \rho_n \iff \land_{k=1}^n \text{ostep}(\mathcal{P}, \rho, \rho_{p-1}) \Rightarrow \rho_p
$$

In order to simulate a $O$ logic program in $M$, we compile each $O$ term $s$ in $\mathcal{P}$ to a $M$ term $t$. 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 “decompile” the assignment, $(\sigma, m, l)^{-1} \mapsto \rho$. Links are an accessory piece of information whose description is deferred to section 2.4. We write $\mathcal{P}_p = \{ \mathcal{P}_p, \mathcal{P}_p \}$ and $s \in \mathcal{P} \Rightarrow \exists \rho, s \in \mathcal{P}_p$.

We simulate each run in $O$ with a run in $M$ as follows:

$$
\begin{align*}
\text{mstep}(Q, p, \sigma, L) & \iff (\sigma', \ L') \iff (\sigma, \ L) \iff (\sigma', \ L') \\
\text{murun}(\mathcal{P}, n) & \iff \rho_n \iff Q \times M \times L_0 = \{(t,m,l) | s \in \mathcal{P}, \sigma, \ (t, m, l) \\
\land_{p=1}^n \text{mstep}(Q, p, \sigma_{p-1}, L_{p-1}) \iff \sigma_p, \ L_p \\
\sigma_{n, \ M, L_n}^{-1} \mapsto \rho_n
\end{align*}
$$

Higher-Order unification for free! Conference ‘17, July 2017, Washington, DC, USA
By analogy with \( \mathcal{P} \), we write \( \mathcal{Q}_p \) and \( \mathcal{Q}_{p} \) for the two \( M \) terms being unified at step \( p \), and we write \( \mathcal{Q}_p \) for the set \{ \( \mathcal{Q}_p, \mathcal{Q}_{p} \) \}. \text{mstep} 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 \( \simeq_m \). \text{mrun} compiles all terms in \( \mathcal{P} \), then executes each step, and finally compiles the solution. We claim:

**Proposition 2.2 (Run equivalence).** \( \forall \mathcal{P}, \forall n, \text{if } \mathcal{P} \subseteq \mathcal{L} \)
\[ \text{orun}(\mathcal{P}, n) \mapsto \rho \land \text{mrun}(\mathcal{P}, n) \mapsto \rho' \Rightarrow \forall s \in \mathcal{P}, ps \simeq_{\alpha} \rho's \]
That is, the two executions give equivalent results if all terms in \( \mathcal{P} \) are in the pattern fragment. Moreover:

**Proposition 2.3 (Simulation fidelity).** In the context of \( \text{mrun} \), \( \mathcal{P} \subseteq \mathcal{L} \) we have that \( \forall p \in 1 \ldots n \),
\[ \text{ostep}(\mathcal{P}, p, \rho_{p-1}) \mapsto \rho_p \Leftrightarrow \text{mstep}(\mathcal{Q}_p, p, \sigma_{p-1}, \mathcal{L}_{p-1}) \mapsto (\sigma_p, \mathcal{L}_p) \]
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 \( \text{mrun} \) handles terms outside \( \mathcal{L} \) in the following sense:

**Proposition 2.4 (Fidelity recovery).** In the context of \( \text{mrun} \), if \( \rho_{p-1} \in \mathcal{L} \) (even if \( \mathcal{P} \not\subseteq \mathcal{L} \)) then
\[ \text{ostep}(\mathcal{P}, p, \rho_{p-1}) \mapsto \rho_p \Leftarrow \text{mstep}(\mathcal{Q}_p, p, \sigma_{p-1}, \mathcal{L}_{p-1}) \mapsto (\sigma_p, \mathcal{L}_p) \]

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

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

### 2.4 The solution (in a nutshell)

A term \( s \) is compiled to a term \( t \) where every “problematic” subterm \( p \) is replaced by a fresh unification variable \( h \) with an accessory link that represents a suspended unification problem \( h \simeq_m p \). As a result, \( \simeq_m \) is “well behaved” on \( t \), in the sense that it does not contradict \( =_\alpha \) 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 \( X x_1 \ldots x_n \) 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 (\( \diamond \eta \)).** \( \diamond \eta \) is the set of terms \( \lambda x.s \) such that \( \exists p, \rho \lambda x.s \) is an eta expansion.

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

**Definition 2.7 (\( \diamond \mathcal{L} \)).** \( \diamond \mathcal{L} \) is the set of terms of the form \( X t_1 \ldots t_n \) 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, there exists a substitution \( \rho \) such that \( ps \in \mathcal{L} \).

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

**Definition 2.8 (Well behaved set).** Given a set of terms \( X \subseteq \mathcal{M} \),
\[ \mathcal{W}(X) \equiv \forall t \in \mathcal{P}(X), t \not\in (\diamond \beta \cup \diamond \eta \cup \diamond \mathcal{L}) \]

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 \( \mathcal{O} \),
\[ (s) \mapsto (t, m, l) \Rightarrow \mathcal{W}(t) \]

**Proposition 2.10 (compiler-useful).** Given two terms \( s_1 \) and \( s_2 \), if \( \exists p, ps_1 \simeq_{\alpha} ps_2 \), then
\[ (s_i) \mapsto (t_i, m_i, l_i) \text{ for } i \in \{1, 2\} \Rightarrow t_1 \simeq_m t_2 \Rightarrow \sigma \]

In other words the compiler outputs terms in \( \mathcal{W} \), even if its input is not. And if two terms can unify in the source language then these terms also unify in the target language. Note that the property holds for any substitution, \( p \) could be given by an oracle and/or not necessarily be a most general one: when applied to terms in \( \mathcal{W} \), \( \simeq_m \) does not contradict \( \simeq_{\alpha} \). Also note that a compiler that translate all terms to unification variables would satisfy this property, but would fail to verify proposition 2.3.

