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-04547069

HAL Id: hal-04547069
https://inria.hal.science/hal-04547069
Preprint submitted on 15 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
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

1 INTRODUCTION

Meta languages such as Elf [18], Twelf [20], λProlog [13], and Isabelle [26] have been utilized to specify various logics [5, 16, 17, 4]. The use of these meta languages facilitates this task in two key ways. The first and most well-know one is that variable binding and substitution come for free. The second one is that these meta languages come equipped with some form of unification, a cornerstone of proof construction and proof search.

The object logic we are interested in is Coq’s Calculus of Inductive Constructions (CIC), and we aim to implement a form of proof search known as type-class [25, 21] resolution. Type-class solvers are unification based on proof search procedures reminiscent of Prolog, which back-chain lemmas taken from a database of “type-class instances”. Given this analogy with Logic Programming we want to leverage the Elpi [23] meta-programming language, a dialect of λProlog, already used to extend Coq in various ways [23, 22, 8, 6]. In this paper, we focus on one aspect of this work, precisely how to reuse the higher-order unification procedure of the meta language in order to simulate a higher-order logic program for the object language.

We take as an example the Decision and Finite type classes from the Stdpp [11] library. The class Decision identifies predicates equipped with a decision procedure, while Finite identifies types whose inhabitants can be enumerated in a (finite) list. The following three type-class instances state that: 1) the type of natural numbers smaller than n, called fin n, is finite; 2) the predicate nfact n nf, relating a natural number n to the number of its prime factors nf, is decidable; 3) the universal closure of a predicate has a decision procedure if its domain is finite and if the predicate is decidable.

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

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

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

The proof found by the solver back-chains on rule 3 (the only rule about the ∀ quantifier), and then solves the premises with rules 1 and 2 respectively. Note that rule 3 features a second-order parameter P that represents a function of type A → Prop (a 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 = nfact x 3. This higher-order problem falls in the so-called pattern-fragment L [12] and admits a unique solution ρ that assigns the term λ x. nfact x 3 to P.
In order to implement such a search in Elpi, we shall describe the encoding of CIC terms and then the encoding of instances as rules. Elpi comes equipped with an Higher Order Abstract Syntax (HOAS [19]) datatype of CIC terms, called tm, that includes (among others) the following constructors:

- **type lam**: `tm -> (tm -> tm) -> tm`, % lambda abstraction
- **type app**: `list tm -> tm`, % n-ary application
- **type all**: `tm -> (tm -> tm) -> tm`, % forall quantifier
- **type con**: `string -> tm`, % constants

Following the standard syntax of λProlog [13], the meta-level binding of a variable x in an expression e is written as «x \ e», while square brackets delimit a list of terms separated by comma. For example, the term «V y, nfact y 3» is encoded as follows:

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

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

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

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

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

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

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

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

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

Since \(Pm\) is a higher-order unification variable of type \(\text{tm} \rightarrow \text{tm}\), with \(x\) in its scope, the unification problem \((p')\) admits one solution:

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

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

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

This simple example is sufficient to show that the encoding we seek is not trivial and does not only concern the head of rules, but the entire sequence of unification problems that constitute the execution of a logic program. In fact the solution for \(P\) above generates a \(\text{Coq}\) \(β\)-redex in the second premise (the predicate under the \(\pi\) \(\|\) \(\pi\)).

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

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

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

**Contributions.** In this paper we identify a minimal language \(\mathcal{F}_0\) in which the problems sketched in the introduction can be formally described. We detail an encoding of a logic program in \(\mathcal{F}_0\) to a strongly related logic program in \(\mathcal{H}_0\) (the language of the meta-language) and we show that the higher-order unification procedure of the meta language \(\mathcal{H}_0\) can be efficiently used to simulate a higher-order unification procedure \(\mathcal{H}_0\) for the object language that features \(ηβ\)-conversion. We show how \(\mathcal{H}_0\) can be extended with heuristics to deal with problems outside the pattern fragment.

Section 2 formally states the problem and gives the intuition behind our solution; section 3 sets up a basic simulation of first-order logic programs, section 4 and section 5 extend it to higher-order logic programs in the pattern fragment while section 7 goes beyond the pattern fragment. Section 8 discusses the implementation in Elpi. The λProlog code discussed in the paper can be accessed at the address https://github.com/FissoreD/ho-unif-for-free.

## 2 PROBLEM STATEMENT AND SOLUTION

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

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

To reason about unification, we provide a description of the \(\mathcal{F}_0\) and \(\mathcal{H}_0\) languages where unification variables are first-class terms, i.e. they have a concrete syntax as shown in fig. 1. Unification variables in \(\mathcal{F}_0\) have explicit scope.

```
```

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

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

Figure 1: The \(\mathcal{F}_0\) and \(\mathcal{H}_0\) languages
Notational conventions. When we write $\mathcal{H}_0$ terms outside code blocks we follow the usual $\lambda$-calculus notation, reserving $f, g, a, b$ for constants, $x, y, z$ for bound variables and $X, Y, Z, F, G, H$ for unification variables. However, we need to distinguish between the "application" of a unification variable to its scope and the application of a term to a list of arguments. We write the scope of unification variables in subscript while we use juxtaposition for regular application. Here are few examples:

$$f\ a$$ app [con "f", con "a"]
$$\lambda x.\lambda y. \ F\ x\ y$$ app [uva $F$, con "x", con "y"]
$$\lambda x.\ F\ a$$ app [uva $F$, con "a"]
$$\lambda x.\ F\ x\ x$$ app [uva $F$, con "x", con "x"]

When it is clear from the context, we shall use the same syntax for $\mathcal{T}_0$ terms (although we never unifies variables). We use $s, t_0, \ldots$ for terms in $\mathcal{T}_0$ and $t, t_1, \ldots$ for terms in $\mathcal{H}_0$.

### 2.2 Equational theories an unification

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

#### 2.2.1 Term equality: $\equiv$ and $\simeq$

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, $\equiv_\alpha$ has rules for $\eta$ and $\beta$-equivalence.

$$\text{type } (\equiv_\alpha) \ \text{fm} \rightarrow \text{fm} \rightarrow_\alpha \text{ o}.$$  
$$f\ con =_\alpha f\ con\ X.$$  
$$\text{app } A =_\alpha \text{app } B :- \text{forall2 } (\equiv_\alpha) \ A \ B.$$  
$$\lambda x. F\ a =_\alpha \lambda y. \ F\ x\ y.$$  
$$\text{uva } N =_\alpha \text{uva } N.$$  
$$\text{fapp } K =_\alpha \text{fapp } K\ F\ x\ y.$$  
$$\text{pi }\ x\ :=\ \beta\ T\ [x]\ (T' \ x),\ \ x =_\alpha x \Rightarrow F\ x =_\alpha F\ x,\ T' =_\alpha T'.$$

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

For brevity, in this section, we consider the substitution for $\mathcal{T}_0$ and $\mathcal{H}_0$ identical. We defer to section 3.1 a more precise description pointing out their differences.

### Term unification: $\simeq_\alpha$ vs. $\simeq_\beta$. $\mathcal{H}_0$’s unification signature is:

$$\text{type } (\simeq_\alpha) \ \text{tm} \rightarrow \text{tm} \rightarrow_\alpha \text{ o}.$$  
$$\text{run } (\equiv_\alpha) \ \text{tm} \rightarrow \text{tm} \rightarrow_\alpha \text{ o}.$$  

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

### 2.3 The problem: logic-program simulation

We represent a logic program run in $\mathcal{T}_0$ as a sequence of steps of length $N$. At each step, we unify two terms $p, q$ taken from the list of all unification problems $P$. 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.

$$\text{frun } (\equiv_\alpha) \ \text{N} \Rightarrow \rho_N \equiv_\alpha \rho_0 \rightarrow \rho'$$

In order to simulate a $\mathcal{T}_0$ logic program in $\mathcal{H}_0$, we compile each $\mathcal{T}_0$ term $s$ in $\mathcal{P}$ to a $\mathcal{H}_0$ term $t$. We write this translation as $(s) \mapsto (\mathfrak{t}, \mathfrak{m}, \mathfrak{l})$. The implementation of the compiler is detailed in sections 3, and here we just point out that it additionally produces a variable mapping $\mathfrak{m}$ and a list of links $\mathfrak{l}$. The variable map connects unification variables in $\mathcal{H}_0$ to variables in $\mathcal{T}_0$ and is used to "decode" the assignment, $(\mathfrak{s}, \mathfrak{m}, \mathfrak{l})^{-1} \rightarrow \rho$. Links are an accessory piece of information whose description is deferred to section 2.4.

We write $\mathcal{T}_p = \{ \mathfrak{t}_1, \mathfrak{t}_2 \}$ and $s \in \mathcal{P} \Rightarrow \exists \mathfrak{t}_1, \mathfrak{t}_2 \in \mathcal{P}$.

