



HAL
open science

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-04547069v1

HAL Id: hal-04547069

<https://inria.hal.science/hal-04547069v1>

Preprint submitted on 15 Apr 2024 (v1), last revised 13 May 2024 (v3)

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

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



Distributed under a Creative Commons Attribution 4.0 International License

Higher-Order unification for free!

Reusing the meta-language unification for the object language

Davide Fissore

davide.fissore@inria.fr

Université Côte d’Azur, Inria

France

Enrico Tassi

enrico.tassi@inria.fr

Université Côte d’Azur, Inria

France

ABSTRACT

Specifying and implementing a proof system from scratch requires significant effort. Logical Frameworks and Higher Order Logic Programming Languages provide dedicated, high-level Meta Languages (ML) to facilitate this task in two key ways: 1) variable binding and substitution are for free when ML binders represent object logic ones; 2) proof construction, and even proof search, are greatly simplified by leveraging the unification procedure provided by the ML. Notable examples of ML are Elf [18], Twelf [20], λ Prolog [13] and Isabelle [26] which have been utilized to implement various formal systems such as First Order Logic [5], Set Theory [17], Higher Order Logic [16], and even the Calculus of Constructions [4].

The object logic we are interested in is Coq’s Calculus of Inductive Constructions (CIC)[24]. We aim to develop a higher-order unification-based proof search procedure for it using the ML Elpi [3], a dialect of λ Prolog. Elpi’s equational theory includes $\eta\beta$ equivalence and features a higher-order unification procedure \approx_λ restricted to the pattern fragment [12]. Elpi offers an encoding of CIC suitable for meta-programming [23, 22, 8, 6] but restricts \approx_λ to roughly first-order unification problems only. We refer to this basic encoding as \mathcal{F}_0 .

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

KEYWORDS

Logic Programming, Meta-Programming, Higher-Order Unification

ACM Reference Format:

Davide Fissore and Enrico Tassi. 2024. Higher-Order unification for free!: Reusing the meta-language unification for the object language. In . ACM, New York, NY, USA, 21 pages. <https://doi.org/10.1145/nmnnnnn.nnnnnnn>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference’17, July 2017, Washington, DC, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
<https://doi.org/10.1145/nmnnnnn.nnnnnnn>

1 INTRODUCTION

Meta languages such as Elf [18], Twelf [20], λ Prolog [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 [24] 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 λ Prolog, 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:  $\forall n$ , Finite (fin n). (* r1 *)
```

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

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

```
 $\forall x:A$ , Decision (P x)  $\rightarrow$  Decision ( $\forall x:A$ , P x).
```

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

```
Decision ( $\forall x$ : fin 7, nfact x 3) (* g *)
```

The proof found by the solver back-chains on rule 3 (the only rule about the \forall quantifier), and then solves the premises with rules 1 and 2 respectively. Note that rule 3 features a second-order parameter P that represents a function of type $A \rightarrow \text{Prop}$ (a predicate over A). The solver has to infer a value for P by unifying the conclusion of rule 3 with the goal, and in particular, it has to solve the unification problem $P\ x = \text{nfact}\ x\ 3$. This higher-order problem falls in the so-called pattern-fragment \mathcal{L} [12] and admits a unique solution ρ that assigns the term $\lambda x. \text{nfact}\ x\ 3$ to P .

In order to implement such a search in Elpi, we shall describe the encoding of CIC terms and then the encoding of instances as rules. Elpi comes equipped with an Higher Order Abstract Syntax (HOAS [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 λ Prolog [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 `« $\forall y:t$, 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 w\ p` introduces a fresh nominal constant `w` for the premise `p`.

```

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2 PROBLEM STATEMENT AND SOLUTION

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

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

To reason about unification, we provide a description of the \mathcal{F}_0 and \mathcal{H}_0 languages where unification variables are first-class terms, i.e. they have a concrete syntax as shown in fig. 1. Unification variables in \mathcal{F}_0 (fuva term constructor) have no explicit scope: the arguments of a higher-order variable are given via the `fapp` constructor. For example the term `«P x»` is represented as `«fapp [fuva N, x]»`, where `N` is the memory address of `P` and `x` is a bound variable.

In \mathcal{H}_0 , the representation of `«P x»` is instead `«uva 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 fuva addr -> fm.     type uva addr -> list tm -> tm.

```

Figure 1: The \mathcal{F}_0 and \mathcal{H}_0 languages

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

```

241   f a           app [con "f", con "a"]
242    $\lambda x.\lambda y.F_{xy}$    lam x\< \ lam y\< \ uva F [x, y]
243    $\lambda x.F_x a$        lam x\< \ app [uva F [x], con "a"]
244    $\lambda x.F_x x$        lam x\< \ app [uva F [x], x]

```

When it is clear from the context, we shall use the same syntax for \mathcal{F}_o terms (although we never subscript unification variables). We use s, s_1, \dots for terms in \mathcal{F}_o and t, t_1, \dots for terms in \mathcal{H}_o .

2.2 Equational theories an unification

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

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

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

```

261   type (=o) fm -> fm -> o.           (=o)
262   fcon X =o fcon X.
263   fapp A =o fapp B :- forall2 (=o) A B.
264   flam F =o flam G :- pi x\< \ x =o x => F x =o G x.
265   fuva N =o fuva N.
266   flam F =o T :-                       (eta)
267     pi x\< \ beta T [x] (T' x), x =o x => F x =o T' x.
268   T =o flam F :-                       (eta_r)
269     pi x\< \ beta T [x] (T' x), x =o x => T' x =o F x.
270   fapp [flam X | L] =o T :- beta (flam X) L R, R =o T. (beta_l)
271   T =o fapp [flam X | L] :- beta (flam X) L R, T =o R. (beta_r)
272
273   type (=lambda) tm -> tm -> o.
274   con C =lambda fcon C.
275   app A =lambda fapp B :- forall2 (=lambda) A B.
276   lam F =lambda flam G :- pi x\< \ x =lambda x => F x =lambda G x.
277   uva N A =lambda fuva N B :- forall2 (=lambda) A B.

```

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

Substitution: ρs and σt . We write $\sigma = \{X \mapsto t\}$ for the substitution that assigns the term t to the variable X . We write σt for the application of the substitution to a term t , and $\sigma X = \{\sigma t \mid t \in X\}$ when X is a set of terms. We write $\sigma \subseteq \sigma'$ when σ is more general than σ' . We shall use ρ for \mathcal{F}_o substitutions, and σ for the \mathcal{H}_o ones.

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

Term unification: \approx_o vs. \approx_λ . \mathcal{H}_o 's unification signature is:

```

295   type ( $\approx_\lambda$ ) tm -> tm -> subst -> subst -> o.
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348

```

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

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

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

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

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

2.3 The problem: logic-program simulation

We represent a logic program *run* in \mathcal{F}_o as a sequence of *steps* of length \mathcal{N} . At each step, p we unify two terms, \mathbb{P}_{p_l} and \mathbb{P}_{p_r} , taken from the list of all unification problems \mathbb{P} . The composition of these steps starting from the empty substitution ρ_0 produces the final substitution $\rho_{\mathcal{N}}$, which is the result of the logic program execution.

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

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

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

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

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

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

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

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

PROPOSITION 2.2 (SIMULATION FIDELITY). *In the context of hrun , if $\mathbb{P} \subseteq \mathcal{L}$ we have that $\forall p \in 1 \dots \mathcal{N}$,*

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

In particular, this property guarantees that a *failure* in the \mathcal{F}_0 run is matched by a failure in \mathcal{H}_0 at the same step. We consider this property very important from a practical point of view since it guarantees that the execution traces are strongly related, and in turn, this enables a user to debug a logic program in \mathcal{F}_0 by looking at its execution trace in \mathcal{H}_0 .

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

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

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

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

This property has practical relevance since in many logic programming implementations, including Elpi, the order in which unification problems are tackled does matter. The simplest example is the sequence $F \simeq \lambda x.a$ and $F a \simeq a$: the second problem is not in \mathcal{L} and has two unifiers, namely $\sigma_1 = \{ F \mapsto \lambda x.x \}$ and $\sigma_2 = \{ F \mapsto \lambda x.a \}$. The first problem picks σ_2 , making the second problem re-enter \mathcal{L} .

Backtracking. We omit it from our model of 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 $h \simeq_{\lambda} p$. As a result \simeq_{λ} is “well behaved” on t , in the sense that it does not contradict $=_o$ as it would otherwise do on the “problematic” sub-terms.

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

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

An example of a $\diamond\beta$ term is the application $F x$. This term is problematic since the application node of its syntax tree cannot be used to justify a unification failure, i.e. by properly instantiating F the term head constructor may become a λ , or a constant, or remain an application.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3 BASIC COMPILATION AND SIMULATION

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

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

```

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

```

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

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

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

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

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

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

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

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

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

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

It is worth examining the code of `deref`, which applies the substitution to a \mathcal{H}_0 term. Notice how assignments are moved to the current scope, i.e. the `abs`-bound variables are renamed with the names in the scope of the unification variable occurrence.

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

Note that `move` strongly relies on invariant 1: the length of the arguments of all occurrences of a unification variable is the same. Hence, they have the same simple type for the meta-level, and

therefore the number of `abs` nodes in the assignment matches that length. This guarantees that `move` never fails.

```
type move assignment -> list tm -> tm -> o.
move (abs Bo) [H|L] R :- move (Bo H) L R.
move (val A) [] A.
```

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

3.2 Links (\mathbb{L})

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

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

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

INVARIANT 2 (LINK LEFT HAND SIDE). *The left-hand side of a suspended link is a variable.*

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

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

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

3.3 Compilation

The simple compiler described in this section serves as a base for the extensions in sections 4, 5 and 7. Its main task is to beta normalize the term and map one syntax tree to the other. In order to bring back the substitution from \mathcal{H}_0 to \mathcal{F}_0 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.