**Proposition 2.11 (\( \mathcal{W} \)-preservation).** \( \forall \mathcal{Q}, \forall \mathcal{L}, \forall \rho, \forall \sigma, \forall o \cdot \rho \mathcal{Q} \rightarrow \mathcal{W}(\mathcal{Q}) \)
**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{mstep}, hence \( \simeq_m \) can continue
to operate properly. In sections 3, 5 and 7 we describe how to rec-
ognize terms in \(\land\), \(\land\) and \(\land\) and how progress takes care of
them preserving \(W\) and ensuring 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: they are equivalent only by using decompilation
to take into account terms in \(L\). Then, by induction on the number of
steps, 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:

\[
\text{kind } \text{inc}tx \text{ type } \rightarrow \text{type.}
\]

\[
\text{type } \text{abs} \ (\text{tm } \rightarrow \text{inc}tx \ A) \rightarrow \text{inc}tx \ A.
\]

\[
\text{type } \text{assign} \ (\text{tm } \rightarrow \text{inc}tx \ A) \rightarrow \text{inc}tx \ A.
\]

\[
\text{type } \text{subst} \ (\text{tm } \rightarrow \text{inc}tx \ A).
\]

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
\(\text{inc}tx\) container, and in particular its abs binding constructor.

\[
\text{kind } \text{inc}tx \text{ type } \rightarrow \text{type.} \quad \text{(\(\cdot\)\(\cdot\))}
\]

\[
\text{type } \text{abs} \ (\text{tm } \rightarrow \text{inc}tx \ A) \rightarrow \text{inc}tx \ A.
\]

\[
\text{type } \text{assign} \ (\text{tm } \rightarrow \text{inc}tx \ A) \rightarrow \text{inc}tx \ A.
\]

\[
\text{type } \text{subst} \ (\text{tm } \rightarrow \text{inc}tx \ A).
\]

A solution to a \(O\) variable is a plain term, that is, \(f\text{subst}\) is an
abbreviation for \(\text{mem ftm}\).

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

\[
\text{kind } \text{fvariable} \text{ type.}
\]

\[
\text{type } \text{fvar} \text{ addr } \rightarrow \text{fvariable.}
\]

\[
\text{type } \text{hvariable} \text{ type.}
\]

\[
\text{type } \text{hvar} \text{ addr } \rightarrow \text{arity } \rightarrow \text{hvariable.}
\]

\[
\text{kind } \text{mapping} \text{ type.}
\]

\[
\text{type } \langle\rightarrow\rangle \text{ fvariable } \rightarrow \text{hvariable } \rightarrow \text{mapping.}
\]

\[
\text{type } \text{mmap} \ (\text{list mapping}).
\]

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

\[\text{Invariant 1 (Unification-variable arity). Each variable} \ \nu \ \text{ in} \ \ M \ \text{has a (unique) arity} \ \pi \ \text{and each occurrence} \ \text{(uva A L)} \ \text{is such that} \ \ L \ \text{has length} \ \pi.\]

\[
\text{type } \text{malloc} \text{ fvariable } \rightarrow \text{hvariable } \rightarrow \text{mmap } \rightarrow \text{mmap } \rightarrow \text{subst } \rightarrow \text{subst } \rightarrow \text{o. (malloc)}
\]

\[
\text{m-alloc} \ Fv \ Hv \ M \ S \ S1 : \text{- mem M (Fv } \rightarrow \text{Hv), 1.}
\]

\[
\text{m-alloc} \ Fv \ Hv \ M \ (\text{Fv } \rightarrow \text{Hv|M}) \ S \ S1 : \text{- Hv = hv N _, new N S S1. (malloc)}
\]

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

It is worth examining the code of deref, which applies the sub-
stitution 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.

\[
\text{type } \text{deref} \text{ subst } \rightarrow \text{tm } \rightarrow \text{tm } \rightarrow \text{o. (}\sigma\text{)}
\]

\[
\text{deref } _c \ (\text{con O}) \ (\text{con C}).
\]

\[
\text{deref } S (\text{app A}) (\text{app B}) : \text{- map (deref} S) A B.
\]

\[
\text{deref} S (\text{lam} F) (\text{lam} G):=\pi x \ (\text{deref} S x x \Rightarrow \text{deref} S (F x) (G x)).
\]

\[
\text{deref} S (\text{uva} N L) R : \text{- set? \ N S A,}\]

\[
\text{move} A L T, \text{deref} S T R.
\]

\[
\text{deref} S (\text{uva} N A) (\text{uva} N B) : \text{- unset? \ N S,}\]

\[
\text{map (deref} S) A B.
\]

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.

\[
\text{type } \text{move} \text{ assignment } \rightarrow \text{list tm } \rightarrow \text{tm } \rightarrow \text{o.}
\]

\[
\text{move} (\text{abs Bo} [\text{H}]) R L : \text{- move (Bo} H) L R.
\]

\[
\text{move} (\text{val} A) [\emptyset A : \text{- \_}].
\]

We write \(\sigma = \{ A_x y \mapsto y \} \) for the assignment \(\text{abs x} \text{abs y} \mapsto \text{y} \) and
\(\sigma = \{ A \mapsto \lambda x y. y \} \) for \(\text{\lambda x }\text{\lambda y} \mapsto \text{y} \).

