A Meta-Language for Ornamentation in ML

ANONYMOUS AUTHOR(S)

Ornaments are a way to describe changes in datatype definitions that preserve their recursive structure, reorganizing, adding, or dropping some pieces of data so that functions operating on the bare definition can be partially and sometimes totally lifted into functions operating on the ornamented structure. We propose an extension of ML with higher-order ornaments, demonstrate its expressiveness with a few typical examples, study the metatheoretical properties of ornaments, and show their elaboration process. We introduce a meta-language above ML in which we can elaborate a most generic lifting of bare code, so that ornamented code can then be obtained by instantiation of the generic lifting, followed by staged reduction and some remaining simplifications. We use logical relations to closely relate the ornamented code to the bare code.

CCS Concepts: •**Software and its engineering** → **Functional languages;**

ACM Reference format:
Anonymous Author(s). 2016. A Meta-Language for Ornamentation in ML. 1, 1, Article 1 (January 2016), 60 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Inductive datatypes and parametric polymorphism were two key features introduced in the ML family of languages in the 1980's, at the core of the two popular languages OCaml and Haskell. Datatypes stress the algebraic structure of data while parametric polymorphism allows to exploit universal properties of algorithms working on algebraic structures.

Datatype definitions are inductively defined as labeled sums and products over primitive types. This restricted language allows the programmer to describe, on the one hand, these recursive structures and, on the other hand, how to populate them with data of either primitive types or types given as parameters.

Datatypes can be factorized through their recursive structures. For example, the type of leaf binary trees and the type of node binary trees both share a common binary-branching structure and are isomorphic but functions operating on them must be defined independently. Having established the structural ties between two datatypes, one soon realizes that both admit strikingly similar functions, operating similarly over their common recursive structure. Users sometimes feel they are repeatedly programming the same operations over and over again with only minor variations. The refactoring process by which one adapts existing code to work on another, similarly-structured datatype requires non-negligible efforts from the programmer. Can this process be automated?

The strong typing discipline of ML is already very helpful for code refactoring. When modifying a datatype definition, it points out all the ill-typed occurrences where some rewriting ought to be performed. However, while in most cases the adjustments are really obvious from the context, they still have to be manually performed, one after the other. Furthermore, changes that do not lead to type errors will be left unnoticed.

Our goal is not just that the new program typechecks, but to carefully track all changes in datatype definitions to automate most of this process. Besides, we wish to have some guarantee that the new version behaves consistently with the original program except for the code that is manually added.

The recent theory of ornaments [3, 4] is the right framework to tackle these challenges. It defines conditions under which a new datatype definition can be described as an *ornament* of another. In essence, a datatype ornaments another if they both share the same recursive skeleton.

2016. XXXX-XXXX/2016/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Williams et al. have already explored the interest of such an approach and exploited the structural ties relating a datatype and its ornamented counterpart [16]. In particular, they have demonstrated how functions operating only on the structure of some datatype could be semi-automatically lifted to its ornamented version.

We build on their work, generalizing and especially formalizing their approach. As them, we also consider an ML setting where ornaments are a primitive notion rather than encoded. To be self-contained we remind a few typical uses of ornaments in ML, mostly taken from their work, but also propose new ones.

Our contributions are the following: we extend the definition of ornaments to the higher-order setting and give them a semantics using logical relations, which allows to maintain a close correspondence between the bare code and the lifted code; we propose a principled approach to the lifting process, introducing an intermediate meta-language above ML in which lifted functions have a most general syntactic elaborated form, before they are instantiated into concrete liftings, reduced, and simplified back to ML code. Although designed as a tool, our meta-language is a restricted form of a dependently-typed language that keeps track of selected branches during pattern matching and could perhaps also be useful for other purposes.

The rest of the paper is organized as follows. In the next section, we introduce ornaments by means of examples. The lifting process, which is the core of our contribution is described intuitively in section §3. We introduce the meta-language in §4 and present its meta-theoretical properties in §5. We introduce a logical relation on meta-terms in §6 that serves both for proving the meta-theoretical properties and for the lifting elaboration process. In §7, we show how the meta-construction can be eliminated by meta-reduction. In §8, we give a formal definition of ornaments, based on a logical relation. In §9, we describe the lifting process that transforms a lifting declaration into actual ML code, and we justify its correctness. We discuss a few other issues in §10 and related works in §11.