```
type comp fm -> tm -> mmap -> mmap -> links -> links ->
  subst -> subst -> o.
comp (fcon C) (con C) M M L L S S.
comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :-
  comp-lam F F1 M1 M2 L1 L2 S1 S2.
comp (fuva A) (uva B [I]) M1 M2 L L S1 S2 :-
  m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :-
  fold6 comp A A1 M1 M2 L1 L2 S1 S2.
```

523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
568
569
570
571
572
573
574
575
576
577
578
579
580

D:non
mi
pi-
ace

```

581 type compile fm -> tm -> mmap -> mmap -> links -> links ->
582     subst -> subst -> o.
583 compile F G M1 M2 L1 L2 S1 S2 :-
584     beta-normal F F', comp F' G M1 M2 L1 L2 S1 S2.

```

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

explain
fold6

```

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

```

In the code above, the syntax $\text{pi } x \ y \dots$ is syntactic sugar for iterated pi abstraction, as in $\text{pi } x \ \text{pi } y \dots$.

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.

```

602 type close-links (tm -> links) -> links -> o.
603 close-links (v\ [X |L v] [X|R] :- !, close-links L R.
604 close-links (v\ [X v|L v] [abs X|R] :- close-links L R.
605 close-links (_\ []) [].

```

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

3.4 Execution

A step in \mathcal{H}_0 consists 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.

```

616 type hstep tm -> tm -> links -> links -> subst -> subst -> o.
617 hstep T1 T2 L1 L2 S1 S3 :-
618     (T1 ≈λ T2) S1 S2,
619     progress L1 L2 S2 S3.

```

Note that the infix notation $((A \approx_\lambda B) C D)$ is syntactic sugar for $((\approx_\lambda) A B C D)$.

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

```

626 type progress links -> links -> subst -> subst -> o.
627 progress L L2 S1 S3 :-
628     progress1 L L1 S1 S2,
629     occur-check-links L1,
630     if (L = L1, S1 = S2)
631         (L2 = L1, S3 = S1)
632         (progress L1 L2 S2 S3).

```

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

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

3.5 Substitution decompilation

Decompiling the substitution involves three steps.

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

The second step involves allocating new variables in the memory of \mathcal{F}_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 $F_{ab} \mapsto G_a$.

The final step is to decompile each assignment. Decompiling a term is straightforward since \mathbb{M} is a bijection. The only complex part concerns the `abs` node. In our simple setting, the `flam` node carries no additional information (other than the function body), so each `abs` node can be trivially converted to a `flam` 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 $\text{compile } S \ T \ [] \ M \ [] \ _ \ []$ then $\text{decompile } M \ T \ S$*

3.6 Definition of \approx_o and its properties

We already have all the pieces to show the code of \approx_λ .

```

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

```

So far the compiler is very basic. It does not really enforce that the terms passed to `hstep` are in \mathcal{W} , and indeed makes no use of the higher-order capabilities of the meta language (all generated variables have an empty scope). Still, we can prove that \approx_o is a good "first-order" unification algorithm if the input already happens to be in \mathcal{W} . Later, when the compiler will enforce proposition 2.8 the proof will be adjusted to cover for the new cases.

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

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

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

D:Should
also
de-
com-
pile
links...

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

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

PROOF SKETCH. Trivial since progress1 is a no-op and fstep and hstep are the same, and by lemma 3.2 \approx_λ is equivalent to \approx_o . \square

3.7 Notational conventions

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

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

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

4 HANDLING OF $\diamond\beta$

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

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

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

4.1 Compilation and decompilation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5 HANDLING OF $\diamond\eta$

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

$$\begin{aligned} \mathbb{P} &= \{ \lambda x.Xx \approx_o f \} \\ \mathbb{T} &= \{ \lambda x.A_x \approx_\lambda f \} \\ \mathbb{M} &= \{ X \mapsto A^1 \} \end{aligned}$$

While $\lambda x.Xx \approx_o f$ does admit the solution $\rho = \{X \mapsto f\}$, the corresponding problem in \mathbb{T} does not: $\text{lam } x \backslash \text{uva } A \ [x]$ and $\text{con} "f"$ start with different, rigid, term constructors hence \approx_λ fails.

In order to guarantee proposition 2.1, we detect lambda abstractions that can disappear by η -contraction (section 5.1), and we modify the compiler so that it generates fresh unification variables in their place and moves the problematic term from \mathbb{T} to \mathbb{L} (section 5.2). The compilation of the problem \mathbb{P} above is refined to:

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

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

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

Each η -link is kept in the link store \mathbb{L} during execution and is activated under some conditions. Activation is implemented by extending the progress1 predicate defined in section 3.4.

fix
in-
den-
ta-
tion

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

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 η -expansion, i.e. if there exists a substitution ρ such that $\rho(\lambda x.r) =_o s$. The detection of lambda abstractions that can “disappear” is not as trivial as it may seem. Here a few examples:

$$\begin{array}{ll} \lambda x.f(Ax) & \in \diamond\eta \quad \rho = \{A \mapsto \lambda x.x\} \\ \lambda x.f(Ax)x & \in \diamond\eta \quad \rho = \{A \mapsto \lambda x.a\} \\ \lambda x.fx(Ax) & \notin \diamond\eta \\ \lambda x.\lambda y.f(Ax)(Byx) & \in \diamond\eta \quad \rho = \{A \mapsto \lambda x.x, B \mapsto \lambda y.\lambda x.y\} \end{array}$$

The first two examples are easy, and show how a unification variable can expose or erase a variable in its scope, turning the resulting term into an η -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 η -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 $\lambda y.f(Ax)(Byx)$ is in $\diamond\eta$ itself. If it is, it could η -contract to $f(Ax)$ making $\lambda x.f(Ax)$ a potential η -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 β -normal term s may-contract-to a name x if there exists a substitution ρ such that $\rho s =_o x$.

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

- (1) $n = 0$ and $t = x$;
- (2) t is the application of x to a list of terms l and each l_i may-contract-to x_i (e.g. $\lambda x_1 \dots x_n.x x_1 \dots x_n =_o x$);
- (3) t is a unification variable with scope W , and for any $v \in \{x, x_1 \dots x_n\}$, there exists a $w_i \in W$, 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 β -normal form there is only one rule that can play a role (namely η_l), hence if the term s is not exactly x (case 1) it can only be an η -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 η contraction. In that case, the term t is under the spine of binders $x_1 \dots x_n$, t can either be x applied to terms that can may-contract-to these variables (case 2), or a unification variable that can be assigned to that application (case 3). \square

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

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

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

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

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

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

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

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

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

5.2 Compilation and decompilation

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