3.2 Links \((L)\)

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

\[
\text{kind } \text{baselink} \text{ type.}
\]

\[
\text{type } \text{link-eta} \text{ tm } \rightarrow \text{tm } \rightarrow \text{baselink.}
\]

\[
\text{type } \text{link-lam} \text{ tm } \rightarrow \text{tm } \rightarrow \text{baselink.}
\]

\[
\text{typeabbrev links (list (inc}tx \text{ baselink))}.\]

The right-hand side of a link, the problematic term, can occur
under binders. To accommodate this situation, the compiler wraps
baselink using the inc}tx container (see \(\cdot\)\(\cdot\)\(\cdot\) also used for subst).

\[
\text{Invariant 2 (Link left hand side). The left-hand side of a suspended link is a variable.}
\]

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 \(x \mapsto \text{Ax} = \text{L} F x \mapsto a\) corresponds to:

\[
\text{abs x} \text{\_ val (link-lam (uva A [x]) (app[uva F [x],con "a"]))}
\]

3.3 Compilation

The simple compiler described in this section serves as a basis
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 using (malloc).

The signature of the \( \text{comp} \) 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. With respect to section 2, the signature also allows for updates to the substitution.

\[
\text{type comp} \rightarrow \text{tm} \rightarrow \text{mmap} \rightarrow \text{mmap} \rightarrow \text{links} \rightarrow \text{links} \rightarrow \text{sub} \rightarrow \text{sub} \rightarrow \text{o}.
\]

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

\[
\text{comp} \text{(flam} F \text{) (lam} F1 \text{) M1 M2 L1 L2 S1 S2} \rightarrow (cA).
\]

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

\[
\text{close-links} \text{L R}.
\]

\[
\text{type close-links} \rightarrow \text{links} \rightarrow \text{links} \rightarrow \text{o}.
\]

\[
\text{type hstep} \rightarrow \text{tm} \rightarrow \text{links} \rightarrow \text{links} \rightarrow \text{sub} \rightarrow \text{sub} \rightarrow \text{o}.
\]

\[
\text{hstep} \text{T1 T2 L1 L2 S1 S3} \rightarrow
\]

\[
\text{progress} L1 L2 S2 S3.
\]

Note that the infix notation \((A \equiv_m B) \in D\) is syntactic sugar for \((\langle \equiv_m \rangle) A B C D\).

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

\[
\text{type progress links} \rightarrow \text{links} \rightarrow \text{sub} \rightarrow \text{sub} \rightarrow \text{o}.
\]

\[
\text{progress} L L3 S S3 \Rightarrow
\]

\[
\text{occur-check-links} L1 \text{L2 S1 S2}
\]

\[
\text{scope-check} L1 \text{L2 S1 S2}
\]

\[
\text{if} (L = L2, S = S2) \text{(L3 = L2, S3 = S2)}
\]

\[
\text{(progress} L2 L3 S2 S3).
\]

3.4 Execution

A step in \( M \) consists of 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.

\[
\text{type hstep} \rightarrow \text{tm} \rightarrow \text{links} \rightarrow \text{links} \rightarrow \text{sub} \rightarrow \text{sub} \rightarrow \text{o}.
\]

\[
\text{hstep} T1 T2 L1 L2 S1 S3 \Rightarrow
\]

\[
\text{(T1} \equiv_m \text{T2) S1 S2,}
\]

\[
\text{progress} L1 L2 S2 S3.
\]
Higher-Order unification for free! Conference ’17, July 2017, Washington, DC, USA

Definition 3.4 \((W_P)\). \(W_P(X) \iff \forall t \in \mathbb{P}(X), t \not\in (\eta \cup \downarrow \beta)\)

Definition 3.5 \((W_{\eta})\). \(W_{\eta}(X) \iff \forall t \in \mathbb{P}(X), t \not\in \downarrow \eta\)

4 HANDLING OF \(\downarrow \eta\)

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

\[\mathcal{P} = \{ \lambda x. f (X x) a \} \}
\[\mathcal{Q} = \{ \lambda x. f (A x) a \} \}
\[\mathcal{M} = \{ X \mapsto A^0 \} \]

The unification problem \(\mathcal{Q}\), while trying to unify \(A x\) and \(X\), is equivalent to \(\text{app} \ [\text{uva} A \ [\text{x}]]\) 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 \(\text{cO}\), where pattern-fragment is a predicate checking if a list of terms is a list of distinct names.

\[\text{comp} (\text{fapp} \ [\text{uva} A \ [\text{Ag}])] \text{ (uva B Ag1)} \ M2 \ L \ L S1 S1 S2 \ :-\]
\[\text{pattern-fragment} \ Ag, 1, \]
\[\text{fold} 6 \text{ comp} Ag \ Ag1 \ M1 \ M2 \ L \ L S1 S1 S2, \]
\[\text{len} \ Ag \text{ Arity}, \]
\[\text{m-alloc} \ (\text{fv} \ A) \ (hv \ B \ (\text{arity Arity})) \ M1 \ M2 \ S1 S2.\]

Note that compiling \(Ag\) cannot create new mappings nor links, since \(Ag\) is made of bound variables, and the hypothetical rule \((H_3)\) loaded by \(\text{comp-\&m} \) grants this property.

Decompilation. No change to \(\text{commit-link}\) since no link is added.

Progress. No change to \(\text{progress}\) since no link is added.

Lemma 4.1 (Properties of \(\approx_{\eta}\) in \(W_{\eta}\)). Given the updated compilation scheme, the \(\approx_{\eta}\) of section 3.6 is a good unification (as per proposition 2.1) for \(\approx_{\eta}\) in the domain \(W_{\eta}\).