We simulate each run in $\mathcal{T}_0$ with a run in $\mathcal{H}_0$ as follows:

$$\text{hstep } (\equiv_\alpha) \ \text{tm} \rightarrow \text{tm} \rightarrow_\alpha \text{ o}.$$  
$$\text{hstep } (\equiv_\alpha) \ \text{tm} \rightarrow \text{tm} \rightarrow_\alpha \text{ o}.$$  

By analogy with $\mathcal{P}$, we write $\mathcal{T}_p$ and $\mathcal{T}_p$ for the two $\mathcal{H}_0$ terms being unified at step $p$, and we write $\mathcal{T}_p$ for the set $\{ \mathfrak{t}_1, \mathfrak{t}_2 \}$. hstep is made of two sub-steps: a call to the meta language unification and a check for progress on the set of links, that intuitively will compensate for the weaker equational theory honored by $\simeq_\alpha$. hrun compiles all terms in $\mathcal{P}$, then executes each step, and finally decompiles the solution. We claim:
Proposition 2.1 (Simulation). ∀P, ∀N, if P ⊆ L
frun(P, N) ⇐⇒ ρN ⇐⇒ hrun(P, N) ⇐⇒ ρN
That is, the two executions give the same result if all terms in P are
in the pattern fragment. Moreover:

Proposition 2.2 (Simulation Fidelity). In the context of hrun,
if P ⊆ L we have that ∀ρ ∈ 1 . . . N,

fstep(P, p, ρP−1) ⧿ ρp ⧿ hstep(P, t, p, σp−1, p−1) ⧿ (σp−1, p−1)

In particular, this property guarantees that a failure in the F run
is matched by a failure in H 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 H by looking
at its execution trace in F.

We also claim that hrun handles terms outside L in the following
sense:

Proposition 2.3 (Fidelity recovery). In the context of hrun, if
ρP−1P ∈ L (even if ρP /∈ L) then

fstep(P, p, ρP−1) ⧿ ρp ⧿ hstep(P, t, p, σp−1, p−1) ⧿ (σp−1, p−1)

In other words, if the two terms involved in a step re-enter L,
then hstep and fstep are again related, even if ρP /∈ L and hence
proposition 2.2 does not apply. Indeed, the main difference between
proposition 2.2 and proposition 2.3 is that the assumption of the
former is purely static, it can be checked upfront. When this assump-
tion 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 pro-
gramming implementations, including Elpi, the order in which
unification problems are tackled does matter. The simplest ex-
ample is the sequence F ≐ λx.a and F a ≐ a; the second problem is
not in L and has two unifiers, namely σ1 ≐ { F ≐ λx.a } and
σ2 ≐ { F ≐ λx.a }. The first problem picks σ2, making the second
problem re-enter L.

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

2.4 The solution (in a nutshell)
A term s is compiled to a term t where every “problematic’ sub term
p is replaced by a fresh unification variable h with an accessory link
that represents a suspended unification problem h ≐ p. As a result
≡h is “well behaved” on t, in the sense that it does not contradict
≡o as it would otherwise do on the “problematic” sub-terms.

We now define “problematic” and “well behaved” more formally.
We use the ⊗ symbol since it stands for “possibly” in modal logic
and all problematic terms are characterized by some “uncertainty”.

Definition 2.4 (⊗β). ⊗β is the set of terms of the form X x1 . . . xn
such that x1 . . . xn are distinct names (of bound variables).

An example of a ⊗β term is the application F x. This term is prob-
lematic since the application node of its syntax tree cannot be used
to justify a unification failure, i.e. by properly instantiating F the
term head constructor may become a λ, or a constant, or remain
an application.

Definition 2.5 (⊙η). ⊙η is the set of terms s such that ∃ρ, ρs is
an eta expansion.

An example of a term s in ⊙η is λx.λy.F y x since the substitution
ρ = { F /⇒ λα.λb.f b a } makes ρs = λx.λy.f x y, which is the
eta long form of f. This term is problematic since its leading λ
abstraction cannot justify a unification failure against a constant f.

Definition 2.6 (⊙L). ⊙L is the set of terms of the form X t1 . . . tn
such that t1 . . . tn are not distinct names.

These terms are problematic for the very same reason terms in ⊗β
are, but they cannot be handled directly by the unification of the
meta language, which is only required to handle terms in L. Still,
there exists a substitution ρ such that ρs ∈ L.

We write P(t) the set of sub-terms of t, and we write P(X) =
∪t∈X P(t) when X is a set of terms.

Definition 2.7 (Well behaved set). Given a set of terms X ⊆ H0,

W(X) ⇐⇒ ∀t ∈ P(X), t /∈ (⊙β ∪ ⊙η ∪ ⊙L)

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

Proposition 2.8 (W-enforcing). Given two terms s1 and s2, if
∃ρ, ρs1 ≡o ρs2, then

(s1) ⧿ (t1, m1, l1) for i ∈ {1, 2} ⇒ t1 ⧿ t2 ⧿ σ

In other words the compiler outputs terms in W, even if its input
is not. Note that the property holds for any substitution, ρ could be
given by an oracle and/or not necessarily be a most general one: in
W ≐ 1 simply does not contradict ≡o.

Proposition 2.9 (W-preservation). ∀X, ∀P, ∀σ, ∀σ′

W(σT) ∧ σTρ1 ≐ σTρ2, σ′ ⇒ σ′ ⇒ W(σ′T)

W(σT) ∧ progress(L, σ) ⇒ (σ, σ′) ⇒ W(σ′T)

Proposition 2.9 is key to proving propositions 2.1 and 2.2. Inform-
ally, it says that the problematic terms moved on the side by the
compiler are not reintroduced by hstep, hence ≐h can continue
to operate properly. In sections 3, 5 and 7 we describe how the
compiler recognizes terms in ⊗β, ⊙η and ⊙L and how progress
takes care of them preserving W and ensuring propositions 2.1
to 2.3.

3 BASIC COMPILATION AND SIMULATION

3.1 Memory map (M) and substitution (ρ and σ)
Unification variables are identified by a (unique) memory address.
The memory and its associated operations are described below:

type abbrev (mem A) (list (option A)).
type set? addr → mem A → A → o.
type unset? addr → mem A → o.
type assign addr → mem A → A → mem A → o.
type new mem A → addr → mem A → o.
If a memory cell is none, then the corresponding unification variable is not set. assign sets an unset cell to the given value, while new finds the first unused address and sets it to none.

Since each $H_a$ 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.

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

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

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

When a single $fvariable$ occurs multiple times with different numbers of arguments, the compiler generates multiple mappings for it, on a first approximation, and then 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 substitution to a $H_a$ 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.

**3.2 Links ($L$)**

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

**Invariant 2 (Link left hand side).** The left-hand side of a suspended link is a variable.

New links are suspended by construction. If the left-hand side is assigned during a step, then the link is considered for progress and possibly eliminated. This is discussed in section 5 and section 7.

When detailing examples, we represent links as equations between two terms under a context. The equality sign is subscripted under binders. To accommodate this situation, the compiler wraps baselink using the inctx container (see $\cdot \cdot \cdot$ also used for subst).

**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 $H_a$ to $\mathcal{F}_a$ the compiler builds a “memory map” connecting the kind of variables using routine (malloc).

The signature of the comp predicate below allows for the generation of links (suspended unification problems), which play no role in this section but play a major role in sections 4, 5 and 7. With respect to section 2, the signature also allows for updates to the substitution.
The code above uses that possibility in order to allocate space for the variables, i.e. it sets their memory address to none (a details not worth mentioning in the previous sections).

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

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

### 3.4 Execution

A step in \( \mathcal{H}_0 \) consists 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.

### 3.5 Substitution decompilation

Decompling the substitution involves three steps.

First and foremost, problematic terms stored in \( L \) have to be moved back into the game: a suspended link must be turned into a valid assignment. This operation is possible thanks to invariant 2 (Link Left Hand Side), which ensures that no link causes an occur-check (3.4.2) and the fact that \( L \) is duplicate-free (??).

The second step involves allocating new variables in the memory of \( T_0 \). This technicality is required because some higher-order unifications may require pruning a variable. For example, \( F x y = F x z \) requires allocating a variable \( G \) in order to express the assignment \( F ab \mapsto Ga \).

The final step is to decomplie each assignment. Decompling a term is straightforward since \( M \) is a bijection. The only complex part concerns the abs node. In our simple setting, the \( \text{flatten} \) node carries no additional information (other than the function body), so each abs node can be trivially converted to a \( \text{flatten} \) one. However, in the case of CIC, where lambdas carry the type of the bound variable, one must store this information somewhere. Note that this information is similar to the arity of variables; that in CIC, unification variables have a (function) type, and this type can be used to annotate the lambdas needed to express their assignment.

**Lemma 3.1 (Compilation round trip).** If \( \text{compile} \ S T \searrow M \searrow L \) then \( \text{decompile} \ M T S \)

### 3.6 Definition of \( \equiv_0 \) and its properties

We already have all the pieces to show the code of \( \equiv_\lambda \).