```
comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
  maybe-eta (flam F) [], !,
  alloc S1 A S2,
  comp-lam F F1 M1 M2 L1 L2 S2 S3,
  get-scope (lam F1) Scope,
  L3 = [val (link-eta (uva A Scope) (lam F1)) | L2].
```

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

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

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

Decompilation. The decompilation of a η -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 η -link (by definition 5.9).

5.3 Progress

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

929 *Definition 5.7 (η -progress-lhs).* A link $\Gamma \vdash X =_{\eta} T$ is removed
930 from \mathbb{L} when X becomes rigid. Let $y \in \Gamma$, there are two cases:

- 931 (1) if $X = a$ or $X = y$ or $X = f a_1 \dots a_n$ we unify the η -
932 expansion of X with T , that is we run $\lambda x.X x \simeq_{\lambda} T$
933 (2) if $X = \lambda x.t$ we run $X \simeq_{\lambda} T$.

934 *Definition 5.8 (η -progress-rhs).* A link $\Gamma \vdash X =_{\eta} T$ is removed
935 from \mathbb{L} when either 1) *maybe-eta* T does not hold (anymore) or
936 2) by η -contracting T to T' where T' does not start with the lam
937 constructor. In the first case, X is unified with T , and in the second
938 one, X is unified with T' (under the context Γ).

940 There is a third case in which a link is removed from \mathbb{L} , namely
941 when the lhs is assigned to a variable that is the lhs of another
942 η -link.

943 *Definition 5.9 (η -progress-deduplicate).* A link $\Gamma \vdash X_{\vec{s}} =_{\eta} T$
944 is removed from \mathbb{L} when another link $\Delta \vdash X_{\vec{r}} =_{\eta} T'$ is in \mathbb{L} . By
945 invariant 1 the length of \vec{s} and \vec{r} is the same; hence we can move
946 the term T' from Δ to Γ by renaming its bound variables, i.e. $T'' =$
947 $T'[\vec{r}/\vec{s}]$. We then run $T \simeq_{\lambda} T''$ (under the context Γ).

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

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

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

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

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

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

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

972 PROOF. We unify the rhs of two η -link. Both have the shape
973 $\lambda x.t$ and by lemma 5.10, $\mathcal{W}(t)$. \square

974 LEMMA 5.14. *The introduction of η -link guarantees proposition 2.8
975 (\mathcal{W} -ENFORCING)*

976 PROOF SKETCH. By definition 5.4. \square

977 LEMMA 5.15. *The progress of η -link guarantees proposition 2.9
978 (\mathcal{W} -PRESERVATION)*

979 PROOF SKETCH. By lemmas 5.11 to 5.13, every unification per-
980 formed by the activation of a η -link is done between terms in \mathcal{W} ,
981 therefore, the substitution remains \mathcal{W} . \square

LEMMA 5.16. *progress terminates.*

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

THEOREM 5.17 (FIDELITY IN $\diamond\eta$). *Given a list of unification prob-
lems \mathbb{P} , if $\forall t, t \in \mathcal{P}(\mathbb{P}) \Rightarrow t \notin \diamond\mathcal{L}$ 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-deduplicate acti-
vate a η -link when, in the original unification problem, a $\diamond\eta$ term
is unified with respectively a well-behaved term or another $\diamond\eta$
term. In both cases, the links trigger a unification which succeeds
iff the same unification in \mathcal{F}_0 succeeds, guaranteeing proposition 2.2.
 η -progress-rhs never fails, in fact, this progression refines a variable
to a rigid term and plays no role in proposition 2.2. \square

Example of η -progress-lhs. The example at the beginning of
section 5, once $\sigma = \{ A \mapsto f \}$, triggers η -progress-lhs since the
link becomes $\vdash f =_{\eta} \lambda x.B_x$ and the lhs is a constant. This rule
runs $\lambda x.f x \simeq_{\lambda} \lambda x.B_x$, resulting in $\sigma = \{ A \mapsto f; B_x \mapsto f \}$. By
the definition of η -link decompilation, decompilation generates
 $\rho = \{ X \mapsto \lambda x.f x \}$, since: X is mapped to B_x and B_x is decompiled
into $\lambda x.f x$.

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

$$\begin{aligned} \mathbb{P} &= \{ \lambda x.(X x) \simeq_o \lambda x.(Y x) \} \\ \mathbb{T} &= \{ A \simeq_{\lambda} C \} \\ \mathbb{M} &= \{ X \mapsto B^1 \quad Y \mapsto D^1 \} \\ \mathbb{L} &= \{ \vdash A =_{\eta} \lambda x.B_x \quad \vdash C =_{\eta} \lambda x.D_x \} \end{aligned}$$

The result of $A \simeq_{\lambda} C$ is that the two η -link share the same lhs. By
unifying the two rhs we get $\sigma = \{ A \mapsto C, B \mapsto D \}$. In turn, given
the map \mathbb{M} , this second assignment is decompiled to $\rho = \{ X \mapsto Y \}$
as expected.

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

6 MAKING \mathbb{M} A BIJECTION

In section 3.1, we introduced the definition of memory map \mathbb{M} . This
map allows to decompile the \mathcal{H}_0 substitution to the object language.
Given the higher order nature of CIC the same unification variable
 X is used multiple times with different arities.

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

In the unification problems \mathbb{P} above, we see that X is used with
arity 2 in \mathbb{P}_1 and with arity 1 in \mathbb{P}_2 . In order to preserve invariant 1
(UNIFICATION-VARIABLE ARITY) the compiler did generate two en-
tries in \mathbb{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

is w
sig-
maa
de-
fined?

what
does
as a
func-
tor
mean?

shall
we
move
the
proof
here?

D:rip
sec
3.1

Fix:
show
that
5.7
an
5.8
com-
pen-
sate
for
miss-
ing
eta
rule

maps to, namely E and C , would not be detected. We remedy by post-processing the compiler's output by adding additional η -link to ensure the followgin property:

PROPOSITION 6.1 (\mathbb{M} IS A BIJECTION). *After compilation, for each \mathcal{F}_0 variable X in \mathbb{P} and for each \mathcal{H}_0 variable A in \mathbb{T} there is exactly one entry $X \mapsto A^n$ (for any arity n).*

Note that the execution in \mathcal{H}_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 $0 \leq i < d$,

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

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

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

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

THEOREM 6.4 (FIDELITY WITH MAP-DEDUPLICATION). *Given a list of unification problems \mathbb{P} , such that $\forall t, t \in \mathcal{P}(\mathbb{P}) \Rightarrow \mathcal{W}(t) \vee t \in \diamond\eta$, if \mathbb{P} contains the same \mathcal{F}_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 \mathcal{F}_0 variables X_1, X_2 with different arities are compiled to two variables A^n and C^m related by a chain of η -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 η -link is triggered by η -progress-lhs instantiating the next variable in the chain of links. \square

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 η -link: $x \vdash E_x =_{\eta} \lambda y. C_{xy}$. After deduplication the compiler output is as follows:

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

The resolution of \mathbb{T}_1 assigns $\lambda x. \lambda y. x$ to A that in turn triggers \mathbb{L}_3 and then \mathbb{L}_4 by η -progress-lhs, one per λ in the solution of A . The unification variable C_{yx} is therefore unified with x (the second

variable of its scope). As a result \mathbb{L}_1 becomes $x \vdash E_x =_{\eta} \lambda y. y$, and the rhs is no more in $\diamond\eta$ so η -progress-rhs fires and E_x is unified with $\lambda y. y$, resulting in $\sigma = \{ A \mapsto \lambda x. \lambda y. x; B_x \mapsto \lambda y. x; C_{yx} \mapsto x; E_x \mapsto \lambda y. y \}$.

The remaining step \mathbb{T}_2 identifies D with F , hence the resulting \mathcal{F}_0 substitution is $\rho = \{ X \mapsto \lambda x. \lambda y. y; Y \mapsto \lambda x. (f (\lambda y. y)) x \}$.

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

Even if we do not require \approx_{λ} to handle problems outside of \mathcal{L} it is worth handling terms in $\diamond\mathcal{L}$ 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 \mathbb{P}_2 is $X a \approx_o a$ that admits two different substitutions: $\rho_1 = \{ X \mapsto \lambda x. x \}$ 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:

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

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

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

Note that the shape of such as rhs is $\langle\langle \text{app } [uva \ X \ S \ | \ L] \rangle\rangle$, where $S = s_1 \dots s_n$ and $L = t_1 \dots t_m$.

7.1 Compilation and decompilation

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

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

The list Ag is split into the two parts: Pf is in \mathcal{L} , and can be empty; $Extra$ cannot be empty and is such that append $Pf Extra Ag$. The rhs of the \mathcal{L} -link is the application of a fresh variable C having in scope all names in $Pf1$ (the compilation of Pf). The variable B ,

Not sure I fully fixed the proof, but the old oen was not ok

1103
1104
1105
1106
1107
verificare,
la
vec-
chia
soluzione
era
sbal-
lata
1110
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160

returned has the compiled term, is a fresh variable having in scope all the free variables occurring in `Pf1` and `Extra1` (the compilation of `Extra`). This construction enforces invariant 4.

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

7.2 Progress

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

Definition 7.1 (\mathcal{L} -progress-refine). Let σ be a substitution such that σt_1 is a name s not occurring in $s_1 \dots s_n$. If $m = 1$, then l is removed and lhs is unified with $X_{s_1 \dots s_n} s$. If $m > 1$, then l is replaced by the link $\Gamma \vdash T =_{\mathcal{L}} Y_{s_1 \dots s_n} s t_2 \dots t_m$ (where Y is a fresh variable of arity $n + 1$) and link $\Gamma \vdash X_{s_1 \dots s_n} =_{\eta} \lambda x. Y_{s_1 \dots s_n} x$ is added to \mathbb{L} .

Definition 7.2 (\mathcal{L} -progress-rhs). Link l is removed from \mathbb{L} if $X_{s_1 \dots s_n}$ is instantiated to a term t and $t t_1 \dots t_m \beta$ -reduces to a $t' \in \mathbb{L}$.

Definition 7.3 (\mathcal{L} -progress-fail). progress fails whenever: there exists another link $l' \in \mathbb{L}$ 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 a \mathcal{L} -link in the store \mathbb{L} . If l is activated by \mathcal{L} -progress-rhs, then it disappears from \mathbb{L} and progress terminates. Otherwise, the rhs of l is made by a variable applied to m arguments. At each activation of \mathcal{L} -progress-refine, l is replaced by a new \mathcal{L} -link l^1 having $m - 1$ arguments. At the m^{th} iteration, the \mathcal{L} -link l^m has no more arguments and is removed from \mathbb{L} . Note that at the m^{th} iteration, m new η -link have been added to \mathbb{L} , however, by lemma 5.16, the algorithm terminates. Finally \mathcal{L} -progress-fail also guarantees termination since it makes progress immediately fails. \square

THEOREM 7.5 (FIDELITY WITH \mathcal{L} -link). The introduction of \mathcal{L} -link guarantees proposition 2.3 (FIDELITY RECOVERY) if \approx_o fails outside \mathcal{L} .

PROOF SKETCH. Let \mathbb{P}_i be the first problem in \mathbb{P} not in \mathcal{L} and let σ be the substitution obtained solving $\mathbb{T}_1 \dots \mathbb{T}_{i-1}$. If $\sigma \mathbb{T}_i$ is in \mathcal{L} , then each variable that the compiler used to replace a problematic term is assigned and hence by definitions 7.1 and 7.2 the associated \mathcal{L} -link been solved and removed. Note that in this case all calls to \approx_{λ} are between terms in \mathcal{L} and by theorem 5.17 fidelity is guaranteed. If $\sigma \mathbb{T}_i$ is still in $\diamond \mathcal{L}$, then by definition 7.3 unification fails as the corresponding unification in \mathcal{F}_o would. \square

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

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

Initially the \mathcal{L} -link rhs is a variable D applied to the E_x . The first unification problem results in $\sigma = \{ A \mapsto \lambda x.x \}$. In turn this instantiation triggers \mathbb{L}_1 by η -progress-lhs and E_x is assigned to x . Under this substitution the \mathcal{L} -link becomes $x \vdash C_x =_{\mathcal{L}} (D x)$, and by \mathcal{L} -progress-refine it is replaced with the link: $\vdash E =_{\eta} \lambda x.D_x$, while C_x is unified with D_x . The second unification problem assigns f to B , that in turn activates the second η -link (f is assigned to C), and then all the remaining links are solved. The final \mathcal{H}_o substitution is $\sigma = \{ A \mapsto \lambda x.x; B \mapsto f; C_x \mapsto (f x); D \mapsto f; E_x \mapsto x; F_x \mapsto C_x \}$ and is decompiled into $\rho = \{ X \mapsto \lambda x.x; Y \mapsto f \}$.

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

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

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

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

Working with terms in \mathcal{L} is sometime too restrictive [1] and we could find in literature a few strategies to go beyond \mathcal{L} without implementing Huet's algorithm [10]. Some implementations of λ Prolog [15] such as Teyjus [14] delay the resolution of $\diamond \mathcal{L}$ unification problems until the substitution makes them reenter \mathcal{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 \mathbb{P}_1 is in $\diamond \mathcal{L}$.

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

If we want the object language unification to delay the first unification problem (waiting for X to be instantiated), we can relax definition 7.3. Instead of failing when the lhs of \mathbb{L}_1 becomes rigid (equal to a), we keep it in \mathbb{L} until the head of its rhs also become rigid. In this case, since both the lhs and rhs have rigid heads, they can be unified. While this relaxed rule does not break proposition 2.2 (SIMULATION FIDELITY) per se, the occur-check-links procedure becomes incomplete since invariant 2 (LINK LEFT HAND SIDE) is broken. Also note that delaying unification outside \mathcal{L} can leave \mathcal{L} -link for the decompilation phase. Therefore commit-links should be modified accordingly.

If instead we want \approx_o 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:

I changed this part, it is OK?

this is a little more complex than the opening problem

```

1277 type app-align-heuristic tm -> tm -> subst -> subst -> o.
1278 app-align-heuristic (app [HD|L1]) (app [UV|L2]) S1 S2 :-
1279   append Prefix L2 L1,
1280   ho.beta HD Prefix T,
1281   (UV  $\approx_\lambda$  T) S1 S2.

```

Note that UV is in \mathcal{L} as well as T .

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 \mathcal{H}_o 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_λ , but `hstep` also needs to check for progress. Luckily Elpi extends [9] λ 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 η -link are suspended or make progress.