Proof sketch. If we look at the \(O\) terms, there is one more interesting case, namely \(\text{fapp}[\text{uva} X] [\mathcal{W}]\) when \(\mathcal{W}\) are distinct names compiled to \(\mathcal{W}\). In this case the \(M\) problem is \(Y_{\mathcal{W}} \equiv m t\) that succeeds with \(\sigma = \{ Y_{\mathcal{W}} \mapsto t \ (\mathcal{W}[\mathcal{Y}])\}\), which in turn is decomposed to \(\rho = \{ Y \mapsto \lambda \mathcal{Y}. s (\mathcal{W}[\mathcal{Y}])\}\). Thanks to \(\beta_1(\lambda \mathcal{Y}. s (\mathcal{W}[\mathcal{Y}])\) \(= s\).

Lemma 4.2 (\(W\)-partial-enforcement). \(\forall s, (s) \mapsto (t, m, l), \text{ if } W_P(s) \text{ then } W(t)\).

In other word the compiler eliminates all subterms that are in \(\downarrow \beta\).

Theorem 3.4 (Fidelity in \(W_{\eta}\)). Proposition 2.2 (Run equivalence) and proposition 2.3 (Simulation fidelity) hold if \(W_{\eta}(\mathcal{P})\).

Proof sketch. Thanks to lemma 4.2 \(\approx_m\) is as powerful as \(\approx_{\eta}\) in \(\downarrow \beta\), as well as in \(\downarrow \eta\) by lemma 4.1.

5 HANDLING OF \(\downarrow \eta\)

A term \(\lambda x. t\) \(x\) is said to be the \(\eta\)-expansion of \(t\) if \(x\) does not occur free in \(t\), and conversely, \(t\) is the \(\eta\)-contraction of \(\lambda x. t\) \(x\). The equational theory of \(O\) identifies these terms, but the current compilation scheme does not, as shown by the following example: while

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 \(O\) while \(Q\) their corresponding compiled version with memory mapping \(\mathcal{M}\) and links \(L\). For example:

\[\mathcal{P} = \{ p1 \approx_{\eta} p2 \}
\[\mathcal{Q} = \{ t1 \equiv_{\mathcal{M}} t2 \}
\[\mathcal{M} = \{ X1 \mapsto A^0_1 \}
\[\mathcal{L} = \{ t \mapsto a \equiv_{\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, \(\mathcal{Q}\) corresponds to the \(M\) problem \(t1 \equiv_{\mathcal{M}} t4\).

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

Lemma 3.2 (Properties of \(\approx_{\eta}\) in \(W_P\)). The implementation of \(\approx_{\eta}\) above is a good unification for \(\approx_{\eta}\) (as per proposition 2.1) in the domain \(W\).

Proof sketch. In this setting, \(\approx_m\) is as strong as \(\approx_{\eta}\) on ground terms. What we have to show is that whenever two different \(O\) terms can be made equal by a substitution \(\rho\) (plus the \(\beta_1\) and \(\beta_r\) if needed), we can produce this \(\rho\) by finding a \(\sigma\) via \(\approx_m\) on the corresponding \(M\) terms and by decompiling it. If we look at the syntax of \(O\), terms the only interesting case is \(\text{app}[\text{uva} X] \approx_{\eta}\). In this case, after compilation, we have \(\text{uva} Y \approx_{\mathcal{M}} t\) that succeeds with \(\sigma = \{ Y \mapsto t\}\) and \(\sigma\) is decompiled to \(\rho = \{ X \mapsto s\}\) by proposition 3.1.

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

Proof sketch. Since \(\text{progress}\) is a no-op then ostepp and mstep are the same. By lemma 3.2, \(\approx_m\) is equivalent to \(\approx_{\eta}\) in \(W\).
\begin{align*}
\lambda x. X x \approx_\eta f & \text{ does admit the solution } \rho = \{ X \mapsto f \}, \text{ the corresponding problem in } Q \text{ does not.} \\
P &= \{ \lambda x. X x \approx_\eta f \} \\
Q &= \{ \lambda x. Ax \approx_m f \} \\
M &= \{ X \mapsto A^1 \}
\end{align*}

The reason is that \( \lambda m x \ x uva A [x] \) and \( \text{con"f"} \) start with different, rigid, 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 \( P \) above is refined to:

\begin{align*}
P &= \{ \lambda x. X x \approx_\eta f \} \\
Q &= \{ A \approx_m f \} \\
M &= \{ X \mapsto B^1 \} \\
L &= \{ \nu A \eta \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 \( \odot \eta \) that has the following property:

IVARIANT 3 (\( \eta \)-link RHS). The rhs of any \( \eta \)-link has the shape \( \lambda x.t \) and it is \( W \).

Each \( \eta \)-link is kept in the link store \( L \) during execution and is activated under some conditions. Activation is implemented by extending the progress predicate defined in section 3.4.

### 5.1 Detection of \( \odot \eta \)

When compiling a term \( t \), we need to determine if any subterm \( s \in \mathcal{P}(t) \) of \( t \) of the form \( \lambda x.r \), where \( x \) occurs in \( r \), can be a \( \eta \)-expansion, i.e. if there exists a substitution \( \rho \) such that \( \rho(\lambda x.r) \approx_\eta s \).

The detection of lambda abstractions that can “disappear” is not as trivial as it may seems. Here a few examples:

\begin{align*}
\lambda x.f \ (A \ x) & \in \odot \eta \quad \rho = \{ A \mapsto \lambda x.x \} \\
\lambda x.f \ (A \ x) \ x & \in \odot \eta \quad \rho = \{ A \mapsto \lambda x.a \} \\
\lambda x.f \ (A \ x) \ A & \notin \odot \eta \\
\lambda x.\lambda y.f \ (A \ x) \ (B \ y \ x) & \in \odot \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 \( \eta \)-expansion or not.