### 3.4.2 Occur check

Since compilation moves problematic terms out of the sight of \( \equiv_\lambda \), that procedure can only perform a partial occur check. For example, the unification problem \( X \equiv_\lambda F \) cannot generate a cyclic substitution alone, but should be disallowed if a \( L \) contains a link like \( \lambda x. X \equiv_\eta_{\eta} \lambda z. X_2 \): we don’t know yet if \( Y \) will feature a lambda in head position, but we surely know it contains \( X \), hence \( F \) and that fails the occur check. The procedure \( \text{occur-check-links} \) is in charge of performing this check that is needed in order to guarantee proposition 2.2 (Simulation Fidelity).

So far the compiler is very basic. It does not really enforce that the terms passed to \( \text{hstep} \) are in \( W \), 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 \( \equiv_0 \) is a good "first-order" unification algorithm if the input already happens to be in \( W \). Later, when the compiler will enforce proposition 2.8 the proof will be adjusted to cover for the new cases.

**Lemma 3.2 (Properties of \( \equiv_0 \)).** The following properties hold for \( \equiv_0 \):

\[
\begin{align*}
\mathcal{W}(t_1 \equiv_\lambda t_2) & \Rightarrow t_1 \equiv_\lambda t_2 \Rightarrow \rho \Rightarrow \rho t_1 \equiv_\lambda \rho t_2 \quad (3) \\
\mathcal{W}(t_1 \equiv_\lambda t_2) & \Rightarrow \rho t_1 \equiv_\lambda \rho t_2 \Rightarrow \exists \rho', t_1 \equiv_\lambda t_2 \Rightarrow \rho' \wedge \rho' \subseteq \rho \\ & \Rightarrow \rho' \subseteq \rho 
\end{align*}
\]
Proof sketch. In this setting, $\sigma_\lambda$ is as strong as $\approx_o$ on ground terms. What we have to show is that whenever two different $\gamma_\tau$ terms can be made equal by a substitution $\rho$ (plus the $\beta_i$ and $\beta_r$ if needed), we can produce this $\rho$ by finding a $\sigma$ via $\sigma_\lambda$ on the corresponding $\mathcal{H}_\tau$ terms and by decompiling it. If we look at the syntax of $\gamma_\tau$, terms the only interesting case is $fuva \ x \approx_o s$. In this case, after compilation, we have $Y \sigma_\lambda t$ that succeeds with $\sigma = \{ Y \rightsquigarrow t \}$ and $t$ is decompiled to $\rho = \{ Y \rightsquigarrow s \}$ by lemma 3.1. \qed

Theorem 3.3 (Fidelity in $\mathcal{W}$). Proposition 2.1 (Simulation) and proposition 2.2 (Simulation Fidelity) hold if $\mathcal{W}(\mathcal{P})$.

Proof sketch. Trivial since progress is a no-op and fstep and hstep are the same, and by lemma 3.2 $\sigma_\lambda$ is equivalent to $\approx_o$. \qed

3.7 Notational conventions

In the following sections, we adopt this notation discuss the compiler’s output $\mathcal{P}$ represents the input set of problems that compiled to $\mathcal{T}$ with memory mapping $\mathcal{M}$ and $\mathcal{L}$. For example:

$$\begin{align*}
\mathcal{P} &= \{ p_1 \approx_o p_2 \quad p_3 \approx_o p_4 \} \\
\mathcal{T} &= \{ t_1 \approx_o t_2 \quad t_3 \approx_o t_4 \} \\
\mathcal{M} &= \{ X_1 \mapsto A_1^T \quad X_2 \mapsto A_2^T \} \\
\mathcal{L} &= \{ T \vdash a \approx_o b \}
\end{align*}$$

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, $T_2$ corresponds to the $\mathcal{H}_\sigma$ problem $t_3 \approx_o t_4$.

4 HANDLING OF $\Diamond \beta$

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

$$\begin{align*}
\mathcal{P} &= \{ \lambda x. (f \ (x \ x)) \approx_o \lambda x. (f \ x \ a) \} \\
\mathcal{T} &= \{ \lambda x. (f \ (A \ x)) \approx_o \lambda x. (f \ x \ a) \} \\
\mathcal{M} &= \{ X \mapsto A^B \}
\end{align*}$$

The unification problem $T_1$ fails while trying to unify $A \ x$ and $\ x$, which is equivalent to app [uva A [T]], versus $\ x$. In order to exploit the higher-order unification algorithm of the meta language, we need to compile the $\gamma_\tau$ term $x$ into the $\mathcal{H}_\sigma$ term $A \ x$, which is uva A [X].

4.1 Compilation and decompilation

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

$$\begin{align*}
\text{comp} (\text{fapp} [\text{fuva} \ A[Ag]]) &\quad (\text{uva} \ B \ Ag1) \ M1 \ M2 \ L \ L \ S1 \ S2 := \\
\text{pattern-fragment} \ Ag_1, \ !, \\
\text{fold6} &\quad \text{comp} \ Ag \ Ag1 \ M1 \ M1 \ L \ L \ S1 \ S1, \\
\text{len} \ Ag \ \text{Arity}, \\
\text{m-alloc} &\quad (fv \ A) \ (hv \ B \ (\text{arity} \ \text{Arity})) \ M1 \ M2 \ S1 \ S2.
\end{align*}$$

Note that compiling $\ Ag$ cannot create new mappings nor links, since $\ Ag$ is made of bound variables, and the hypothetical rule ($H_2$) loaded by comp-$\text{link}$ grants this property.

Decompilation. Since no link is created by the compilation of $\Diamond \beta$ terms, no modification to the commit-link is needed.

Proof. Similarly to decomposition, since no link is produced, no modification to the progress predicate is needed.

Definition 4.1 ($\mathcal{W}(\mathcal{P})$). $\mathcal{W}(\mathcal{P}) \ (X) \ (t) \not\in (\mathcal{\Diamond} \eta \cup \mathcal{\Diamond} \mathcal{L})$

Lemma 4.2 (Properties of $\approx_o$). The following properties hold for $\approx_o$ where

$$\begin{align*}
\mathcal{W}(\mathcal{P}) &\quad (t_1, t_2) \Rightarrow t_1 \approx_o t_2 \Rightarrow \rho \Rightarrow p_1 \approx_o p_2 \tag{5} \\
\mathcal{W}(\mathcal{P}) &\quad (t_1, t_2) \Rightarrow t_1 \approx_o t_2 \Rightarrow \exists \rho', t_1 \approx_o t_2 \Rightarrow \rho' \land \rho' \subseteq \rho \tag{6}
\end{align*}$$

Proof sketch. If we look at the $\gamma_\tau$ terms, there is one more interesting case, namely $fapp [fuva \ x] \approx_o s$ when $\ x$ are distinct names compiled to $\ x$. In this case the $\mathcal{H}_\tau$ problem is $Y \ x \approx_o t$ that succeeds with $\sigma = \{ Y \rightsquigarrow t \}$, which in turn is decompiled to $\rho = \{ Y \Rightarrow \lambda y.s \ \bar{w} \}$.

Theorem 4.3 ($\mathcal{W}(\mathcal{P})$-enforcement). Given two terms $s_1$ and $s_2$ in $\Diamond \beta$, if $\exists \rho, p_1 \approx_o p_2$, then

$$\{ s_1 \} \Rightarrow (t_1, m, l_1) \Rightarrow \{ i \in \{ 1, 2 \} \Rightarrow t_1 \approx_o t_2 \Rightarrow \sigma \}$$

Proof sketch. Thanks to lemma 4.3 $\sigma_\lambda$ is as powerful as $\approx_o$ in $\Diamond \beta$, as well as in $\mathcal{W}(\mathcal{P})$.

5 HANDLING OF $\Diamond \eta$

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, $\ i$ is the $\eta$-contraction of $\lambda x. t \ x$. The equational theory of $\gamma_\tau$ identifies these terms, but the current compilation scheme does not, as shown by the following example:

$$\begin{align*}
\mathcal{P} &= \{ \lambda x. X \ x \approx_o f \} \\
\mathcal{T} &= \{ \lambda x. A \ x \approx_o f \} \\
\mathcal{M} &= \{ X \mapsto A^1 \}
\end{align*}$$

While $\lambda x. X \ x \approx_o f$ does admit the solution $\rho = \{ X \mapsto f \}$, the corresponding problem in $\mathcal{T}$ does not: $\lambda m \ \lambda x. X \ [x]$ and $\lambda m \ \lambda x. X$ start with different, rigid, term constructors hence $\approx_o$ fails.