```

1302 link-eta L R :- var L, maybe-eta R HolesInR, !,
1303   declare_constraint (link-eta L R) [L|HolesInR].
1304 link-eta L R :- not (var L), !, eta-progress-lhs L R.
1305 link-eta L R :- not (maybe-eta R), !, contract R S, L = S.

```

improve
code

The second snippet illustrate the deduplication of η -link. The syntax $\langle N \triangleright G \text{ ?- } P \rangle$ denotes a λ Prolog sequent, that is a goal P under a set G of hypothetical rules (introduced by \Rightarrow) and where the program context binds (via the π operator) `elpiN` heigenvariables. Sequents are presented to CHR with their higher order unification variables replaced by “frozen constants”, that is A_{xyz} becomes $\langle \text{uvar } f_A [x,y,z] \rangle$ for a fresh constant f_A . Frozen constants are “defrost” when they are part of a new goal (see [9, section 4.3]).

```

1316 constraint link-eta {
1317   rule (N1  $\triangleright$  G1 ?- link-eta (uvar X LX1) T1) % match
1318     / (N2  $\triangleright$  G2 ?- link-eta (uvar X LX2) T2) % remove
1319     | (relocate LX1 LX2 T2 T2') % condition
1320     <=> (N1  $\triangleright$  G1 ?- T1 = T2'). % new goal

```

The first directive matches a constraint whose lhs is a variable X ; the second matches *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_o as a regular routine in the ML, i.e. write rules as follows:

```

1332 decision X :- X  $\approx_o$  (all A  $\times$  app [P, x]), finite A,
1333   pi x \ decision (app [P, x]).

```

1334

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:

```

1342 decision (all A  $\times$  P x) :- finite A, pi x \ decision (P x).

```

However, this encoding has two big limitations. First it is not always feasible to adopt it for 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 λ 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 arder 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.

sucks

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 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 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:

- 1393 2018. URL: <https://api.semanticscholar.org/CorpusID:51885863>.
1394
- 1395 [2] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT
1396 Press, 2013. ISBN: 0262026651. 1451
- 1397 [3] Cvetan Dunchev et al. “ELPI: Fast, Embeddable, λ Prolog
1398 Interpreter”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin
1399 Davis et al. Vol. 9450. 2015, pp. 460–468. doi: 10.1007/978-3-
1400 662-48899-7_32. URL: http://dx.doi.org/10.1007/978-3-662-48899-7_32. 1452
- 1401 [4] Amy Felty. “Encoding the Calculus of Constructions in a
1402 Higher-Order Logic”. In: ed. by M. Vardi. IEEE, June 1993,
1403 pp. 233–244. doi: 10.1109/LICS.1993.287584. 1453
- 1404 [5] Amy Felty and Dale Miller. “Specifying theorem provers
1405 in a higher-order logic programming language”. In: *Ninth International Conference on Automated Deduction*. Ed. by
1406 Ewing Lusk and Ross Overbeck. 310. Argonne, IL: Springer,
1407 May 1988, pp. 61–80. doi: 10.1007/BFb0012823. 1454
- 1408 [6] Davide Fissore and Enrico Tassi. “A new Type-Class solver
1409 for Coq in Elpi”. In: *The Coq Workshop 2023*. Bialystok, Poland,
1410 July 2023. URL: <https://inria.hal.science/hal-04467855>. 1455
- 1411 [7] Thom Fruehwirth. *Constraint Handling Rules - What Else?*
1412 2017. arXiv: 1701.02668 [cs.PL]. 1456
- 1413 [8] Benjamin Grégoire, Jean-Christophe L echenet, and Enrico
1414 Tassi. “Practical and sound equality tests, automatically –
1415 Deriving eqType instances for Jasmin’s data types with Coq-
1416 Elpi”. In: *CPP ’23: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023: Proceedings
1417 of the 12th ACM SIGPLAN International Conference on Certified
1418 Programs and Proofs. Boston MA USA, France: ACM,
1419 Jan. 2023, pp. 167–181. doi: 10.1145/3573105.3575683. URL:
1420 <https://inria.hal.science/hal-03800154>. 1457
- 1421 [9] FERRUCCIO GUIDI, CLAUDIO SACERDOTI COEN, and
1422 ENRICO TASSI. “Implementing type theory in higher order
1423 constraint logic programming”. In: *Mathematical Structures in Computer Science* 29.8 (2019), pp. 1125–1150. doi: 10.1017/
1424 S0960129518000427. 1458
- 1425 [10] G erard P. Huet. “A Unification Algorithm for Typed lambda-
1426 Calculus”. In: *Theor. Comput. Sci.* 1.1 (1975), pp. 27–57. doi:
1427 10.1016/0304-3975(75)90011-0. URL: [https://doi.org/10.1016/0304-3975\(75\)90011-0](https://doi.org/10.1016/0304-3975(75)90011-0). 1459
- 1428 [11] RALF JUNG et al. “Iris from the ground up: A modular
1429 foundation for higher-order concurrent separation logic”.
1430 In: *Journal of Functional Programming* 28 (2018), e20. doi:
1431 10.1017/S0956796818000151. 1460
- 1432 [12] Dale Miller. “Unification under a mixed prefix”. In: *Journal of Symbolic Computation* 14.4 (1992), pp. 321–358. doi: 10.1016/0747-7171(92)90011-R. 1461
- 1433 [13] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. doi: 10.1017/CBO9781139021326. 1462
- 1434 [14] Gopalan Nadathur. “The Metalanguage λ prolog and Its Implementation”. In: *Functional and Logic Programming*. Ed. by Herbert Kuchen and Kazunori Ueda. Berlin, Heidelberg:
1435 Springer Berlin Heidelberg, 2001, pp. 1–20. ISBN: 978-3-540-44716-0. 1463
- 1436 [15] Gopalan Nadathur and Dale Miller. “An Overview of Lambda-
1437 Prolog”. In: June 1988, pp. 810–827. 1464
- 1438 [16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. 1465
- 1439 [17] Lawrence C. Paulson. “Set theory for verification. I: from
1440 foundations to functions”. In: *J. Autom. Reason.* 11.3 (Dec. 1993), pp. 353–389. ISSN: 0168-7433. doi: 10.1007/BF00881873. URL: <https://doi.org/10.1007/BF00881873>. 1466
- 1441 [18] F. Pfenning. “Elf: a language for logic definition and verified
1442 metaprogramming”. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, California, USA: IEEE Press, 1989, pp. 313–322. ISBN: 0818619546. 1467
- 1443 [19] F. Pfenning and C. Elliott. “Higher-order abstract syntax”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 199–208. ISBN: 0897912691. doi: 10.1145/53990.54010. URL: <https://doi.org/10.1145/53990.54010>. 1468
- 1444 [20] Frank Pfenning and Carsten Sch urmann. “System Description: Twelf – A Meta-Logical Framework for Deductive Systems”. In: *Automated Deduction – CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206. ISBN: 978-3-540-48660-2. 1469
- 1445 [21] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, C esar Mu oz, and Sof iene Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-71067-7. 1470
- 1446 [22] Enrico Tassi. “Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq”. In: *ITP 2019 - 10th International Conference on Interactive Theorem Proving*. Portland, United States, Sept. 2019. doi: 10.4230/LIPIcs.CVIT.2016.23. URL: <https://inria.hal.science/hal-01897468>. 1471
- 1447 [23] Enrico Tassi. “Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog dialect)”. In: *The Fourth International Workshop on Coq for Programming Languages*. Los Angeles (CA), United States, Jan. 2018. URL: <https://inria.hal.science/hal-01637063>. 1472
- 1448 [24] The Coq Development Team. *The Coq Reference Manual – Release 8.18.0*. <https://coq.inria.fr/doc/V8.18.0/refman>. 2023. 1473
- 1449 [25] P. Wadler and S. Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. doi: 10.1145/75277.75283. URL: <https://doi.org/10.1145/75277.75283>. 1474
- 1450 [26] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. “The Isabelle Framework”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, C esar Mu oz, and Sof iene Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 33–38. ISBN: 978-3-540-71067-7. 1475
- 1451 1476
- 1452 1477
- 1453 1478
- 1454 1479
- 1455 1480
- 1456 1481
- 1457 1482
- 1458 1483
- 1459 1484
- 1460 1485
- 1461 1486
- 1462 1487
- 1463 1488
- 1464 1489
- 1465 1490
- 1466 1491
- 1467 1492
- 1468 1493
- 1469 1494
- 1470 1495
- 1471 1496
- 1472 1497
- 1473 1498
- 1474 1499
- 1475 1500
- 1476 1501
- 1477 1502
- 1478 1503
- 1479 1504
- 1480 1505
- 1481 1506
- 1482 1507
- 1483 1508

1509	[27] Markus Wenzel. “Type classes and overloading in higher-order logic”. In: <i>Theorem Proving in Higher Order Logics</i> . Ed. by Elsa L. Gunter and Amy Felty. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 307–322. ISBN: 978-3-540-69526-4.	1567
1510		1568
1511		1569
1512		1570
1513		1571
1514		1572
1515		1573
1516		1574
1517		1575
1518		1576
1519		1577
1520		1578
1521		1579
1522		1580
1523		1581
1524		1582
1525		1583
1526		1584
1527		1585
1528		1586
1529		1587
1530		1588
1531		1589
1532		1590
1533		1591
1534		1592
1535		1593
1536		1594
1537		1595
1538		1596
1539		1597
1540		1598
1541		1599
1542		1600
1543		1601
1544		1602
1545		1603
1546		1604
1547		1605
1548		1606
1549		1607
1550		1608
1551		1609
1552		1610
1553		1611
1554		1612
1555		1613
1556		1614
1557		1615
1558		1616
1559		1617
1560		1618
1561		1619
1562		1620
1563		1621
1564		1622
1565		1623
1566		1624

APPENDIX

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