The third example shows that when a variable occurs outside the scope of a unification variable, it cannot be erased and can hence prevent a term from being an \( \eta \)-expansion.

The last example shows the recursive nature of the check we need to implement. The term starts with a pair of two lambdas, hence the whole term is in \( \odot \eta \) iff the inner term \( \lambda y.f \ (A \ x) \ (B \ y \ x) \) is in \( \odot \eta \) itself. If it is, it could \( \eta \)-contract to \( f \ (A \ x) \) making \( \lambda x.f \ (A \ x) \) a potential \( \eta \)-expansion.

We can now define more formally how \( \odot \eta \) terms are detected together with its auxiliary functions:

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

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

**Proof sketch.** Since our terms are in \( \beta \)-form there is only one rule that can play a role (namely \( \eta_1 \)), hence if the term \( s \) is not exactly \( x \) (case 1) it can only be an \( \eta \)-expansion of \( x \), or a unification variable that can be assigned to \( x \), or a combination of both. If \( s \) begins with a lambda, then the lambda can only disappear by \( \eta \) contraction. In that case, the term \( t \) is under the spine of binders \( x_1 \ldots x_n \) can either be \( x \) applied to terms that can may-contract-to these variables (case 2), or a unification variable that can be assigned to that application (case 3).

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

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 \( t \). Moreover, 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 \( \odot \eta \) detection:

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

1. \( t \) is a constant or a name applied to the arguments \( l_1 \ldots l_m \) 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 \( l_1 \ldots l_{m-n} \);
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 (\( \odot \eta \) detection).** If \( t \) is a \( \beta \)-normal term and \( t \in \odot \eta \) 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 \( \odot \eta \), i.e. there exists no substitution \( \rho \) such that \( \rho t \) is an \( \eta \)-expansion. A simple counter example is \( \lambda x. f \ (A \ x) \ (A \ x) \) since \( x \) does not 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.

\begin{align*}
\text{comp} \ (\lambda m \ F \ (uva \ A \ \text{Scope}) \ M1 \ M2 \ L1 \ L3 \ S1 \ S3) \ :- \\
\text{maybe-eta} \ (\lambda m \ F \ (\lambda m \ F1)) \ [1], \\
\text{alloc} \ S1 \ A \ S2, \\
\text{comp-lam} \ F \ F1 \ M1 \ M2 \ L1 \ S2 \ S3, \\
\text{get-scope} \ (\lambda m \ F1) \ \text{Scope}, \\
L3 = [\text{val} \ (\text{link-eta} \ (\lambda m \ A \ \text{Scope}) \ (\lambda m \ F1)) \ | \ L2].
\end{align*}
When Π is an n-lh of F and Φ is a variable, the unification problem becomes (Π, Φ) = (Π, Φ). By invariant 3, we can unify the Π and Φ by renaming them to Π₁ and Φ₁, respectively.

**Lemma 5.10.** Let Π, Φ be a pair of terms in W, then W(Π), enforcing invariant 3.

**Proof sketch.** By construction, every “problematic” term in O is replaced with a variable in the corresponding M term. Therefore, W(Π).

**Lemma 5.11.** Given a Π-answer Γ ⊢ X = Π x.t, the unification done by η-progress-lh is between terms in W.

**Proof sketch.** Let be the substitution, such that W(σX) (since η preserves W). By invariant 3, we have W(Π). If σX = Π x.t, then r is unified with t and both terms are in W. Otherwise, Π x.t is unified with x.t, and 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 Π-unifications Γ₁ ⊢ X ≈ Π λ x.t and Γ₂ ⊢ Y ≈ Π λ x.t', the unification is performed between t and t'. Both terms are Π and therefore, unification is done between Π terms.

**Lemma 5.13.** The progress of Π-unification guarantees proposition 2.11 (W-preservation).

**Proof sketch.** By lemmas 5.11 and 5.12, every unification performed by the activation of a Π-unification is done between terms in W.

**Lemma 5.14.** Progress terminates.

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

**Theorem 5.15 (Fidelity in W).** Given a list of unification problems P, such that W(Π), if the memory map is bijective, then the introduction of Π-unification guarantees proposition 2.3 (Simulation fidelity).

**Proof sketch.** We want to prove that step succeeds iff mstep succeeds, where mstep is made by a unification step u and a link progression p. Without loss of generality we consider an Π-unification problem of the form s₁ = Π s₂ where s₁ ∈ Φ and is compiled to a variable X with the accessor link Γ ⊢ X = Π λ x.t₁. The unification u always succeeds, since X is a fresh variable, we only have to consider the link progression p.

- s₂ ∈ Φ, in this case s₂ is compile to t₂ and the unification step u assigns t₂ to X and triggers Π-progress-lh. If s₁ and s₂ unify thanks to η then progression p unifies λ x.t₁ with the η-expansion of t₂ namly λ x.t₂, otherwise they unify thanks to Π and the link progress p unifies λ x.t₁ with t₂.
- Alternatively, if s₁ and s₂ fail to unify, it cannot be because of the Π constructor in the head of t₁, since Π or η try to unify under it, eventually finding two different subterms t₂ and t₂' by theorem 4.3. The unification fails on terms t₁ and t₂' even if Π-progress-lh η-expands t₂.
- s₂ ∈ Φ, in this case, the unification problem becomes X = Π Y with the extra link Y = Π λ x.t₂. After the unification of X and Y, Π-progress-deduplicate is fired and, once the λ are crossed, t₁ and t₂ are unified. This unification succeeds iff s₁ = s₂ succeeds by theorem 4.3.