In order to guarantee proposition 2.1, 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 $\mathcal{T}$ to $\mathcal{L}$ (section 5.2). The compilation of the problem $\mathcal{P}$ above is refined to:

$$\begin{align*}
\mathcal{P} &= \{ \lambda x. X \ x \approx_o f \} \\
\mathcal{T} &= \{ A \approx_o f \} \\
\mathcal{M} &= \{ X \mapsto B^1 \} \\
\mathcal{L} &= \{ T \vdash a \approx_o \lambda x.B \}
\end{align*}$$

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:

Invariant 3 ($\eta$-link rhs). The rhs of any $\eta$-link has the shape $\lambda x. t$ and $t$ is not a lambda.
5.1 Detection of $\Diamond \eta$

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

\[
\begin{align*}
\lambda x. f \, (A \, x) & \in \Diamond \eta \quad \rho = \{ A \mapsto \lambda x. a \} \\
\lambda x. f \, (A \, x) \, x & \in \Diamond \eta \quad \rho = \{ A \mapsto \lambda x. a \} \\
\lambda x. f \, (A \, x) & \notin \Diamond \eta \\
\lambda x. f \, (A \, x) \, (B \, y \, x) & \in \Diamond \eta \quad \rho = \{ A \mapsto \lambda x. x, B \mapsto \lambda y. \lambda x. y \}
\end{align*}
\]

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 $\eta$-expansion or not. The third example shows that when a variable occurs outside the scope of a unification variable, it cannot be erased and can hence prevent a term from being an $\eta$-expansion. The last example shows the recursive nature of the check we need to implement. The term starts with a spine of two lambdas, hence the whole term is in $\Diamond \eta$ iff the inner term $\lambda y. f \, (A \, x) \, (B \, y \, x)$ is in $\Diamond \eta$ itself. If it is, it could $\eta$-contract to $f \, (A \, x)$ making $\lambda x. f \, (A \, x)$ a potential $\eta$-expansion.

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

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

**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 $x_1$ (e.g. $\lambda x_1 \ldots x_n. x \, x_1 \ldots x_n =_\approx x$);
3. $t$ is a unification variable with scope $W$, and for any $v \in \{ x, x_1, \ldots, x_n \}$, there exists a $w_i \in W$, such that $w_i$ may-contract-to $v$ (if $n = 0$ this is equivalent to $x \in W$).

**Proof sketch.** Since our terms are in $\beta$-normal form there is only one rule that can play a role (namely $\eta_i$), 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 $x_1 \ldots x_n$, it can either be $x$ applied to terms that can $\eta$-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 $\forall \rho, x \in \mathcal{P}(\rho t)$

\[
\begin{align*}
&\text{In other words, } x \text{ occurs-rigidly in } t \text{ if it occurs in } t \text{ outside of the scope of a unification variable } X; \text{ otherwise, an instantiation of } X \text{ can make } x \text{ disappears from } t. \text{ Moreover, note that } \eta \text{-contracting } t \text{ cannot make } x \text{ disappear, since } x \text{ is not a locally bound variable inside } t.
\end{align*}
\]

We can now describe the implementation of $\Diamond \eta$ 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 $x_j$ there exists a $w_j \in W$ such that $w_j$ may-contract-to $x_j$.

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

**Proof sketch.** Follows from definition 5.3 and lemma 5.2 $\Box$

### 5.2 Compilation and decompilation

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

\[
\begin{align*}
\text{comp (flam F) \ (uva A \ Scope) M1 M2 L1 L3 S1 S3 \ :-} \\
\text{maybe-eta (flam F) \ [1],} \\
\text{alloc S1 A S2,} \\
\text{comp-lam F F1 M1 M2 L1 L2 S2 S3,} \\
\text{get-scope (lam F1) Scope,} \\
\text{L3 = \{val (link-eta (uva A Scope) (lam F1)) \ | \ L2\}.}
\end{align*}
\]

Whenever $\text{flam F}$ is detected to be in $\Diamond \eta$ it is compiled to $\text{lam F1}$ and replaced by the fresh variable $A$. This variable sees all the names free in $\text{lam F1}$. The variable is connected to $\text{lam F1}$ via a $\eta$-link. Invariant invariant 2 (LINK LEFT HAND SIDE) holds for this link, moreover:

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

**Proof sketch.** By contradiction, suppose that the rule above is applied and that the rhs of the link is $\lambda x. \lambda y. t$, where $x$ and $y$ occur in $t$. If $\text{maybe-eta } \lambda y. t$ holds then the recursive call to comp (made by $\text{comp-lam}$) must have put a fresh variable in its place, so this case is impossible. Otherwise, if $\text{maybe-eta } \lambda y. t$ does not hold, then also $\text{maybe-eta } \lambda x. \lambda y. t$ does not hold either, contradicting the assumption that the rule was applied. $\Box$

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

### 5.3 Progress

$\eta$-links are meant to delay the unification of “problematic” terms until we know for sure if the term has to be $\eta$-contracted or not.
In both cases, we have here?

Definition 5.7 (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 = f a1 ... an we unify the η-expansion of X with T, that is we run λx. X x ≡λ T
2) if X = λx.t we run X ≡λ T.

Definition 5.8 (η-progress-rhs). A link Γ ⊢ X ≡η T is removed from L when either 1) maybe-eta T does not hold (anymore) or 2) by η-contracting T to T′ where T′ does not start with the 1am constructor. In the first case, X is unified with T, and in the second one, X is unified with T′ (under the context Γ).

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

Definition 5.9 (η-progress-duplicate). A link Γ ⊢ X1 ≡η T is removed from L when another link Γ ⊢ X2 ≡η T′ is in L. By invariant 1 the length of s and the s is the same; hence we can move the term T′ from A to Γ by renaming its bound variables, i.e. T″ = T′[s/ʃ]. We then run T ≡λ T′ (under the context Γ).

Lemma 5.10. Let λx.t the rhs of a η-link, then W(t).

Proof sketch. By construction, every "problematic" term in Fη is replaced with a variable in the corresponding Hη term. Therefore, t = W.

Lemma 5.11. Given a η-link l, the unification done by η-progress-lhs is between terms in W.

Proof sketch. Let σ be the substitution, which is W(σ) (by proposition 2.9). Lhs ∈ σ, therefore W(lhs). By η-progress-lhs, if 1) the lhs is a name, a constant or an application, then, λx.lhs x is unified with the rhs. By invariant 3 and lemma 5.10, rhs = λx.t and W(t). Otherwise, 2) the lhs has 1am as functor. In both cases, uniformation is performed between terms in W.

Lemma 5.12. Given a η-link l, the unification done by η-progress-rhs is between terms in W.

Proof sketch. Lhs is a variable, and, by definition 5.8, rhs is either no more a Ω, i.e. rhs is not a η-expansion and, so, W(rhs), otherwise, rhs can reduce to a term which cannot be a η-expansion, and, so, W(rhs). In both cases, the unification between rhs and lhs is done between terms that are in W.

Lemma 5.13. Given a η-link l, the unification done by η-progress-duplicate is between terms in W.

Proof. We unify the rhs of two η-link. Both have the shape λx.t and by lemma 5.10, W(t).

Lemma 5.14. The introduction of η-link guarantees proposition 2.8 (W-enforcing)

Proof sketch. By definition 5.4.

Lemma 5.15. The progress of η-link guarantees proposition 2.9 (W-preservation)

Proof sketch. By lemmas 5.11 to 5.13, every unification performed by the activation of a η-link is done between terms in W, therefore, the substitution remains W.

Lemma 5.16. progress terminates.

Proof sketch. Rules definitions 5.7 and 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 η-link, contraction, η-expansion, relocation (a recursive copy of a finite term).

Theorem 5.17 (Fidelity in Ωη). Given a list of unification problems P, if ∀i, t ∈ P(P) ⇒ t ∈ Q ∧ L, and if the memory map is bijective, then the introduction of η-link guarantees proposition 2.2 (Simulation Fidelity). 1

Proof sketch. η-progress-lhs and η-progress-duplicate activate a η-link when, in the original unification problem, a Ω term is unified with respectively a well-behaved term or another Ω term. In both cases, the links trigger a unification which succeeds if the same unification in Fη succeeds, guaranteeing proposition 2.2. η-progress-rhs never fails, in fact, this progression refines a variable to a rigid term and plays no role in proposition 2.2.

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 t = f = λx.Bx and the lhs is a constant. This rule runs λx.f x ≡λ λx.Bx, resulting in σ = { A ⇒ f ; Bx ⇒ f }.

By the definition of η-link, a variable to a rigid term and plays no role in proposition 2.2.

Example of η-progress-duplicate. A very basic example of η-link deduplication, is given below:

P = { λx.(X y) ≡o λx.(Y y) } T = { A ≡λ C } M = { X ≫ B1 Y ≫ D1 } L = { t A ≡η λx.Bx t C ≡η λx.Dx } The result of A ≡η C is that the two η-link share the same lhs. By unifying the two rhs we get σ = { A ⇒ C, B ⇒ D }.

In turn, given the map M, this second assignment is compiled down to σ = { X ⇒ Y } as expected.

We give an example for rule η-progress-rhs in the next section.

6 MAKING M A BIJECTION

In section 3.1, we introduced the definition of memory map M. This map allows to decompile the Hη substitution to the object language.

Given the higher order nature of CIC the same unification variable X is used multiple times with different arities.