```

1635 kind addr type.
1636 type addr nat -> addr.
1637 typeabbrev (mem A) (list (option A)).
1638 type set? addr -> mem A -> A -> o.
1639 set? (addr A) Mem Val :- get A Mem Val.
1640
1641 type unset? addr -> mem A -> o.
1642 unset? Addr Mem :- not (set? Addr Mem _).
1643
1644 type assign-aux nat -> mem A -> A -> mem A -> o.
1645 assign-aux z (none :: L) Y (some Y :: L).
1646 assign-aux (s N) (X :: L) Y (X :: L1) :- assign-aux N L Y L1.
1647
1648 type assign addr -> mem A -> A -> mem A -> o.
1649 assign (addr A) Mem1 Val Mem2 :- assign-aux A Mem1 Val Mem2.
1650
1651 type get nat -> mem A -> A -> o.
1652 get z (some Y :: _) Y.
1653 get (s N) (_ :: L) X :- get N L X.
1654
1655 type alloc-aux nat -> mem A -> mem A -> o.
1656 alloc-aux z [] [none] :- !.
1657 alloc-aux z L L.
1658 alloc-aux (s N) [] [none | M] :- alloc-aux N [] M.
1659 alloc-aux (s N) [X | L] [X | M] :- alloc-aux N L M.
1660
1661 type alloc addr -> mem A -> mem A -> o.
1662 alloc (addr A as Ad) Mem1 Mem2 :- unset? Ad Mem1,
1663   alloc-aux A Mem1 Mem2.
1664
1665 type new-aux mem A -> nat -> mem A -> o.
1666 new-aux [] z [none].
1667 new-aux [A | As] (s N) [A | Bs] :- new-aux As N Bs.
1668
1669 type new mem A -> addr -> mem A -> o.
1670 new Mem1 (addr Ad) Mem2 :- new-aux Mem1 Ad Mem2.

```

11 THE OBJECT LANGUAGE

```

1674 kind fm type.
1675 type fapp list fm -> fm.
1676 type flam (fm -> fm) -> fm.
1677 type fcon string -> fm.
1678 type fuva addr -> fm.
1679
1680 typeabbrev fsubst (mem fm).

```

```

1683 type fder fsubst -> fm -> fm -> o.
1684 fder _ (fcon C) (fcon C).
1685 fder S (fapp A) (fapp B) :- map (fder S) A B.
1686 fder S (flam F) (flam G) :-
1687   pi x\ fder S x x => fder S (F x) (G x).
1688 fder S (fuva N) R :- set? N S T, fder S T R.
1689 fder S (fuva N) (fuva N) :- unset? N S.
1690
1691 type fderef fsubst -> fm -> fm -> o. (ρs)
1692 fderef S T T2 :- fder S T T1, napp T1 T2.
1693
1694 type (=o) fm -> fm -> o. (=o)
1695 fcon X =o fcon X.
1696 fapp A =o fapp B :- forall2 (=o) A B.
1697 flam F =o flam G :- pi x\ x =o x => F x =o G x.
1698 fuva N =o fuva N.
1699 flam F =o T :- (ηl)
1700   pi x\ beta T [x] (T' x), x =o x => F x =o T' x.
1701 T =o flam F :- (ηr)
1702   pi x\ beta T [x] (T' x), x =o x => T' x =o F x.
1703 fapp [flam X | L] =o T :- beta (flam X) L R, R =o T. (βl)
1704 T =o fapp [flam X | L] :- beta (flam X) L R, T =o R. (βr)
1705
1706 type extend-subst fm -> fsubst -> fsubst -> o.
1707 extend-subst (fuva N) S S' :- mem.alloc N S S'.
1708 extend-subst (flam F) S S' :-
1709   pi x\ (pi S\ extend-subst x S S) => extend-subst (F x) S S'.
1710 extend-subst (fcon _) S S.
1711 extend-subst (fapp L) S S1 :- fold extend-subst L S S1.
1712
1713 type beta fm -> list fm -> fm -> o.
1714 beta A [] A.
1715 beta (flam Bo) [H | L] R :- napp (Bo H) F, beta F L R.
1716 beta (fapp A) L (fapp X) :- append A L X.
1717 beta (fuva N) L (fapp [fuva N | L]).
1718 beta (fcon H) L (fapp [fcon H | L]).
1719 beta N L (fapp [N | L]) :- name N.
1720
1721 type napp fm -> fm -> o.
1722 napp (fcon C) (fcon C).
1723 napp (fuva A) (fuva A).
1724 napp (flam F) (flam G) :- pi x\ napp (F x) (G x).
1725 napp (fapp [fapp L1 | L2]) T :- !,
1726   append L1 L2 L3, napp (fapp L3) T.
1727 napp (fapp L) (fapp L1) :- map napp L L1.
1728 napp N N :- name N.
1729
1730 type beta-normal fm -> fm -> o.
1731 beta-normal (uvar _ _) _ :- halt "Passed uvar to beta-normal".
1732 beta-normal A A :- name A.
1733 beta-normal (fcon A) (fcon A).
1734 beta-normal (fuva A) (fuva A).
1735 beta-normal (flam A) (flam B) :-
1736   pi x\ beta-normal (A x) (B x).
1737 beta-normal (fapp [flam B | L]) T2 :- !,
1738   beta (flam B) L T1, beta-normal T1 T2.
1739
1740

```



```

1741 beta-normal (fapp L) (fapp L1) :-
1742   map beta-normal L L1.
1743
1744 type mk-app fm -> list fm -> fm -> o.
1745 mk-app T L S :- beta T L S.
1746
1747 type eta-contract fm -> fm -> o.
1748 eta-contract (fcon X) (fcon X).
1749 eta-contract (fapp L) (fapp L1) :- map eta-contract L L1.
1750 eta-contract (flam F) T :- eta-contract-aux [] (flam F) T.
1751 eta-contract (flam F) (flam F1) :-
1752   pi x\ eta-contract x x => eta-contract (F x) (F1 x).
1753 eta-contract (fuva X) (fuva X).
1754 eta-contract X X :- name X.
1755
1756 type eta-contract-aux list fm -> fm -> fm -> o.
1757 eta-contract-aux L (flam F) T :-
1758   pi x\ eta-contract-aux [x|L] (F x) T. % also checks H Prefix does not x
1759 eta-contract-aux L (fapp [H|Args]) T :-
1760   rev L LRev, append Prefix LRev Args,
1761   if (Prefix = []) (T = H) (T = fapp [H|Prefix]).
1762
1763
1764 12 THE META LANGUAGE
1765 kind inctx type -> type. (· F ·)
1766 type abs (tm -> inctx A) -> inctx A.
1767 type val A -> inctx A.
1768 typeabbrev assignment (inctx tm).
1769 typeabbrev subst (mem assignment).
1770
1771 kind tm type.
1772 type app list tm -> tm.
1773 type lam (tm -> tm) -> tm.
1774 type con string -> tm.
1775 type uva addr -> list tm -> tm.
1776
1777 type ( $\approx_\lambda$ ) tm -> tm -> subst -> subst -> o.
1778 (con C  $\approx_\lambda$  con C) S S.
1779 (app L1  $\approx_\lambda$  app L2) S S1 :- fold2 ( $\approx_\lambda$ ) L1 L2 S S1.
1780 (lam F1  $\approx_\lambda$  lam F2) S S1 :-
1781   pi x\ (pi S\ (x  $\approx_\lambda$  x) S S) => (F1 x  $\approx_\lambda$  F2 x) S S1.
1782 (uva N Args  $\approx_\lambda$  T) S S1 :-
1783   set? N S F,!, move F Args T1, (T1  $\approx_\lambda$  T) S S1.
1784 (T  $\approx_\lambda$  uva N Args) S S1 :-
1785   set? N S F,!, move F Args T1, (T  $\approx_\lambda$  T1) S S1.
1786 (uva M A1  $\approx_\lambda$  uva N A2) S1 S2 :- !,
1787   pattern-fragment A1, pattern-fragment A2,
1788   prune! M A1 N A2 S1 S2.
1789 (uva N Args  $\approx_\lambda$  T) S S1 :- not_occ N S T, pattern-fragment Args,
1790   bind T Args T1, assign N S T1 S1.
1791 (T  $\approx_\lambda$  uva N Args) S S1 :- not_occ N S T, pattern-fragment Args,
1792   bind T Args T1, assign N S T1 S1.
1793
1794 type prune! addr -> list tm -> addr ->
1795   list tm -> subst -> subst -> o.
1796 /* no pruning needed */
1797 prune! N A N A S S :- !.
1798
1799 prune! M A N A S1 S2 :- !, bind (uva M A) A Ass,
1800   assign N S1 Ass S2.
1801 /* prune different arguments */
1802 prune! N A1 N A2 S1 S3 :- !,
1803   new S1 W S2, prune-same-variable W A1 A2 [] Ass,
1804   assign N S2 Ass S3.
1805 /* prune to the intersection of scopes */
1806 prune! N A1 M A2 S1 S4 :- !,
1807   new S1 W S2, prune-diff-variables W A1 A2 Ass1 Ass2,
1808   assign N S2 Ass1 S3,
1809   assign M S3 Ass2 S4.
1810
1811 type prune-same-variable addr -> list tm -> list tm ->
1812   list tm -> assignment -> o.
1813 prune-same-variable N [] [] ACC (val (uva N Args)) :-
1814   rev ACC Args.
1815 prune-same-variable N [X|XS] [X|YS] ACC (abs F) :-
1816   pi x\ prune-same-variable N XS YS [x|ACC] (F x).
1817 prune-same-variable N [_|XS] [_|YS] ACC (abs F) :-
1818   pi x\ prune-same-variable N XS YS ACC (F x).
1819
1820 type permute list nat -> list tm -> list tm -> o.
1821 permute [] _ [].
1822 permute [P|PS] Args [T|TS] :-
1823   nth P Args T,
1824   permute PS Args TS.
1825
1826 type build-perm-assign addr -> list tm -> list bool ->
1827   list nat -> assignment -> o.
1828 build-perm-assign N ArgsR [] Perm (val (uva N PermutedArgs)) :-
1829   rev ArgsR Args, permute Perm Args PermutedArgs.
1830 build-perm-assign N Acc [tt|L] Perm (abs T) :-
1831   pi x\ build-perm-assign N [x|Acc] L Perm (T x).
1832 build-perm-assign N Acc [ff|L] Perm (abs T) :-
1833   pi x\ build-perm-assign N Acc L Perm (T x).
1834
1835 type keep list A -> A -> bool -> o.
1836 keep L A tt :- mem L A, !.
1837 keep _ _ ff.
1838
1839 type prune-diff-variables addr -> list tm -> list tm ->
1840   assignment -> assignment -> o.
1841 prune-diff-variables N Args1 Args2 Ass1 Ass2 :-
1842   map (keep Args2) Args1 Bits1,
1843   map (keep Args1) Args2 Bits2,
1844   filter Args1 (mem Args2) ToKeep1,
1845   filter Args2 (mem Args1) ToKeep2,
1846   map (index ToKeep1) ToKeep1 IdPerm,
1847   map (index ToKeep1) ToKeep2 Perm21,
1848   build-perm-assign N [] Bits1 IdPerm Ass1,
1849   build-perm-assign N [] Bits2 Perm21 Ass2.
1850
1851 type beta tm -> list tm -> tm -> o.
1852 beta A [] A :- !.
1853 beta (lam Bo) [H | L] R :- beta (Bo H) L R1, beta-aux R1 R.
1854 beta (app A) L (app X) :- append A L X.
1855 beta (con H) L (app [con H | L]).
1856

```