**Corollary 5.16 (Properties of Π in W(Π)).** Given the updated compilation scheme, the Π of section 3.6 is a good unification (as per proposition 2.1) for Π in the domain W(Π).

**Example of Π-progress-lh.** The example at the beginning of section 5, once σ = { A → f }, triggers Π-progress-lh since the link becomes t = f = λ x.Bx and the lhs is a constant. This rule runs λ x.f x = Π λ x.Bx, resulting in σ = { A → f ; Bx → f }. By the definition of Π-unification decomposition, decomposition generates p = { X → λ x.f x }, since X is mapped to Bx and Bx is decomposed into λ x.f x.
6 MAKING \( M \) 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. Then it is the case the memory map \( M \) generated by the compiler is not a bijection. For example:

\[
P = \{ \lambda x.(X x) \approx_0 a \quad X \approx_0 \lambda x.a \}
\]

\[
Q = \{ A \approx_m a \quad C \approx_m \lambda x.a \}
\]

\[
M = \{ X \mapsto C^0 \quad X \mapsto B^1 \}
\]

\[
L = \{ \triangleright A \approx_\eta \lambda x.B_x \}
\]

in the unification problems \( P \) above, \( X \) is used with arity 1 in \( P_1 \) and with arity 0 in \( P_2 \). 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, breaking, in turn proposition 2.3. decompilation phase. To address this issue, we post-process the compiler’s output to ensure the following property:

**Proposition 6.1 (\( M \) is a bijection).** After compilation, for each \( O \) variable \( X \) in \( P \) and for each \( M \) variable \( A \) in \( Q \) there is exactly one entry \( X \mapsto A^m \) (for any arity \( n \)).

Note that the execution in \( M \) may require allocating new variables as explained in section 3.5. The property above does not concern 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 \( m_2 : X \mapsto C^n \) where \( m < n \) and \( d = n - m \), align-arity \( m_1 \) \( m_2 \) generates the following \( d \) links, one for each \( i \) such that \( 0 \leq i < d \),

\[
x_0 \ldots x_{m+i} \mapsto B_i^{d} \mapsto \lambda x_{m+i+1}.B_{i+1}^{d} \mapsto \ldots \mapsto x_{m+i+n}
\]

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 higher arity to match the lower arity. Since each \( \eta \)-link can add exactly one lambda, we need as many links as the difference between the two arities.

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

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

**Proof sketch.** By the definition of map-deduplication, any two occurrences of the same \( O \) variable used at different arities are compiled to two variables \( A^m \) and \( C^n \) related by a chain of \( \eta \)-links. Without loss of generality, we pose \( m < n \). If \( A^m \) is assigned to a term \( \lambda \tilde{x}.t \) with \( q \) being the length of \( \tilde{x} \). If \( t \) is a rigid term, then the chain of links between \( A^m \) and \( C^n \) is completely triggered so that \( C^n \) is assigned to \( \lambda \tilde{x}.t \) \( \eta \)-expanded \( n - m \) times. Note that in this case fidelity is guaranteed, since unification to \( C^n \) sohuld also unify with \( A^m \). Otherwise, if \( t \) is a variable, the chain of links is shrunk by \( q \). Note that if this length is bigger then \( n - m \) again the full chain collapse and \( A^m \) and \( C^n \) are again tiny coupled. Otherwise, if \( q < n - m \), it means that \( C^n \) can always be assigned to any term starting with at least \( q \) lambdas. In this second case fidelity is also guaranteed, due to the constraint of the number of lambda by which \( C^n \) should start.

**Note that shortening the chain of links allows unification to fail if \( X_1 \) and \( X_2 \) are unified with different terms.**

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

\[
P = \{ \lambda x.(X x) \approx_0 a \quad X \approx_0 \lambda x.a \}
\]

\[
Q = \{ A \approx_m a \quad C \approx_m \lambda x.a \}
\]

\[
M = \{ X \mapsto B^1 \}
\]

\[
L = \{ \triangleright C \approx_\eta \lambda x.B_x \quad \triangleright A \approx_\eta \lambda x.B_x \}
\]

Note that \( X \mapsto B^1 \) disappears from \( M \) in favour of the auxiliary \( \eta \)-link \( x \mapsto E_x \approx_\eta \lambda y.X_y \). The resolution of \( Q_1 \) assigns \( a \) to \( A \). This wakes up \( L_2 \) by \( \eta \)-progress-lhs, assigning \( a \mapsto B_x \). The resolution of \( Q_2 \) instantiates \( C \mapsto \lambda x.a \), which, in turn triggers \( L_1 \) by \( \eta \)-progress-lhs. This link perfoms \( \lambda x.a \approx_m \lambda x.(a.x) \), which, as expected, fails.

7 HANDLING OF \( \psi \mathbb{L} \)

Even if we do not require \( \approx_m \) to handle problems outside of \( \mathbb{L} \) it is worth handling terms in \( \psi \mathbb{L} \) since, in practice, these terms often re-enter \( \mathbb{L} \) at runtime, as observed in [15] for example, or to accommodate for the introduction of heuristics.