P = { λx.λy.(X y x) ≡o λx.λy.x λx.(f (X x) x) ≡o Y } T = { A ≡λ λx.λy.x D ≡λ F } M = { X ≫ E1 Y ≫ F0 X ≫ C1 } L = { t D ≡η λx.(f X x) t A ≡η λx.Bx } In the unification problems P above, we see that X is used with arity 2 in P1, and with arity 1 in P2. In order to preserve invariant 1 (Unification-variable arity) the compiler did generate two entries in M for X, and in turn this breaks proposition 2.2 (Simulation Fidelity) since two incompatible assignments to the variables X

1This premise is fulfilled in section 6
maps to, namely $E$ and $C$, would not be detected. We remedy by post-processing the compiler’s output by adding additional $\eta$-link to ensure the followgeom property:

**Proposition 6.1 ($\eta$ is a bijection).** After compilation, for each $T_0$ variable $X$ in $P$ and for each $T_0$ variable $A$ in $T$ there is exactly one entry $X \mapsto A^0$ (for any arity $n$).

Note that the execution in $T_0$ may require allocating new variables as explained in section 3.5. The property above does not concern these.

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 $0 \leq i < d$,

$$x_0 \ldots x_{m+i} \mapsto b_i^{i+1} \mapsto \gamma \lambda x_{m+i+1} \ldots x_{m+i+2}$$

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 $m_2 : X \mapsto C^n$ 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 $P$, such that $\forall t \in P \exists W(t) \forall t \in \eta$, if $P$ contains the same $T_0$ variable used at different arities, then map-deduplication guarantees proposition 2.2 (Simulation Fidelity)

**Proof sketch.** By the definition of map-deduplication, any two occurrences of the same $T_0$ variables $X_1, X_2$ with different arities are compiled to two $A^n$ and $C^m$ related by a chain of $\eta$-link. We show that whenever one of the two is instantiated with a rigid term the the chain becomes shorter and hence, eventually, the other variable is instantiated too. Without loss of generality we assume $m < n$ and $A$ is instantiated. The corresponding $\eta$-link is triggered by $\eta$-progress-lhs instantiated the next variable in the chain of links.

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

If we look back the example give at the beginning of this section, we can deduplicate $X \mapsto E^1$, $X \mapsto C^2$ by removing the first mapping and adding the auxiliary $\eta$-link: $x \mapsto E_\gamma = \eta \lambda y. C_{xy}$. After deduplication the compiler output is as follows:

$$P = \{ \lambda x. \lambda y. (x y) \mapsto_\sigma \lambda x. \lambda y. x \mapsto_\sigma \lambda x. (f (x x) x) \mapsto_\sigma Y \}$$

$$T = \{ A \mapsto_\delta \lambda x. \lambda y. x \mapsto_\delta \lambda x. (f (E x) x) \mapsto_\delta F \}$$

$$M = \{ Y \mapsto F \mapsto X \mapsto C^2 \}$$

$$L = \{ A \mapsto_\eta \lambda y. C_{xy} \mapsto_\eta \lambda y. (f (E x) x) \mapsto_\eta L_1 \mapsto_\eta \lambda x. (f (E x) x) \mapsto_\eta A \mapsto_\eta \lambda x. B x \mapsto_\eta B x \mapsto_\eta \lambda y. C_{xy} \}$$

The resolution of $L_1$ assigns $\lambda x. y x$ to $A$ that in turn triggers $L_3$ and then $L_4$ by $\eta$-progress-lhs, one per $\eta$ in the solution of $A$. The unification variable $C_{xy}$ is therefore unified with $x$ (the second variable of its scope). As a result $L_1$ becomes $x \mapsto E_x = \eta \lambda y. y$, and the rhs is no more in $\square$ so $\eta$-progress-rhs fires and $E_x$ is unified with $\lambda x. y$, resulting in $\sigma = \{ A \mapsto \lambda x. \lambda y. x ; B x \mapsto \lambda x. x ; C_{xy} \mapsto x ; E_x \mapsto \lambda y. y \}$.

The remaining step $T_2$ identifies $D$ with $F$, hence the resulting $T_0$ substitution is $\rho = \{ x \mapsto \lambda x. \lambda y. y ; Y \mapsto \lambda x. (f (\lambda y. y)) x \}$.

### 7 Handling of $\Omega \setminus \mathcal{L}$

Even if we do not require $\approx_1$ to handle problems outside of $\mathcal{L}$ it is worth handling terms in $\Omega \setminus \mathcal{L}$ since, in practice, these terms often re-enter $\mathcal{L}$ at runtime, as observed in [14] for example, or to accommodate for the introduction of heuristics.

In the following example problem $P_2$ is $X a \equiv_0 a$ that admits two different substitutions: $p_1 = \{ X \mapsto \lambda x x \}$ and $p_2 = \{ X \mapsto \lambda x. a \}$. The unification algorithm alone cannot close, 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:

$$P = \{ X \mapsto_\delta \lambda x. a \}$$

$$T = \{ A \mapsto_\delta \lambda x. a \}$$

$$M = \{ X \mapsto A^0 \}$$

We want to support this scenario but we have to improve a little our compiler and progress routines. In particular $T_1$ generates $\sigma = \{ A \mapsto \lambda x. a \}$ making $\sigma T_2$ equal to $(\lambda x. a) a \equiv_1 a$ that $\equiv_1$ cannot solve since it lacks rules $\beta_1$ and $\beta_2$.

To address this problem the compiler must recognize and replace $\Omega \setminus \mathcal{L}$ terms 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 ($\eta$-link rhs).** The rhs of any $\mathcal{L}$-link has the shape $X_1 \ldots X_m$ where $X$ is a unification variable in $\mathcal{L}$ (with scope $s_1$) 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 $\text{app} [uva \ X \ S \mid L]$, where $S = s_1 \ldots s_n$ and $L = t_1 \ldots t_m$.

### 7.1 Compilation and decompilation

For the detection of $\Omega \setminus \mathcal{L}$ it is sufficient to identify applications with a flexible head and arguments that are not in distinct names. The following rule for $\mathcal{L}$ compilation is inserted just before rule ($\epsilon_0$).

$$\text{comp (appHeight (uva A)) (uva B Scope)} M_1 M_3 L_3 S_3 S_4 : !$$

$$\text{pattern-prefix Ag Pf Extra}$$

$$\text{len Pf Arity}$$

$$\text{alloc S_1 B S_2}$$

$$\text{m-alloc (fv K) (hv C (arity Arity)) M_1 M_2 S_3 S_5}$$

$$\text{fold6 comp Pf Pf M_2 M_2 L_1 L_1 S_3 S_5}$$

$$\text{fold6 comp Extra Expr M_2 M_3 L_1 L_2 S_3 S_5}$$

$$\text{Beta = app [uva C Pf] [Extra]}$$

$$\text{get-scoping Beta Scope}$$

$$\text{L_3 = [val (linkfail (uva B Scope) Beta) \mid L_2]}$$

The list $Ag$ is split into the two parts: $Pf$ is in $\mathcal{L}$ and can be empty; $Extra$ cannot be empty and is such that append $Pf Extra Ag$. The rhs of the $\mathcal{L}$-link is the application of a fresh variable $C$ having in scope all names in $Pf_1$ (the compilation of $Pf$). The variable $B,$
Decomposition. All $L$-1link should be solved before decompli-
ca tion. If any $L$-1link remains in $L$, decomposition fails.

7.2 Progress

Given a $L$-1link $l$ of the form $\Gamma + T = L X_{S_1,...,S_n} t_1 \ldots t_m$, we pro-
\provide the following activation rules:

**Definition 7.1 (L-progress-refine).** Let $\sigma$ be a substitution such
that $\sigma(t)$ is a fresh variable having in scope all the free variables occurring in $Pf_1$ and $Extra_1$ (the compilation of $Extra$). This construction enforces invariant 4.

Let $\Gamma + T \equiv L Y_{S_1,...,S_n} t_2 \ldots t_m$ (where $Y$ is a fresh variable of
arity $n + 1$) and link $\Gamma + X_{S_1,...,S_n} \equiv \eta \lambda x. Y_{S_1,...,S_n} x$ is added to $L$.

**Definition 7.2 (L-progress-rhs).** Link $l$ is removed from $L$ if
$X_{S_1,...,S_n}$ is instantiated to a term $t$ and $t t_1 \ldots t_m \beta$-reduces to a $\bar{t} \in L$.

**Definition 7.3 (L-progress-fail).** Progress fails whenever: there
exists another link $l' \in L$ with same lhs as $l$; or the lhs of $l$ become
rigid.

In order to accommodate for the implementation of heuristics we
shall relax these failure conditions in section 7.3.

**Lemma 7.4. progress terminates**

**Proof sketch.** Let $l$ be a $L$-1link in the store $L$. If $l$ is activated
by $L$-progress-rhs, then it disappears from $L$ and progress termi-
nates. Otherwise, the rhs of $l$ is made by a variable applied to $m$
arguments. At each activation of $L$-progress-refine, it is replaced by
a new $L$-1link $l^i$ having $m = 1$ arguments. At the $m^i$th iteration,
the $L$-1link $l^m$ has no more arguments and is removed from $L$.