2 Examples of ornaments

Let us discover ornaments by means of examples. All examples preceded by a green vertical bar have been processed by a prototype implementation[1], which follows some OCaml-like[2] syntax.

Output of the prototype appears with a wider red vertical bar, and only differ from the output of the prototype by α-conversion[3]. The code that appears without a vertical mark is internal intermediate code for sake of explanation and has not been processed.

2.1 Code refactoring

The most striking application is perhaps code refactoring, which is often an annoying but necessary task when programming. We start with an example where refactoring is isomorphic, reorganizing a sum data structure into a sum of sums. Assume given the following datatype representing arithmetic expressions, together with an evaluation function.

```
type expr =
  | Const of int
  | Add of expr * expr
  | Mul of expr * expr
```

```
let rec eval a = match a with
  | Const i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

The programmer may realize that Add and Mul are two binary operators that can be factorized, and thus prefer the following version expr' using an auxiliary type of binary operators (left-hand side). There is a bidirectional

[1]Available at url http://pauillac.inria.fr/~remy/ornaments/.
[2]http://caml.inria.fr/
[3]In the future, the prototype could be instrumented to choose more pertinent names for variable that are introduced by ornamentation.

mapping between these two datatypes, which may be described as an ornament (right-hand side):

```
type binop = Add' | Mul'
type expr' =
  | Const' of int
  | Binop' of binop * expr' * expr'
```

```
type ornament oexpr : expr → expr' with
  | Const i → Const' i
  | Add (u, v) → Binop' (Add', u, v)
  | Mul (u, v) → Binop' (Mul', u, v)
```

The compiler now has enough information to automatically lift the old version of the code to the new version.

```
let eval' = lifting eval : oexpr → _
```

The expression oexpr → _ is an ornament signature, which follows the syntax of types but replacing type constructors by ornaments. (The wildcard is part of the ornament specification that may be inferred; it could have been replaced by int, which is an abstract type and is not ornamented, so we may use int in place of an identity ornament.)ere, the compiler will automatically compile eval' to the expected code, without any further user interaction:

```
let rec eval' a = match a with
  | Const' i → i
  | Binop' (Add', u, v) → add (eval' u) (eval' v)
  | Binop' (Mul', u, v) → mul (eval' u) (eval' v)
```

Not only this is well-typed, but the semantics is also preserved—by construction. Notice that this code refactoring also works in the other direction: we could have instead started with the definition of eval', defined the reverse ornament from expr' to expr, and obtained eval as a lifting of eval'.

Lifting also works with higher-order types and recursive datatype definitions with negative occurrences. For example, we could have extended arithmetic expressions with nodes for abstraction and application:

```
type expr = ... | Abs of (expr → expr) | App of expr * expr
```

and the lifting of expr into expr would also recursively lift the type expr → expr into expr' → expr' (this example is detailed in the extended version).

Although this is a simple example of refactoring where no information is added to the datatype, more general refactoring can often be decomposed into similar isomorphic transformations that do not lose any information, and other transformations as described next that decorate an existing node with new pieces of information.

2.2 Code refinement

As explained in the introduction, many data-structures have the same recursive structure and only differ by the other (non-recursive) information carried by their nodes. For instance, lists can be seen as an ornament of Peano numerals:

```
type nat = Z | S of nat
type 'a list = Nil | Cons of 'a * 'a list
```

```
type ornament 'a natlist : nat → 'a list with
  | Z → Nil
  | S m → Cons (_, m)
```

The ornament is syntactically described as a mapping from the bare type nat to the lifted type 'a list. However, this mapping may be incompletely determined, as is the case here, since we do not know which element to attach to a Cons node coming from a successor node. (The ornament definition may also be read in the reverse direction, which defines a projection from 'a list to nat, the length function! but we do not use this information hereafter.)

The addition on numbers may have been defined as follows (on the left-hand side):

```
let rec add m n = match m with
  | Z → n
  | S m' → S (add m' n)
val add : nat → nat → nat
```