In the following example problem \( P_2 \) is \( X \mapsto a \approx_0 \) that admits two different substitions: \( \rho_1 = \{ X \mapsto \lambda x.x \} \) and \( \rho_2 = \{ X \mapsto \lambda x.a \} \). The unification algorithm alone cannot chose, since no solution is more general than the other, although the first problem solve this dilemma at run-time making the second problem well behaved:

\[
P = \{ X \approx_0 \lambda x.a \quad (X x) \approx_0 a \}
\]

\[
Q = \{ A \approx_m \lambda x.a \quad (A x) \approx_m a \}
\]

\[
M = \{ X \mapsto A^0 \}
\]

We want to support this scenario but we have to improve a little our compiler and progress routines. In particular \( Q_1 \) generates \( \sigma = \{ A \mapsto \lambda x.a \} \) making \( \sigma Q_2 \) equal to \( (\lambda x.a) a \approx_m a \) that \( \approx_m \) cannot solve since it lacks rules \( \beta_1 \) and \( \beta_2 \).

To address this problem the compiler must recognize and replace \( \psi \mathbb{L} \) terms with fresh variables and generate a new kind of links that we call \( \psi \)-link.

In addition to invariant 2 (Link left hand side), the term on the rhs of a \( \psi \)-link has the following property:

**Invariant 4 (\( \psi \)-link rhs).** The rhs of any \( \psi \)-link has the shape \( X_1 \ldots t_m \) where \( X \) is a unification variable in \( \mathbb{L} \) (with scope \( s_1 \ldots s_n \)) and \( t_1 \ldots t_m \) is a list of terms such that \( \lambda > 0 \) and \( t_i \) is either a variable occurring in \( s_1 \ldots s_n \) or a term other than a variable.

Note that the shape of such as rhs is \( \langle uva X S \mid L \rangle \), where \( S = s_1 \ldots s_n \) and \( L = t_1 \ldots t_m \).
7.1 Compilation and decomposition

By inserting this rule before rule (cgb), we identify terms \( \diamond L \), i.e. flexible applications with arguments that are not distinct names:

\[
\text{comp} \text{ (app fapp \{fuva A\Ag\}) (uva B Scope) \{M1 M3 L1 L3 S1 S4 := l, pattern-fragment-prefix Ag PF Extra, alloc SI B S2, len PF Ar, m-alloc (fv A) (hv C (arity Ar)) M1 M2 S2 S3, fold6 comp PF PF1 M2 L1 L1 S1 S3, fold6 comp Extra Extral M3 L1 L2 S3 S4, Beta = app (uva C PF1 | Extra1), get-scote Beta Scope, L3 = [val (link-llam (uva B Scope) Beta) | L2].}
\]

The list Ag is split into the two parts: PF is in L, and can be empty: Extra cannot be empty and is such that append PF Extra Ag. The rhs of the \( L \)-link is the application of a fresh variable \( \eta \) having in scope all names in PF1 (the compilation of PF). The variable B, returned has the compiled term, is a fresh variable having in scope all the free variables occurring in PF1 and Extral (the compilation of Extra). This construction enforces invariant 4.

**Lemma 7.1 (W-enforcement).** \( \forall s \in L \), then \( W(t) \).

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

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

7.2 Progress

Let \( l \) be a \( L \)-link of the form \( \Gamma \vdash T = \lambda X_{s_1} \ldots s_n t_1 \ldots t_m \)

**Definition 7.2 (\( L \)-progress-refine).** Let \( \sigma \) be a substitution such that \( \sigma_i \) is a name not occurring. If \( \eta = \sigma \), then \( l \) is removed and lhs is unified with \( X_{s_1} \ldots s_n \). If \( m = 1 \), then \( l \) is replaced by the link \( \Gamma \vdash T = L X_{s_1} \ldots s_n z t_2 \ldots t_m \) (where \( Y \) is a fresh variable of arity \( n + 1 \)) and link \( \Gamma \vdash X_{s_1} \ldots s_n = \lambda x Y x_{s_1} \ldots s_n x \) is added to \( L \).

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

**Definition 7.4 (\( L \)-progress-fail).** progress fails whenever there exists another link \( l' \in L \) with the same lhs or the lhs become rigid.

We relax this condition and accommodate for heuristics in 7.3.

Finally, in order to correctly correlate a variable appearing in the lhs of a \( \eta \)-link and in the rhs of a \( L \)-link, we should introduce the following \( \eta \)-link projection rule.

**Definition 7.5 (\( \eta \)-progress-rhs).** A link \( \Gamma \vdash X = \eta T \) is removed from \( L \) when either 1) the \( \eta \)-eta rule does not hold (anymore) or 2) by \( \eta \)-contracting \( T \) to \( T' \) where \( T' \) does not start with the \( \lambda \) constructor. In the first case, \( X \) is unified with \( T \), and in the second one, \( X \) is unified with \( T' \) (under the context \( \Gamma \)).

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

**Lemma 7.6.** \( \theta \)-progress terminates

**Proof sketch.** Let \( l \) a \( L \)-link in the store \( \Gamma \). If \( l \) is activated by \( L \)-progress-rhs, then it disappears from \( \Gamma \) and \( \theta \)-progress terminates. Otherwise, the rhs of \( l \) is made by a variable applied to \( m \)

arguments. At each activation of \( L \)-progress-refine, \( l \) is replaced by a new \( L \)-link \( l' \) having \( m = 1 \) arguments. At the \( m \)-th iteration, the \( L \)-link \( l'' \) has no more arguments and is removed from \( L \). Note that at the \( m \)-th iteration, \( m \) new \( \eta \)-link have been added to \( L \), however, by lemma 5.14, the algorithm terminates. \( L \)-progress-fail guarantees termination since it immediately fails and \( \eta \)-progress-rhs terminates since it executes terminating instructions and if its permisces succeed, then considered \( \eta \)-link is removed from \( L \).

**Theorem 7.7 (Fidelity in \( L \)).** The introduction of \( L \)-link generates proposition 2.4 (Fidelity Recovery) if \( \eta \) fails outside \( L \).