```

1857 beta X L (app[X|L]) :- name X.
1858
1859 type beta-aux tm -> tm -> o.
1860 beta-aux (app [HD|TL]) R :- !, beta HD TL R.
1861 beta-aux A A.
1862
1863 /* occur check for N before crossing a functor */
1864 type not_occ addr -> subst -> tm -> o.
1865 not_occ N S (uva M Args) :- set? M S F,
1866   move F Args T, not_occ N S T.
1867 not_occ N S (uva M Args) :- unset? M S, not (M = N),
1868   forall1 (not_occ_aux N S) Args.
1869 not_occ _ _ (con _).
1870 not_occ N S (app L) :- not_occ_aux N S (app L).
1871 /* Note: lam is a functor for the meta language! */
1872 not_occ N S (lam L) :- pi x\ not_occ_aux N S (L x).
1873 not_occ _ _ X :- name X.
1874 /* finding N is ok */
1875 not_occ N _ (uva N _).
1876
1877 /* occur check for X after crossing a functor */
1878 type not_occ_aux addr -> subst -> tm -> o.
1879 not_occ_aux N S (uva M _) :- unset? M S, not (N = M).
1880 not_occ_aux N S (uva M Args) :- set? M S F,
1881   move F Args T, not_occ_aux N S T.
1882 not_occ_aux N S (app L) :- forall1 (not_occ_aux N S) L.
1883 not_occ_aux N S (lam F) :- pi x\ not_occ_aux N S (F x).
1884 not_occ_aux _ _ (con _).
1885 not_occ_aux _ _ X :- name X.
1886 /* finding N is ko, hence no rule */
1887
1888 /* copy T T' vails if T contains a free variable, i.e. it
1889   performs scope checking for bind */
1890 type copy tm -> tm -> o.
1891 copy (con C) (con C).
1892 copy (app L) (app L') :- map copy L L'.
1893 copy (lam T) (lam T') :- pi x\ copy x x => copy (T x) (T' x).
1894 copy (uva A L) (uva A L') :- map copy L L'.
1895
1896 type bind tm -> list tm -> assignment -> o.
1897 bind T [] (val T') :- copy T T'.
1898 bind T [X | TL] (abs T') :- pi x\ copy X x => bind T TL (T' x).
1899
1900 type deref subst -> tm -> tm -> o. (σt)
1901 deref _ (con C) (con C).
1902 deref S (app A) (app B) :- map (deref S) A B.
1903 deref S (lam F) (lam G) :-
1904   pi x\ deref S x x => deref S (F x) (G x).
1905 deref S (uva N L) R :- set? N S A,
1906   move A L T, deref S T R.
1907 deref S (uva N A) (uva N B) :- unset? N S,
1908   map (deref S) A B.
1909
1910 type move assignment -> list tm -> tm -> o.
1911 move (abs Bo) [H|L] R :- move (Bo H) L R.
1912 move (val A) [] A.

```

```

1915 type deref-assmt subst -> assignment -> assignment -> o.
1916 deref-assmt S (abs T) (abs R) :- pi x\ deref-assmt S (T x) (R x)
1917 deref-assmt S (val T) (val R) :- deref S T R.
1918
1919
1920

```

13 THE COMPILER

```

1921 kind arity type.
1922 type arity nat -> arity.
1923 kind fvariable type.
1924 type fv addr -> fvariable.
1925 kind hvariable type.
1926 type hv addr -> arity -> hvariable.
1927 kind mapping type.
1928 type (<->) fvariable -> hvariable -> mapping.
1929 typeabbrev mmap (list mapping).
1930
1931 typeabbrev scope (list tm).
1932 typeabbrev inctx ho.inctx.
1933 kind baselink type.
1934 type link-eta tm -> tm -> baselink.
1935 type link-llam tm -> tm -> baselink.
1936 typeabbrev link (inctx baselink).
1937 typeabbrev links (list link).
1938
1939 macro @val-link-eta T1 T2 :- ho.val (link-eta T1 T2).
1940 macro @val-link-llam T1 T2 :- ho.val (link-llam T1 T2).
1941
1942 type get-lhs link -> tm -> o.
1943 get-lhs (val (link-llam A _)) A.
1944 get-lhs (val (link-eta A _)) A.
1945
1946 type get-rhs link -> tm -> o.
1947 get-rhs (val (link-llam _ A)) A.
1948 get-rhs (val (link-eta _ A)) A.
1949
1950 type occurs-rigidly fm -> fm -> o.
1951 occurs-rigidly N N.
1952 occurs-rigidly _ (fapp [fuva _|_] ) :- !, fail.
1953 occurs-rigidly N (fapp L) :- exists (occurs-rigidly N) L.
1954 occurs-rigidly N (flam B) :- pi x\ occurs-rigidly N (B x).
1955
1956 type reducible-to list fm -> fm -> fm -> o.
1957 reducible-to _ N N :- !.
1958 reducible-to L N (fapp [fuva _|Args] ) :- !,
1959   forall1 (x\ exists (reducible-to [] x) Args) [N|L].
1960 reducible-to L N (flam B) :- !,
1961   pi x\ reducible-to [x | L] N (B x).
1962 reducible-to L N (fapp [N|Args] ) :-
1963   last-n {len L} Args R,
1964   forall2 (reducible-to []) R {rev L}.
1965
1966 type maybe-eta fm -> list fm -> o. (◇η)
1967 maybe-eta (fapp [fuva _|Args] ) L :- !,
1968   forall1 (x\ exists (reducible-to [] x) Args) L, !.
1969
1970
1971

```