```
let rec append m n = match m with
  | Nil → n
  | Cons (x, m') → Cons(x, append m' n)
val append : 'a list → 'a list → 'a list
```

Observe, the similarity with append, given above on the right-hand side. Having already recognized an ornament between nat and list, we expect append to be definable as a lifting of add:

```
let append0 = lifting add : _ natlist → _ natlist → _ natlist
```

However, this returns an incomplete skeleton:

```
let rec append0 m n = match m with
    | Nil → n
    | Cons (x, m') → Cons (#2, append0 m' n)
```

Indeed, this requires to build a cons node from a successor node, which is underdetermined. This is reported to the user by leaving a labeled hole #2 in the ornamented code. The programmer may use this label to provide a *patch* that will fill the hole in the skeleton. The patch may use all bindings in context, which are the same as the bindings already in context at the same location in the bare version. In particular, the first argument of Cons cannot be obtained directly, but only by pattern matching again on *m*:

```
let append = lifting add : _ natlist → _ natlist → _ natlist
  with #2 ← match m with Cons(x0, _) → x0
```

The lifting is now complete, and produces exactly the code of append given above. The superfluous pattern matching in the patch has been automatically removed: the patch "**match** m **with** Cons(x_0,_) → x_0" has not just been inserted in the hole, but also simplified by observing that x_0 is actually equal to x and need not be extracted again from m. This simplification process relies on the ability of the meta-language to maintain equalities between terms via dependent types, and is needed to make the lifted code as close as possible to manually written code. This is essential, since the lifted code may become the next version of the source code to be read and modified by the programmer. This is a strong argument in favor of the principled approach that we present next and formalize in the rest of the paper.

Although the hole cannot be uniquely determined by ornamentation alone, it is here the obvious choice: since the append function is polymorphic we need an element of the same type as the unnamed argument of Cons, so this is the obvious value to pick—but not the only one, as one could also look further in the tail of the list. Instead of giving an explicit patch, we could give a tactic that would fill in the hole with the "obvious choice" in such cases. However, while important in practice, this is an orthogonal issue related to code inference which is not the focus of this work. Below, we stick to the case where patches are always explicit and we leave holes in the skeleton when patches are missing.

While this example may seem anecdotal, and is chosen here for pedagogical purposes, there is actually a strong relation between recursive data-structures and numerical representations at the heart of several works [8, 12].

2.3 Global compilation optimizations

Interestingly, code refactoring can also be used to enable global compilation optimizations by changing the representation of data structures. For example, one may use sets whose elements are members of a large sum datatype $\tau_I \triangleq \Sigma^{j\in J} A_j \mid \Sigma^{k\in K}(A_k \text{ of } \tau_k)$ where τ_J is the sum $\Sigma^{j\in J} A_j$, say τ_J containing a few constant constructors and τ_K are the remaining cases. One may then chose to split cases into two sum types τ_J and τ_K and use the isomorphism τ_I set $\approx \tau_J$ set $\times \tau_K$ set to enable the optimization of τ_J set, for example by representing all cases as an integer—when $|J|$ is not too large.

2.4 Hiding administrative data

Sometimes data structures need to carry annotations, which are useful information for certain purposes, not at the core of the algorithms. A typical example is location information attached to abstract syntax trees for reporting purposes. The problem with data structure annotations is that they often obfuscate the code. We show how ornaments can be used to keep programming on the bare view of the data structures and lift the code to the ornamented view with annotations. In particular, scanning algorithms can be manually written on the bare structure and automatically lifted to the ornamented structure with only a few patches to describe how locations must be used for error reporting.

Consider for example, the type of λ-expressions and its evaluator:

```
type expr =
  | Abs of (expr → expr)
  | App of expr * expr
  | Const of int
```

```
let rec eval e = match e with
  | App (u, v) →
      (match eval u with Some (Abs f) → Some (f v)
      | _ → None)
  | v → Some (v)
```

To add locations, we instrument the data-structure as follow, which we declare as an ornament of ”expr”:

```
type loc = Location of string * int * int
type expr' =
  | App' of (expr' * loc) * (expr' * loc)
  | Abs' of (expr' * loc → expr' * loc)
  | Const' of int
```