**Proof sketch.** Let \( P_i \) be the first problem in \( P \), such that it is not in \( L \) and le \( \sigma \) be the substitution obtained solving \( Q_1 \ldots Q_1 \). After the execution of \( \text{mstf}_{P_{i-1}} \), each variable \( X \) corresponding to a \( \eta \)-term suffer in the original Unification problem, is instantiated if it is known for sure that \( X \) is no more a \( \eta \)-term. If \( \sigma Q_i \) is in \( L \), then by definitions 7.2 and 7.3 the \( L \)-link is solved and removed. Note that, in this case, all calls to \( \Rightarrow m \) are between terms in \( L \) and by theorem 4.3 fidelity is guaranted. If \( \sigma Q_i \) is still in \( \eta \), then by definition 7.4 unification fails as the corresponding unification in \( O \) would.

\( \square \)

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 \( \lambda \)Prol [16] and Teyjus [15] delay the resolution of \( \diamond L \) unification problems until the substitution makes them reenter \( L \). Other systems, for example of the unification algorithm of Coq used in its type-class solver [23], apply heuristics like preferring projection over mimic, and commit to that solution.

In this section we show how we can implement these strategies by simply adding (or removing) rules to the progress predicate. In the example below \( P_1 \) is in \( \diamond L \).

\[
\begin{align*}
P &= \{ (X a) \Rightarrow a \quad X \Rightarrow \lambda x Y \} \\
Q &= \{ A \Rightarrow m a \quad B \Rightarrow m \lambda x C \} \\
M &= \{ Y \Rightarrow C^0 \quad X \Rightarrow B^0 \} \\
L &= \{ \vdash A \Rightarrow L \quad (B a) \}
\end{align*}
\]

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 \( L_1 \) 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 \( \text{occur-check-links} \) 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 \( \text{commit-links} \) should be modified accordingly.

If instead we want \( \Rightarrow m \) 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 \( X \) to \( Y \). This heuristic is used by [23].
8 ACTUAL IMPLEMENTATION IN ELPI

In this paper we did study a compiler and 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 M as well.

The main difference is that we cannot implement mrun since it comes as part of the runtime of the programming language. In particular the runtime iterates \( \simeq \), but mstep also needs to check for progress. Luckily Elpi extends \([10]\) JProlog with syntactic constraints (suspended goals) and Constraint Handling Rules (CHR) \([7]\) to operate on them. An important design choice is that constraints are suspended on a user computed list of unification variables, and are resumed as soon as one of them is assigned, before any other existing goal is considered. In turn this property enables a timely activation of links that grants proposition 2.3 (SIMULATION FIDELITY). For brevity we only provide two pseudo-code snippets. The first one depicts how \( \eta \)-link are suspended or make progress.

\[
\begin{align*}
\text{link-eta } L & := \text{var } L, \text{ maybe-eta } R, \text{!,} \\
& \text{declare_constraint (link-eta } L \text{) } [L,R], \\
& \text{link-eta } L \text{ R := not } (\text{var } L), \text{!, eta-progress-lhs } L \text{ R.} \\
& \text{link-eta } L \text{ R := not } (\text{maybe-eta } R), \text{!, contract } R \text{ S, L = S.}
\end{align*}
\]

The second snippet illustrates the deduplication of \( \eta \)-link. The syntax \( \langle N \mapsto G ?-? \rangle \) denotes a JProlog sequent, that is a goal \( P \) under a set \( G \) of hypothetical rules (introduced by \( \Rightarrow \)) and where the program context binds (via the \( \text{pi} \) operator) \( N \) heigenvariables. Sequents are presented to CHR with their higher order unification variables replaced by "frozen constants", that is \( \lambda_{x,y,z} \) becomes \( \text{uvar } f_3 \langle x,y,z \rangle \text{ for a fresh constant } f_3 \). Frozen constants are "defrost" when they are part of a new goal (see \([10\), section 4.3]).

\[
\begin{align*}
\text{rule (N1 } & \Rightarrow G1 ?-\text{ link-eta (uvar } X \text{ LX1) T1) } \text{ % match} \\
& \langle N2 \Rightarrow G2 ?-\text{ link-eta (uvar } X \text{ LX2) T2) } \text{ % remove} \text{ } \\
& \langle \text{match LX1 LX2 T2 T2'} \rangle \text{ % condition} \text{ } \\
& \langle\Rightarrow (N1 \Rightarrow G1 ?- T1 = T2') \rangle \text{ % new goal}
\end{align*}
\]

The first directive matches a constraint whose lhs is a variable \( X \); the second mathes 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 \( \simeq \) as a regular routine, i.e. rule write as follows

decision \( X := \text{ unify } X \) \( (\text{all } A \ x) \text{ app } [P, x]) \), finite \( A \), \( \pi_1 \text{ decision (app } [P, x]) \).

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 data from 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.

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

decision \( (\text{all } A \ x \ P \ x) := \text{ finite } A \), \( \pi_1 \text{ decision } (P \ x) \).

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 for that one 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 \text{ 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{ 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 JProlog programs) with the one of CC. The encoding is hence tailed 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 \([29]\] 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 only 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 XXX nodes.

Our approach is flexible enough to accommodate different strategies and heuristics to handle terms outside the pattern fragment. Moreover, the unification process we obtain is ready to take advantage of potential improvements to the programming engine, such as tabled search. Finally, our shallow encoding can enable to apply forms of static analysis for the meta language, such as determinacy, also to the object language.

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 combinators like \text{map} 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 exploratory phase.
REFERENCES