Note that at the $m^i$th iteration, $m$ new $\eta$-link have been added
in $L$; however, by lemma 5.16, the algorithm terminates. Finally
$L$-progress-fail also guarantees termination since it makes progress
immediately fails.

**Theorem 7.5 (Fidelity with $L$-1link).** The introduction of $L$-1link
guarantees proposition 2.3 (Fidelity recovery) if $\equiv_\sigma$ fails outside $L$.

**Proof sketch.** Let $P_2$ be the first problem in $P$ not in $L$ and le $\sigma$
be the substitution obtained solving $S_1 \ldots S_1$. If $\sigma S_1$ is in $L$, then
each variable that the compiler used to replace a problem term is
assigned and hence by definitions 7.1 and 7.2 the associated $L$-1link
been solved and removed. Note that in this case all calls to $\equiv_\lambda$ are
between terms in $L$ and by theorem 5.17 fidelity is guaranteed.

If $\sigma T$ is still in $\Delta L$, then by definition 7.3 unification fails as the
corresponding unification in $P_2$ would.

**Example of L-progress-refine.** Consider the $L$-link below:

\[
P = \{ X \equiv_\lambda \lambda x.x \lambda x.(Y (X x)) \equiv_\eta f \}
\quad
T = \{ A \equiv_\lambda \lambda x.x \lambda \lambda x.f \}
\quad
M = \{ Y \equiv D^0 \lambda \lambda x.a \}
\quad
L = \{ \eta \equiv_\lambda \lambda x.E \quad \lambda \lambda x.C \}
\]

Initially the $L$-1link rhs is a variable $D$ applied to the $E_x$. The
first unification problem results in $\sigma = \{ A \equiv \lambda x.x \}$. In turn this
instantiation triggers $L_1$ by $\eta$-progress-rhs and $E_x$ is assigned to $x$. Under
this substitution the $L$-1link becomes $x + C_x = L (D x)$, and
by $L$-progress-refine it is replaced with the link: $E \equiv_\eta \lambda x.D_x; x$. Here
$C_x$ is unified with $D_x$. The second unification problem assigns $f$ to
$B$, that in turn activates the second $\eta$-1link ($f$ is assigned to $C$), and
then all the remaining links are solved. The final $H_4$ substitution is
$\sigma = \{ A \equiv \lambda x.x; B \equiv f; C_x \equiv (f x); D \equiv f; E_x \equiv x; F_x \equiv C_x \}$ and is decomposed into $\rho = \{ X \equiv \lambda x.x; Y \equiv f \}$.

**Example of L-progress-rhs.** We give a second look at the opening
example in section 7. The problem is compiled into:

\[
P = \{ \lambda x.x \equiv a \}
\quad
T = \{ A \equiv \lambda \lambda x.B \equiv \lambda \lambda x.a \}
\quad
M = \{ Y \equiv B^0 \lambda \lambda x.a \}
\quad
L = \{ \lambda \lambda x \equiv \lambda \lambda x.B \}
\]

The first unification problem is solved by the substitution $\sigma = \{ A \equiv \lambda x.B \}$. The $L$-1link becomes $C \equiv L (\lambda x.B) a$ whose rhs
\begin{itemize}
\item can be $\beta$-reduced to $B$, $B$ is in $L$ and is unified with $C$. The resolution of the second unification problem gives the final substitution $\sigma = \{ A \equiv \lambda x.B; B \equiv C \equiv \lambda \lambda x.a \}$ that is decomposed into $\rho = \{ X \equiv \lambda x.a; Y \equiv a \}$.
\end{itemize}

7.3 Relaxing definition 7.3 (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 [10]. Some implementations of
$\lambda$Prolog [15] such as Teyjus [14] delay the resolution of $\Delta L$ 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 [21], 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 $P_3$ is in $\Delta L$.

\[
P = \{ (X a) \equiv a \lambda x.Y \}
\quad
T = \{ A \equiv \lambda \lambda x.a \lambda \lambda x.C \}
\quad
M = \{ Y \equiv B^0 \lambda \lambda x.a \}
\quad
L = \{ \lambda \lambda x \equiv \lambda \lambda x.B \}
\]

If we want the object language unification to delay the first unifi-
cation problem (awaiting for $X$ to be instantiated), we can relax
definition 7.3. Instead of failing when the lhs of $L_3$ 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 2.2
(Simulation fidelity) per se, the occur-check-1links procedure be-
comes incomplete since invariant 2 (Link left hand side) is broken.

Also note that delaying unification outside $L$ can leave $L$-1link for
the decomposition phase. Therefore commit-1links should be
modified accordingly.

If instead we want $\equiv_\lambda$ to follow the second strategy and pick an
arbitrary solution we can modify progress by applying the desired
heuristic. For instance, in $X a b = Y b$, the last argument of the two
terms is the same and unification can succeed by assigning $X a$ to
$Y$. This heuristic can be implemented by adding the following rule:

\[
\begin{align*}
\Delta L &= \{ (X a) \equiv_{\lambda \lambda x.a} (Y b) \}
\end{align*}
\]
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 \( H_p \) as well.

The main difference is that we cannot implement hrun since it comes as part of the runtime of the programming language. In particular the runtime iterates \( \approx_h \), but hrstep also needs to check for progress. Luckily Elpi extends \([9]\) \( \lambda \text{Prolog} \) 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 2.2 (SIMULATION FIDELITY).

For brevity we only provide two pseudo-code snippets. The first one depicts how \( \eta \text{-link} \) are suspended or make progress.

```prolog
link-eta L R :- \text{var} L, \text{maybe-eta} R \text{HolesInR}, !,
   declare_constraint (link-eta L R) [L|HolesInR].

decide (link-eta L R :- not (\text{var} L), !, \text{eta-progress-lhs} L R).

\text{improve}
```

The second snippet illustrate the deduplication of \( \eta \text{-link} \). The syntax \( \text{seq} \rightarrow \text{goal} \rightarrow \text{goal} \rightarrow \text{goal} \) denotes a \( \lambda \text{Prolog} \) sequence, that is a goal \( P \) under a set \( \text{G} \) of hypothetical rules (introduced by \( \Rightarrow \)) and where the program context binds (via the \( \pi \) operator) elpi1N heigenvariables.

Sequences are presented to CHR with their higher order unification variables replaced by "frozen constants", that is \( A_{X,Y,Z} \) becomes \( \text{\texttt{uvar}} f_a \text{[x,y,z]} \) for a fresh constant \( f_a \). Frozen constants are "defrost" when they are part of a new goal (see \[9\], section 4.3).

```prolog
\text{constraint link-eta (}
rule (N1 \rightarrow G ?, link-eta (\text{\texttt{uvar}} X Lx1) T1) \text{ match}
/ (N2 \rightarrow G ?, link-eta (\text{\texttt{uvar}} X Lx2) T2) \text{ remove}
\mid (\text{\texttt{relocate}} Lx1 Lx2 T2 T2') \text{ condition}
\Leftrightarrow (N1 \rightarrow G ?, T1 \rightarrow T2'). \text{ % new goal}
\text{constraint}
```

The first directive matches a constraint whose lhs is a variable \( X \); the second mathes and removes a constrain on the same variable; the third is a test that accessorially 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
Different strategies can be used to unify terms of the object language. The first approach that comes to mind consist in implementing \( \approx_a \) as a regular routine in the ML, i.e. write rules as follows:

```prolog
decision \( X \rightarrow X \rightarrow (\text{all} A \ x) \rightarrow \text{app} [P, X]), \text{finite} A, \text{pi} \rightarrow \text{decision} [P, X]).
```

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

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

```prolog
decision \( \text{all} A \ x \rightarrow \text{finite} A, \text{pi} \rightarrow \text{decision} (P \ x)).
```

However, this encoding has two big limitations. First it is not always feasible to adapt it for CIC due to the fact that the type system of the ML is too limited to accommodate for that one the object language, e.g. CIC can typecheck variadic functions \([2]\).

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

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 ML in presence of non ground goals, but the authors do not aim at exploiting it to the degree we want.

Another work that is, surprisingly, only superficially related to ours is the type-class engine built in Isabelle’s meta language. In \([27]\) 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 provides a third option that addresses all the concerns mentioned earlier. It capitalizes on the benefit of not requiring to fully implement the unification algorithm of the object language. Instead, it employs the unification capabilities of the meta language, facilitated by the various links to manage "problematic" subterms. As a result of this choice our encoding takes advantage of indexing data structures and mode analysis for clause filtering. It worth mentioning that we only replace terms with variables only when it is strictly needed, leaving the rest of the term structure intact and hence indexable by the meta language logic programming engine. Moreover, the unification process we obtain is ready to take advantage of potential improvements to the programming engine, such as tabled search, and apply forms of static analysis for the meta language, such as determinacy, to the object language. Finally, our approach is flexible enough to accommodate different strategies and heuristics to handle terms outside the pattern fragment, and it is not tightly coupled with CIC.