```

1973 maybe-eta (flam B) L :- !, pi x\ maybe-eta (B x) [x | L].
1974 maybe-eta (fapp [T|Args]) L :- (name T; T = fcon _),
1975   split-last-n {len L} Args First Last,
1976   none (x\ exists (y\ occurs-rigidly x y) First) L,
1977   forall2 (reducible-to []) {rev L} Last.
1978
1979
1980 type locally-bound tm -> o.
1981 type get-scope-aux tm -> list tm -> o.
1982 get-scope-aux (con _) [].
1983 get-scope-aux (uva _ L) L1 :-
1984   forall2 get-scope-aux L R,
1985   flatten R L1.
1986 get-scope-aux (lam B) L1 :-
1987   pi x\ locally-bound x => get-scope-aux (B x) L1.
1988 get-scope-aux (app L) L1 :-
1989   forall2 get-scope-aux L R,
1990   flatten R L1.
1991 get-scope-aux X [X] :- name X, not (locally-bound X).
1992 get-scope-aux X [] :- name X, (locally-bound X).
1993
1994 type names1 list tm -> o.
1995 names1 L :-
1996   names L1,
1997   new_int N,
1998   if (1 is N mod 2) (L1 = L) (rev L1 L).
1999
2000 type get-scope tm -> list tm -> o.
2001 get-scope T Scope :-
2002   get-scope-aux T ScopeDuplicata,
2003   undup ScopeDuplicata Scope.
2004 type rigid fm -> o.
2005 rigid X :- not (X = fuva _).
2006
2007 type comp-lam (fm -> fm) -> (tm -> tm) ->
2008   mmap -> mmap -> links -> links -> subst -> subst -> o.
2009 comp-lam F G M1 M2 L1 L3 S1 S2 :-
2010   pi x y\ (pi M L S\ comp x y M M L L S S) => (Hλ)
2011   comp (F x) (G y) M1 M2 L1 (L2 y) S1 S2,
2012   close-links L2 L3.
2013
2014 type close-links (tm -> links) -> links -> o.
2015 close-links (v\ [X |L v]) [X|R] :- !, close-links L R.
2016 close-links (v\ [X v|L v]) [abs X|R] :- close-links L R.
2017 close-links (_\ []) [].
2018 type comp fm -> tm -> mmap -> mmap -> links -> links ->
2019   subst -> subst -> o.
2020 comp (fcon C) (con C) M M L L S S.
2021 comp (flam F) (uva A Scope) M1 M2 L1 L3 S1 S3 :-
2022   maybe-eta (flam F) [], !,
2023   alloc S1 A S2,
2024   comp-lam F F1 M1 M2 L1 L2 S2 S3,
2025   get-scope (lam F1) Scope,
2026   L3 = [val (link-eta (uva A Scope) (lam F1)) | L2].
2027 comp (flam F) (lam F1) M1 M2 L1 L2 S1 S2 :- (cλ)
2028   comp-lam F F1 M1 M2 L1 L2 S1 S2.
2029 comp (fuva A) (uva B []) M1 M2 L L S1 S2 :-
2030
2031   m-alloc (fv A) (hv B (arity z)) M1 M2 S1 S2.
2032 comp (fapp [fuva A|Ag]) (uva B Ag1) M1 M2 L L S1 S2 :-
2033   pattern-fragment Ag, !,
2034   fold6 comp Ag Ag1 M1 M1 L L S1 S1,
2035   len Ag Arity,
2036   m-alloc (fv A) (hv B (arity Arity)) M1 M2 S1 S2.
2037 comp (fapp [fuva A|Ag]) (uva B Scope) M1 M3 L1 L3 S1 S4 :- !,
2038   pattern-fragment-prefix Ag Pf Extra,
2039   len Pf Arity,
2040   alloc S1 B S2,
2041   m-alloc (fv A) (hv C (arity Arity)) M1 M2 S2 S3,
2042   fold6 comp Pf Pf1 M2 M2 L1 L1 S3 S3,
2043   fold6 comp Extra Extra1 M2 M3 L1 L2 S3 S4,
2044   Beta = app [uva C Pf1 | Extra1],
2045   get-scope Beta Scope,
2046   L3 = [val (link-llam (uva B Scope) Beta) | L2].
2047 comp (fapp A) (app A1) M1 M2 L1 L2 S1 S2 :- (c@)
2048   fold6 comp A A1 M1 M2 L1 L2 S1 S2.
2049
2050 type alloc mem A -> addr -> mem A -> o.
2051 alloc S N S1 :- mem.new S N S1.
2052
2053 type compile-terms-diagnostic
2054   triple diagnostic fm fm ->
2055   triple diagnostic tm tm ->
2056   mmap -> mmap ->
2057   links -> links ->
2058   subst -> subst -> o.
2059 compile-terms-diagnostic (triple D F01 F02) (triple D H01 H02) M1 M3 L1
2060   beta-normal F01 F01',
2061   beta-normal F02 F02',
2062   comp F01' H01 M1 M2 L1 L2 S1 S2,
2063   comp F02' H02 M2 M3 L2 L3 S2 S3.
2064
2065 type compile-terms
2066   list (triple diagnostic fm fm) ->
2067   list (triple diagnostic tm tm) ->
2068   mmap -> links -> subst -> o.
2069 compile-terms T H M L S :-
2070   fold6 compile-terms-diagnostic T H [] M_ [] L_ [] S_,
2071   print-compil-result T H L_ M_,
2072   deduplicate-map M_ M S_ S L_ L.
2073
2074 type make-eta-link-aux nat -> addr -> addr ->
2075   list tm -> links -> subst -> subst -> o.
2076 make-eta-link-aux z Ad1 Ad2 Scope1 L H1 H1 :-
2077   rev Scope1 Scope, eta-expand (uva Ad2 Scope) T1,
2078   L = [val (link-eta (uva Ad1 Scope) T1)].
2079 make-eta-link-aux (s N) Ad1 Ad2 Scope1 L H1 H3 :-
2080   rev Scope1 Scope, alloc H1 Ad H2,
2081   eta-expand (uva Ad Scope) T2,
2082   (pi x\ make-eta-link-aux N Ad Ad2 [x|Scope1] (L1 x) H2 H3),
2083   close-links L1 L2,
2084   L = [val (link-eta (uva Ad1 Scope) T2) | L2].
2085
2086 type make-eta-link nat -> nat -> addr -> addr ->
2087   list tm -> links -> subst -> subst -> o.
2088

```

```

2089 make-eta-link (s N) z Ad1 Ad2 Vars L H H1 :-                2147
2090   make-eta-link-aux N Ad2 Ad1 Vars L H H1.                2148
2091 make-eta-link z (s N) Ad1 Ad2 Vars L H H1 :-                2149
2092   make-eta-link-aux N Ad1 Ad2 Vars L H H1.                2150
2093 make-eta-link (s N) (s M) Ad1 Ad2 Vars Links H H1 :-        2151
2094   (pi x\ make-eta-link N M Ad1 Ad2 [x|Vars] (L x) H H1),    2152
2095   close-links L Links.                                     2153
2096                                                         2154
2097 type deduplicate-map mmap -> mmap ->                        2155
2098   subst -> subst -> links -> links -> o.                  2156
2099 deduplicate-map [] [] H H L L.                               2157
2100 deduplicate-map [(fv 0 <-> hv M (arity LenM)) as X1 | Map1] Map2 progress-beta-link-aux T1 T2 S1 S2 [] :- is-in-pf T2, !,
2101   take-list Map1 ((fv 0 <-> hv M' (arity LenM')) _) !,      (T1 ==1 T2) S1 S2.
2102   std.assert! (not (LenM = LenM')) "Deduplicate map, there is a bug"
2103   print "arity-fix links:" {ppmapping X1} "~!~" {ppmapping ((fv 0 <-> hv M' (arity LenM')))},
2104   make-eta-link LenM LenM' M M' [] New H1 H2,
2105   print "new eta link" {pplinks New},
2106   append New L1 L2,
2107   deduplicate-map Map1 Map2 H2 H3 L2 L3.
2108 deduplicate-map [A|As] [A|Bs] H1 H2 L1 L2 :-
2109   deduplicate-map As Bs H1 H2 L1 L2, !.
2110 deduplicate-map [A|_] _ H _ _ :-
2111   halt "deduplicating mapping error" {ppmapping A} {ho.ppsubst H}.
2112
2113 14 THE PROGRESS FUNCTION
2114
2115 macro @one :- s z.
2116
2117 type contract-rigid list ho.tm -> ho.tm -> ho.tm -> o.
2118 contract-rigid L (ho.lam F) T :-
2119   pi x\ contract-rigid [x|L] (F x) T. % also checks H Prefix does not
2120 contract-rigid L (ho.app [H|Args]) T :-
2121   rev L LRev, append Prefix LRev Args,
2122   if (Prefix = []) (T = H) (T = ho.app [H|Prefix]).
2123
2124 type progress-eta-link ho.tm -> ho.tm -> ho.subst -> ho.subst -> link-eta
2125 progress-eta-link (ho.app _ as T) (ho.lam x\ _ as T1) H H1 [] :- !, not (T1 = ho.uva _ _), !, fail.
2126   ({eta-expand T @one} ==1 T1) H H1.
2127 progress-eta-link (ho.con _ as T) (ho.lam x\ _ as T1) H H1 [] :- !
2128   ({eta-expand T @one} ==1 T1) H H1.
2129 progress-eta-link (ho.lam _ as T) T1 H H1 [] :- !,
2130   (T ==1 T1) H H1.
2131 progress-eta-link (ho.uva _ _ as X) T H H1 [] :-
2132   contract-rigid [] T T1, !, (X ==1 T1) H H1.
2133 progress-eta-link (ho.uva Ad _ as T1) T2 H H [eval-link-eta T1 T2] :- !,
2134   if (ho.not_occ Ad H T2) true fail.
2135
2136 type is-in-pf ho.tm -> o.
2137 is-in-pf (ho.app [ho.uva _ _ | _]) :- !, fail.
2138 is-in-pf (ho.lam B) :- !, pi x\ is-in-pf (B x).
2139 is-in-pf (ho.con _).
2140 is-in-pf (ho.app L) :- forall1 is-in-pf L.
2141 is-in-pf N :- name N.
2142 is-in-pf (ho.uva _ L) :- pattern-fragment L.
2143
2144 type arity ho.tm -> nat -> o.
2145 arity (ho.con _) z.

```