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

Davide Fissore and Enrico Tassi


APPENDIX

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

Note that (a infix b) c d de-sugars to (infix a) b c d. Explain builtin name (can be implemented by loading name after each pi)

10 THE MEMORY

kind addr type.
type addr nat -> addr.
typeabbrev (mem A) (list (option A)).
set? (addr A) Mem Val :- get A Mem Val.

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

assign-aux (s N) (none :: L) Y (some Y :: L) :- assign-aux N L Y L.

assign (addr A) Mem1 Val Mem2 :- assign-aux A Mem1 Val Mem2.

type get nat -> mem A -> A -> o.
get z (some Y :: _) Y.
get (s N) (none :: L) X :- get N L X.

alloc-aux z [] [none] :- 1.
alloc-aux z L L.
alloc-aux (s N) [] [none | M] :- alloc-aux N [] M.
alloc-aux (s N) X :: L L X :: M :- alloc-aux N L M.

alloc (addr A as Ad) Mem1 Val Mem2 :- unset? Ad Mem1,
alloc-aux A Mem1 Mem2.

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

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

11 THE OBJECT LANGUAGE

kind fm type.
type fapp list fm -> fm.
type flam (fm -> fm) -> fm.
type fcon string -> fm.
type fuva addr -> fm.
typeabbrev fs subst (mem fm).

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

10) type fderref fs subst -> fm -> fm -> o.
fderref S T T2 :- fder S T1, napp T1 T2.

10) type (=o) fm -> fm -> o.
con X \=o fcon X.
fapp A \=o fapp B :- forall2 (=o) A B.
flam F \=o flam G :- pi x \=o x \=o F x \=o G x.
fuva N \=o fuva N.
flam F \=o T :- pi x \ beta T [x] (T & x) \=o F x \=o T x.
T \=o flam F :- pi x \ beta T [x] (T & x) \=o F x.
flam [flam X | L] \=o T :- beta (flam X) L R, R \=o T. (βT)
T \=o fapp [flam x | L] :- beta (flam X) L R, T \=o R. (βT)

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

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

10) type napp fm -> fm -> o.
napp (fcon C) (fcon C).
napp (fuva A) (fuva A).
napp (flam F) (flam G) :- pi x \ napp (F x) (G x).
napp (fapp [fapp L | L2] L2) T :- 1,
append L1 L2 L3, napp (fapp L3) T.
napp (fapp L) (fapp L1) :- map napp L L1.
napp N N :- name N.

10) type beta-normal fm -> fm -> o.
beta-normal (uvar _ _) _ :- halt "Passed uvar to beta-normal".
beta-normal A A :- name A.
beta-normal (fcon A) (fcon A).
beta-normal (fuva A) (fuva A).
beta-normal (flam A) (flam B) :- pi x \ beta-normal (A x) (B x).
beta-normal (fapp [flam B | L1] T2 :- 1,
beta (flam B) L T1, beta-normal T1 T2.
12 THE META-LANGUAGE

kind inctx type -> type.  
(type (\_\_\_)) tm -> tm -> subst -> subst -> o.

13 PRUNE

prune! M A N A S1 S2 :: !, bind (uva M A) A Ass,
assign N S1 Ass S2.
/* prune different arguments */
prune! N A1 N A2 S1 S3 :: !,
new S1 W S2, prune-same-variable W A1 A2 [] Ass,
assign N S2 Ass S3.
/* prune to the intersection of scopes */
prune! N A1 M A2 S1 S4 :: !,
new S1 W S2, prune-diff-variables W A1 A2 Ass S2,
assign N S2 Ass S3, assign M S3 Ass S4.

14 PRUNE-SAME-VARIABLE

type prune-same-variable addr -> list tm -> list tm ->
list tm -> assignment -> o.
prune-same-variable N [] [] ACC (val (uva N Args)) ::
rev ACC Args.
prune-same-variable N [X][X]S [X][Y]S ACC (abs F) ::
pi \ x \ prune-same-variable N X Y S [X][Y]S F X.
prune-same-variable N [...][X]S [...][Y]S ACC (abs F) ::
pi \ x \ prune-same-variable N X Y S [X][Y]S F X.

15 PERMUTATE

type permute list nat -> list tm -> list tm -> o.
permute [] [] [].
permute (P | PS) Args [T | TS] ::
nth P Args T,
permute PS Args TS.

16 BUILD-ASSIGN

type build-assign addr -> list tm -> bool ->
list nat -> assignment -> o.
build-assign N ArgsR [] Perm (val (uva N PermutedArgs)) ::
rev ArgsR Args, permute Perm Args PermutedArgs.
build-assign N Acc [tt][L] Perm (abs T) ::
pi \ x \ build-assign-n Acc [x][Acc] L Perm (T x).
build-assign N Acc [ff][L] Perm (abs T) ::
pi \ x \ build-assign-n Acc L Perm (T x).

17 KEEP

type keep list A -> A -> bool -> o.
keep L A tt :: mem L A, !.
keep _ _ ff.

18 PRUNE-DIFF-VARIABLES

type prune-diff-variables addr -> list tm -> list tm ->
assignment -> assignment -> o.
prune-diff-variables N Args1 Args2 Ass1 Ass2 ::
map (keep Args2) Args1 Bits1,
map (keep Args1) Args2 Bits2,
filter Args1 (mem Args2) ToKeep1,
filter Args2 (mem Args1) ToKeep2,
map (index ToKeep1) ToKeep1 IdPerm,
map (index ToKeep1) ToKeep2 Perm1,
build-assign-n N [] Bits1 IdPerm Ass1,
build-assign-n N [] Bits2 Perm2 Ass2.

19 PRUNE-ASSIGNMENT

type prune-assignment addr -> list tm ->
list tm -> list tm -> assignment -> o.
prune-assignment N Args1 Args2 Ass1 Ass2 ::
map (keep Args2) Args1 Bits1,
map (keep Args1) Args2 Bits2,
filter Args1 (mem Args2) ToKeep1,
filter Args2 (mem Args1) ToKeep2,
map (index ToKeep1) ToKeep1 IdPerm,
map (index ToKeep1) ToKeep2 Perm1,
built-assignment-n N [] Bits1 IdPerm Ass1,
built-assignment-n N [] Bits2 Perm2 Ass2.

20 BIND

type bind tm -> tm -> o.
bind (uva M A) A S S ::
bind (uva M A) A S S.
/* bind different arguments */
bind (uva M A) A S S.
new S1 W S2, prune-same-variable W A1 A2 [] Ass,
assign N S2 Ass S3.
/* prune to the intersection of scopes */
bind (uva M A) A S S.
new S1 W S2, prune-diff-variables W A1 A2 Ass S2,
assign N S2 Ass S3, assign M S3 Ass S4.

21 HTML-HEAD

*template html-head*
Higher-Order unification for free!  
Conference’17, July 2017, Washington, DC, USA

beta X L (app[X|L]) ::= name X.

type beta-aux tm -> tm -> o.

beta-aux (app [H|D|T|L]) R := !, beta HD TL R.

beta-aux A A.

/* occur check for N before crossing a functor */
type not_occ addr -> subst -> tm -> o.

not_occ N S (uva M Args) ::= set? M S F,
    move F Args T, not_occ N S T.

not_occ N S (uva M Args) ::= unset? M S, not (N = M),
    forall1 (not_occ_aux N S) Args.

not_occ _ _ (con _).

not_occ N S (app L) ::= not_occ_aux N S (app L).

/* Note: lam is a functor for the meta language! */
not_occ N S (lam L) ::= pi x \not_occ_aux N S (L x).

not_occ _ _ X ::= name X.

/* Finding N is ok */
not_occ N _ (uva _ _).

/* occur check for X after crossing a functor */
type not_occ_aux addr -> subst -> tm -> o.

not_occ_aux N S (uva M _) ::= unset? M S, not (N = M).

not_occ_aux N S (uva M Args) ::= set? M S F,
    move F Args T, not_occ_aux N S T.

not_occ_aux N S (app L) ::= forall1 (not_occ_aux N S) L.

not_occ_aux N S (lam L) ::= pi x \not_occ_aux N S (F x).

not_occ_aux _ _ _ X ::= name X.

/* Finding N is ok, hence no rule */
not_occ N _ _.

/* copy T T’ vails if T contains a free variable, i.e. it
performs scope checking for bind */
type copy tm -> tm -> o.

copy (con C) (con C).

copy (app L) (app L’) ::= map copy L L’.

copy (lam T) (lam T’) ::= pi x \copy x x \=> \copy (T x) (T’ x).

copy (uva A L) (uva A L’) ::= map copy L L’.

type bind tm -> list tm -> assignment -> o.

bind T [ ] (val T’) ::= copy T T’.

bind T [X | TL] (abs T’) ::= pi x \copy x x \=> bind T TL (T’ x).

type deref subst -> tm -> tm -> o.