```

arity (ho.app L) A :- len L A.
type occur-check-err ho.tm -> ho.tm -> ho.subst -> o.
occur-check-err (ho.con _) _ _ :- !.
occur-check-err (ho.app _) _ _ :- !.
occur-check-err (ho.lam _) _ _ :- !.
occur-check-err (ho.uva Ad _) T S :-
  not (ho.not_occ Ad S T).
type progress-beta-link-aux ho.tm -> ho.tm ->
  ho.subst -> ho.subst -> links -> o.
progress-beta-link-aux T1 T2 S1 S2 [] :- is-in-pf T2, !,
  (T1 ==1 T2) S1 S2.
progress-beta-link-aux T1 T2 S S [eval-link-llam T1 T2] :- !.
type progress-beta-link ho.tm -> ho.tm -> ho.subst ->
  ho.subst -> links -> o.
progress-beta-link T (ho.app [ho.uva V Scope | L] as T2) S S2 [eval-link-
  arity T Arity, len L ArgsNb, ArgsNb >n Arity, !,
  minus ArgsNb Arity Diff, mem.new S V1 S1,
  eta-expand (ho.uva V1 Scope) Diff T1,
  ((ho.uva V Scope) ==1 T1) S1 S2.
progress-beta-link (ho.uva _ _ as T) (ho.app [ho.uva Ad1 Scope1 | L1] as
  append Scope1 L1 Scope1L,
  pattern-fragment-prefix Scope1L Scope2 L2,
  not (Scope1 = Scope2), !,
  mem.new S1 Ad2 S2,
  len Scope1 Scope1Len,
  len Scope2 Scope2Len,
  make-eta-link Scope1Len Scope2Len Ad2 [] LinkEta S2 S3,
  if (L2 = []) (NewLinks = LinkEta, T2 = ho.uva Ad2 Scope2)
  (T2 = ho.app [ho.uva Ad2 Scope2 | L2],
  NewLinks = [eval-link-llam T T2 | LinkEta]).
progress-beta-link T1 (ho.app [ho.uva _ _ | _] as T2) _ _ _ :-
  occur-check-err T T2 S1, !, fail.
progress-beta-link T1 (ho.app [ho.uva _ _ | _] as T2) H H [eval-link-llam
  progress-beta-link T1 (ho.app [Hd | T1]) S1 S2 B :-
  ho.lbeta Hd T1 T3,
  progress-beta-link-aux T1 T3 S1 S2 B.
type solve-link-abs link -> links -> ho.subst -> ho.subst -> o.
solve-link-abs (ho.abs X) R H H1 :-
  pi x\ ho.copy x x => (pi S\ ho.deref S x x) =>
    solve-link-abs (X x) (R' x) H H1,
  close-links R' R.
solve-link-abs (@eval-link-eta A B) NewLinks S S1 :- !,
  progress-eta-link A B S S1 NewLinks.
solve-link-abs (@eval-link-llam A B) NewLinks S S1 :- !,

```

```

2205     progress-beta-link A B S S1 NewLinks.
2206
2207 type take-link link -> links -> link -> links -> o.
2208 take-link A [B|XS] B XS :- link-abs-same-lhs A B, !.
2209 take-link A [L|XS] B [L|YS] :- take-link A XS B YS.
2210
2211 type link-abs-same-lhs link -> link -> o.
2212 link-abs-same-lhs (ho.abs F) B :-
2213   pi x\ link-abs-same-lhs (F x) B.
2214 link-abs-same-lhs A (ho.abs G) :-
2215   pi x\ link-abs-same-lhs A (G x).
2216 link-abs-same-lhs (@val-link-eta (ho.uva N _) _) (@val-link-eta (ho.uva N y) tm)->fm _ x y => tm->fm L (B1 x) (B2 y).
2217
2218 type same-link-eta link -> link -> ho.subst -> ho.subst -> o.
2219 same-link-eta (ho.abs F) B H H1 :- !, pi x\ same-link-eta (F x) B H H1 :- !, pi x\ same-link-eta (F x) B H H1 :- !, pi x\ same-link-eta (F x) B H H1 :- !,
2220 same-link-eta A (ho.abs G) H H1 :- !, pi x\ same-link-eta A (G x) H H1 :- !, pi x\ same-link-eta A (G x) H H1 :- !, pi x\ same-link-eta A (G x) H H1 :- !,
2221 same-link-eta (@val-link-eta (ho.uva N S1) A)
2222   (@val-link-eta (ho.uva N S2) B) H H1 :-
2223   std.map2 S1 S2 (x\y\r\ r = ho.copy x y) Perm,
2224   Perm => ho.copy A A',
2225   (A' ==1 B) H H1.
2226
2227 type progress1 links -> links -> ho.subst -> ho.subst -> o.
2228 progress1 [] [] X X.
2229 progress1 [A|L1] [A|L3] S S2 :- take-link A L1 B L2, !,
2230   same-link-eta A B S S1,
2231   progress1 L2 L3 S1 S2.
2232 progress1 [L0|L1] L3 S S2 :- deref-link S L0 L,
2233   solve-link-abs L R S S1, !,
2234   progress1 L1 L2 S1 S2, append R L2 L3.
2235
2236 15 THE DECOMPILER
2237
2238 type abs->lam ho.assignment -> ho.tm -> o.
2239 abs->lam (ho.abs T) (ho.lam R) :- !, pi x\ abs->lam (T x) (R x).
2240 abs->lam (ho.val A) A.
2241
2242 type commit-links-aux link -> ho.subst -> ho.subst -> o.
2243 commit-links-aux (@val-link-eta T1 T2) H1 H2 :-
2244   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2245   (T1' ==1 T2') H1 H2.
2246 commit-links-aux (@val-link-llam T1 T2) H1 H2 :-
2247   ho.deref H1 T1 T1', ho.deref H1 T2 T2',
2248   (T1' ==1 T2') H1 H2.
2249 commit-links-aux (ho.abs B) H H1 :-
2250   pi x\ commit-links-aux (B x) H H1.
2251
2252 type commit-links links -> links -> ho.subst -> ho.subst -> o.
2253 commit-links [] [] H H.
2254 commit-links [Abs | Links] L H H2 :-
2255   commit-links-aux Abs H H1, !, commit-links Links L H1 H2.
2256
2257 type decomp1-subst map -> map -> ho.subst ->
2258   fo.fsubst -> fo.fsubst -> o.
2259 decomp1-subst _ [A|_] _ _ _ :- fail.
2260 decomp1-subst _ [] _ F F.
2261 decomp1-subst Map [mapping (fv V0) (hv VM _)|T1] H F F2 :-
2262
2263   mem.set? VM H T, !,
2264   ho.deref-assmt H T TTT,
2265   abs->lam TTT T', tm->fm Map T' T1,
2266   fo.eta-contract T1 T2, mem.assign V0 F T2 F1,
2267   decomp1-subst Map T1 H F1 F2.
2268 decomp1-subst Map [mapping _ (hv VM _)|T1] H F F2 :-
2269   mem.unset? VM H, decomp1-subst Map T1 H F F2.
2270
2271 type tm->fm map -> ho.tm -> fo.fm -> o.
2272 tm->fm _ (ho.con C) (fo.fcon C).
2273 tm->fm L (ho.lam B1) (fo.flam B2) :-
2274   pi x\y\ tm->fm _ x y => tm->fm L (B1 x) (B2 y).
2275 tm->fm L (ho.app L1) T :- map (tm->fm L) L1 [Hd|T1],
2276   fo.mk-app Hd T1 T.
2277 tm->fm L (ho.uva VM TL) T :- mem L (mapping (fv V0) (hv VM _)),
2278   H Map (tm->fm L) TL T1, fo.mk-app (fo.fuva V0) T1 T.
2279
2280 type add-new-map-aux ho.subst -> list ho.tm -> map ->
2281   map -> fo.fsubst -> fo.fsubst -> o.
2282 add-new-map-aux _ [] _ [] S S.
2283 add-new-map-aux H [T|Ts] L L2 S S2 :-
2284   add-new-map H T L L1 S S1,
2285   add-new-map-aux H Ts L1 L2 S1 S2.
2286
2287 type add-new-map ho.subst -> ho.tm -> map ->
2288   map -> fo.fsubst -> fo.fsubst -> o.
2289 add-new-map _ (ho.uva N _) Map [] F1 F1 :-
2290   mem Map (mapping _ (hv N _)), !.
2291 add-new-map H (ho.uva N L) Map [Map1 | MapL] F1 F3 :-
2292   mem.new F1 M F2,
2293   len L Arity, Map1 = mapping (fv M) (hv N (arity Arity)),
2294   add-new-map H (ho.app L) [Map1 | Map] MapL F2 F3.
2295 add-new-map H (ho.lam B) Map NewMap F1 F2 :-
2296   pi x\ add-new-map H (B x) Map NewMap F1 F2.
2297 add-new-map H (ho.app L) Map NewMap F1 F3 :-
2298   add-new-map-aux H L Map NewMap F1 F3.
2299 add-new-map _ (ho.con _) _ [] F F :- !.
2300 add-new-map _ N _ [] F F :- name N.
2301
2302 type complete-mapping-under-ass ho.subst -> ho.assignment ->
2303   map -> map -> fo.fsubst -> fo.fsubst -> o.
2304 complete-mapping-under-ass H (ho.val Val) Map1 Map2 F1 F2 :-
2305   add-new-map H Val Map1 Map2 F1 F2.
2306 complete-mapping-under-ass H (ho.abs Abs) Map1 Map2 F1 F2 :-
2307   pi x\ complete-mapping-under-ass H (Abs x) Map1 Map2 F1 F2.
2308
2309 type complete-mapping ho.subst -> ho.subst ->
2310   map -> map -> fo.fsubst -> fo.fsubst -> o.
2311 complete-mapping _ [] L L F F.
2312 complete-mapping H [none | T1] L1 L2 F1 F2 :-
2313   complete-mapping H T1 L1 L2 F1 F2.
2314 complete-mapping H [some T0 | T1] L1 L3 F1 F3 :-
2315   ho.deref-assmt H T0 T,
2316   complete-mapping-under-ass H T L1 L2 F1 F2,
2317   append L1 L2 Lall,
2318   complete-mapping H T1 Lall L3 F2 F3.
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362

```

```

2321 type decompile map -> links -> ho.subst ->
2322     fo.fsubst -> fo.fsubst -> o.
2323 decompile Map1 L H0 F0 F02 :-
2324     commit-links L L1_ H0 H01, !,
2325     complete-mapping H01 H01 Map1 Map2 F0 F01,
2326     decomp1-subst Map2 Map2 H01 F01 F02.

```

16 AUXILIARY FUNCTIONS

```

2329 type fold4 (A -> A1 -> B -> B -> C -> C -> o) -> list A ->
2330     list A1 -> B -> B -> C -> C -> o.
2331 fold4 _ [] [] A A B B.
2332 fold4 F [X|XS] [Y|YS] A A1 B B1 :- F X Y A A0 B B0,
2333     fold4 F XS YS A0 A1 B0 B1.
2334
2335 type len list A -> nat -> o.
2336 len [] z.
2337 len [_|L] (s X) :- len L X.

```

2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436