(deref _ (con C) (con C).

deref S (app A) (app B) ::= map (deref S) A B.

deref S (lam F) (lam G) ::= pi x \deref S S x x \=> deref S (F x) (G x).

deref S (uva N L) R ::= set? N S A,
    move A L T, deref S T R.

deref S (uva N A) (uva N B) ::= unset? N S,
    map (deref S) A B.

type move assignment -> list tm -> tm -> o.

move (abs Bo) [H|L|R] ::= move (Bo H) L R.

move (val A) [ ] A.

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

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

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

13 THE COMPILER

kind arity type.

kind arity nat -> arity.

kind fvariable type.

type fv addr -> fvariable.

kind hvariable type.

type hv addr -> arity -> hvariable.

kind mapping type.

type (<>->) fvariable -> hvariable -> mapping.

typeabbrev mmap (list mapping).


typeabbrev scope (list tm).

typeabbrev inctx ho.inctx.

kind baselink type.

type link-eta tm -> tm -> baselink.

type link-llam tm -> tm -> baselink.

typeabbrev link (inctx baselink).

typeabbrev links (list link).

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

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


type get-lhs link -> tm -> o.

get-lhs (val (link-llam A _)) A.

get-lhs (val (link-eta A _)) A.

type get-rhs link -> tm -> o.

get-rhs (val (link-llam _ A)) A.

get-rhs (val (link-eta _ A)) A.


type occurs-rigidly fm -> fm -> o.

occurs-rigidly N N.

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

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

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


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

reducible-to _ N N ::= !.

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

reducible-to L N (flam B) ::= !,
    pi x \reducible-to [X | L] N (B x).

reducible-to L N (fapp [N|Args]) ::= last-n (len L) Args R,
    forall2 (reducible-to [] R) (rev L).


type maybe-eta fm -> list fm -> o.

maybe-eta (fapp [fuva _]) L ::= !,
    forall1 (x) exists (reducible-to [] x) Args L, !.
maybe-eta (flam B) L :- !, pi x \ maybe-eta (B x) [x \ L].

split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.

maybe-eta (fapp [T\Args]) L :- (name T; T = fcon _),
split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.

maybe-eta (fapp [uva A\[Ag]]) L :- (name T; T = fcon _),
split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.

maybe-eta (fapp [uva A\[Ag]]) L :- (name T; T = fcon _),
split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.

maybe-eta (fapp [uva A\[Ag]]) L :- (name T; T = fcon _),
split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.

maybe-eta (fapp [uva A\[Ag]]) L :- (name T; T = fcon _),
split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.

maybe-eta (fapp [uva A\[Ag]]) L :- (name T; T = fcon _),
split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.

maybe-eta (fapp [uva A\[Ag]]) L :- (name T; T = fcon _),
split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.

maybe-eta (fapp [uva A\[Ag]]) L :- (name T; T = fcon _),
split-last-n (len L) Args First Last,
none (x \ exists (y \ occurs-rigidly x \ y) First) L,
forall1 (reducible-to []) (rev L) Last.
14 The Progress Function

macro @one := s 2.

type contract-rigid list ho.tm -> ho.tm -> ho.tm -> o.
contract-rigid L (ho.lambda F T) :=
  pi x\ contract-rigid [x\1] (F x) T. % also checks H Prefix does not see x
contract-rigid L (ho.apply [H\[Arg\]]) T :=
  rev L Lrev, append Prefix Lrev Args,
  if (Prefix = []) (T = H) (T = ho.apply [H]\[Prefix\]).

progress-beta-link ho.tm -> ho.tm -> ho.subst -> ho.subst -> o.
progress-beta-link (ho.lambda x\ as T) (ho.lambda x\ as T) H H1 [] := !,
  (T as T1) H H1.

is-in-pf (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.not_occ Ad H T2) true fail.

is-pf (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

is-pf (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

is-pf (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

arity (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.

if (ho.lambda x\ as x) T H H1 [] := !,
  (T as T1) H H1.
THE DECOMPILER

15 THE DECOMPILER

progress-beta-link A B S S1 NewLinks.

type take-link link -> links -> link -> links -> o.
take-link A B[X|Y] B X5 := link-abs-same-lhs A B, !.

type link-abs-same-lhs link -> link -> o.
link-abs-same-lhs (ho.abs A B) :=
pi x \ link-abs-same-lhs (F x) B.
link-abs-same-lhs (ho.abs A G) :=
pi x \ link-abs-same-lhs A (G x).
link-abs-same-lhs (@val-link-eta (ho.uva N _) _) (@val-link-eta (ho.pi X Y) tag->f -> x y \ tm->fml (B1 x) (B2 y)).
same-link-eta (ho.abs A B) H H1 :=
pi x \ same-link-eta (F x) B.
same-link-eta (ho.abs A G) H H1 :=
pi x \ same-link-eta A (G x).
same-link-eta (@val-link-eta (ho.uva N S1) A)
@ (ho.abs A (ho.uva N S2) B) H H1 :=
std.map2 S1 S2 (x|y) \ r = ho.copy x y),
Perm \ ho.copy A A',
(A' =\ B) H H1.

progress! [ ] X X.
progress! [A|L1] [A|L3] S S2 := take-link A L1 B L2, !,
same-link-eta A B S S1,
progress! L2 L3 S1 S2,
same-link-eta L0|L1 L3 S2 := deref-link L S L0, L,
solve-link-abs L R S S1, !,
progress! L1 L2 S1 S2, append R L2 L3.

15 THE DECOMPILER

type abs->lam ho.assignment -> ho.tm -> o.
abs->lam (ho.abs A) := !, pi x \ abs->lam (T x) (R x).
abs->lam (ho.val A) A.

type commit-links-aux link -> ho.subst -> ho.subst -> o.
commit-links-aux (@val-link-eta T1 T2) H1 H2 :=
ho.deref H1 T1 T1', ho.deref H1 T2 T2',
(T1' == T2') H1 H2.
commit-links-aux (@val-link-llam T1 T2) H1 H2 :=
ho.deref H1 T1 T1', ho.deref H1 T2 T2',
(T1' == T2') H1 H2.
commit-links-aux (ho.abs B) H H1 :=
pi x \ commit-links-aux (B x) H H1.

commit-links-aux links -> ho.subst -> ho.subst -> o.
commit-links [ ] [] H H.
commit-links [Abs | Links] H H2 :=
commit-links-aux Abs H H1, !, commit-links Links H H2.

type decompl-subst map -> map -> ho.subst -> ho.subst -> o.
decompl-subst [ ] _ _ F F.
decompl-subst Map [mapping (fv VO) (hv VM _)][T1] H F F2 :=
mem.set? VM H T, !,
ho.deref-assmt H T TTT,
abs->lam TTT T', tm->fml Map T' T1,
ho.eta-contrace T1 T2, mem.assign VO F T2 F1,
decompl-subst Map T1 H F1 F2.
decompl-subst Map [mapping _ (hv VM _)][T1] H F F2 :=
mem.unset? VM H, decompl-subst Map T1 H F F2.

type tm->fml map -> ho.tm -> map ->
ho.tm -> fml -> o.
add-new-map-aux _ _ [] [ ] S S.
add-new-map-aux H [T|Ts] L L2 S2 :=
add-new-map [T | T L1 L S1],
add-new-map-aux H Ts L1 L2 S1 S2.

add-new-map-ho.tm -> map ->
ho.tm -> fml -> o.
add-new-map _ (ho.uva N L) Map [Map1 | MapL] F1 F3 :=
mem.new F1 F2,
len L Arity, Mapl = mapping (fv M) (hv N (arity Arity)),
add-new-map H (ho.app L) Map [Map1 | MapL] F1 F3 :=
mem.new F1 F2,
add-new-map-ho.tm -> map ->
ho.tm -> fml -> o.
add-new-map _ (ho.uva N L) Map [Map1 | MapL] F1 F3 :=
mem.new F1 F2,
add-new-map-ho.tm -> map -> map ->
ho.tm -> fml -> o.
add-new-map _ (ho.uva N L) Map [] F1 F3 :=
mem.new F1 F2,
add-new-map-ho.tm -> map -> map ->
ho.tm -> fml -> o.
add-new-map _ (ho.uva N L) Map [Map1 | MapL] F1 F3 :=
mem.new F1 F2,
type decompile map -> links -> hosubst ->
   fo.fsubst -> fo.fsubst -> o.

decompile Map1 L HO F0 F02 :-
   commit-links L L1_ HO HO1, !,
   complete-mapping HO1 HO1 Map1 Map2 F0 F01,
   decomp1-subst Map2 Map2 HO1 F01 F02.

16 AUXILIARY FUNCTIONS

type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A ->
   list A1 -> B -> B -> C -> C -> o.

fold4 _ [] [] A A B B.
fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0,%
   fold4 F XS YS A0 A1 B0 B1.

type len list A -> nat -> o.

len [] z.
len [ _ |L] (s X) :- len L X.