```
type ornament add_loc : expr → expr' * loc with
  | Abs f → (Abs' f, _)
  | App (u, v) → (App' (u, v), _)
  | Const i → (Const' i, _)
```

We define a datatype for results which is an ornament of the option type:

```
type ('a, 'err) result =
  | Ok of 'a
  | Error of 'err
```

```
type ornament ('a, 'err) optres :
        'a option → ('a, 'err) result with
  | Some a → Ok a
  | None → Error _
```

If we try to lift the function as before:

```
let eval_incomplete = lifting eval : add_loc → (int, loc) optres
```

The system will only be able to do a partial lifting

```
let rec eval_incomplete e = match e with
    | (Abs' x, x') → Ok e
    | (App'(x, x'), x'') →
      begin match (* _2 *) eval_incomplete x with
        | Ok (Abs' x, x'') → Ok (x x')
        | Ok (App'(x, x'), x'') → Error #6
        | Ok (Const' x, x') → Error #4
        | Error x → Error #11
      end
    | (Const' x, x') → Ok e
```

Indeed, in the erroneous case eval' must now return a value of the form Error (...) instead of None, but it has no way to know which arguments to pass to the constructor, hence the holes labeled #6, #4, and #11 in the three error cases. Notice that the prototype has exploded the wild pattern _ to consider all possible cases at occurrences that have been scrutinized in another branch, and thus requires patches in three cases. Two of these patches are actually different depending on whether the recursive call to eval_incomplete succeeds.

To complete the lifting, we provide the following patch:

```
let eval_loc = lifting eval : add_loc → (int, 'loc) optres with
  | #4,#6 ← (match e with(_,loc) → loc)
  | #11 ← (match x with Error(e) → e)
```

We then obtain the expected complete code:

```
let rec eval_loc a = match a with
    | (Abs' f, _) → Ok a
    | (App' (u, v), loc) →
      begin match eval_loc u with
        | Ok (Abs' f, _) → Ok (f v)
        | Ok (App'(_, _), _) → Error loc
        | Ok (Const' _, _) → Error loc
        | Error x → Error x
      end
    | (Const' i, _) → Ok a
```

Common branches could actually be refactored using wildcard abbreviations whenever possible, leading to the following code, but this has not been implemented yet:

```
let rec eval_loc a → match a with
  | App' (u, v), loc →
      begin match eval_loc u with
        | Ok (Abs f, loc) → Ok (f v)
        | Ok (_, _) → Error loc
        | Error loc' → Error loc'
      end
  | _ → Ok a
```

While this example is limited to the simple case where we only read the abstract syntax tree, some compilation passes often need to transform the abstract syntax tree carrying location information around. More experiment is needed to see how viable the ornament approach is. This might be a case where appropriate tactics for filling the holes would be quite helpful.

This example suggests a new use of ornaments in a programming environment where the bare code and lifted code would be kept in sync, and the user could switch between the two views, using the bare code for the core of its algorithm that need not see all the details and the lifted code only when necessary.

Other uses of ornaments can also be found in prior works [16].

3 Overview of the lifting process

3.1 Encoding ornaments

Ornamentation only affects datatypes, so a program can be lifted by simply inserting some code to translate from and to the ornamented type at occurrences where the base datatype is either constructed or destructed in the original program.

We now explain how this code can be automatically inserted by lifting. For sake of illustration, we proceed in several incremental steps.

Intuitively, the append function should have the same recursive schema as add, and operate on constructors Nil and Cons similarly to the way add proceeds with constructors S and Z. To make this correspondence explicit, we may see a list as a nat-like structure where just the head of the list has been transformed. For that purpose, we introduce an *hybrid* open version of the datatype of Peano naturals, using new constructors Z' and S' corresponding to Z and S but remaining parameterized over the type of the tail:

```
type 'a nat_skel = Z' | S' of 'a
```

Notice that nat_skel is just the type function of which type nat is the fix-point—up to the renaming of constructors. We may now define the head projection of the list into 'a nat_skel[4] where the head looks like a number while the tail is a list:

```
let proj_nat_list = fun m ⇏ match m with
  | Nil → Z'
  | Cons (_, m') → S' m'
val proj_nat_list : 'a list → 'a list nat_skel
```

We use annotated versions of abstractions **fun** x ⇏ *a* and applications *a*#*b* called *meta-functions* and *meta-applications* to keep track of helper code and distinguish it from the original code, but these can otherwise be read as regular functions and applications.

Once an 'a list has been turned into 'a list nat_skel with this helper function, we can pattern match on 'a list nat_skel in the same way we matched on nat in the definition of add. Hence, the definition of append should look like:

```
let rec append₁ = fun m n → match proj_nat_list # m with
  | Z' → n
  | S' m' → ... S' (append₁ m' n) ...
```

In the second branch, we must return a list out of the hybrid list-nat skeleton S' (append m' n). Using a helper function:

```
  | S' m' → cstr_nat_list ₁ (S'(append m' n)) ...
```

Of course, cstr_nat_list requires some supplementary information x to put in the head cell of the list:

```
let cstr_nat_list = fun n x ⇏ match n with
  | Z' → Nil
  | S' n' → Cons (x, n')
val cstr_nat_list : 'a list nat_skel → 'a → 'a list
```

As explained above, this supplementation information is (**match** m **with** Cons (x, _) → x). and must be user provided as patch #1. Hence, the lifting of add into lists is:

[4]Our naming convention is to use the suffix _nat_list for the functions related to the ornament from nat to list.

```
let rec append₂ = fun m n →
  match proj_nat_list # m with
  | Z' → n
  | S' m' → cstr_nat_list # (S'(append₂ m' n)) # (match m with Cons (x, _) → x)
```

This version is now correct, but not final yet, as it still contains the intermediate hybrid structure, which will eventually be eliminated. However, before we see how to do so in the next section, we first check that our schema extends to more complex examples of ornaments.

Assume, for instance, that we also attach new information to the Z constructor to get lists with some information at the end, which could be defined as:

```
type ('a,'b) listend = Nilend of 'b | Consend of 'a * ('a, 'b) listend
```

We may write encoding and decoding functions as above:

```
let proj_nat_listend = fun l → match l with        let cstr_nat_listend = fun n x → match n with
  | Nilend _ → Z'                                     | Z' → Nilend x
  | Consend (_,l') → S' l'                            | S' l' → Consend (x,l')
```

However, a new problem appears: we cannot give a valid ML type to the function cstr_nat_listend, as the argument x should take different types depending on whether n is zero or a successor. This is solved by adding a form of dependent types to our intermediate language—and finely tuned restrictions to guarantee that the generated code becomes typeable in ML after some simplifications. This is the purpose of the next section.

3.2 Eliminating the encoding

The mechanical ornamentation both creates intermediate hybrid data structures and includes extra abstractions and applications. Fortunately, these additional computations can be avoided, which not only removes sources of inefficiencies, but also helps generating more natural code with fewer indirections that looks similar to hand-written code.

We first perform meta-reduction of append_2 which removes all helper functions (we actually give different types to ordinary and meta functions so that meta-functions can only be applied using meta-applications and ordinary functions can only be applied using ordinary applications):

```
let rec append₃ = fun m n →
 match (match m with | Nil → Z' | Cons (x, m') → S' m') with
  | Z' → n
  | S' m' → b
```

where b is (graying the dead branch):

```
match S'(append₃ m' n) with
  | Z' –> Nil
  | S' r' → Cons ((match m with Cons(x, _) → x), r')
```

Still, append_3 computes two extra pattern matchings that do not appear in the manually written version append. Interestingly, both of them can still be eliminated. Extruding the inner match on m in append_3, we get:

```
let rec append₄ = fun m n →
  match m with
  | Nil → (match Z' with Z' → n | S' m' –> b)
  | Cons (x, m') → (match S' m' with Z' –> n | S' m' → b)
```

Since we learn that m is equal to Cons(x,m') in the Cons branch, the expression *b* simplifies to Cons(x, append m' n). After removing all dead branches and useless pattern matching, we obtain the manually-written version append:

```
let rec append = fun m n →
  match m with
    | Nil → n
    | Cons (a, m') → Cons (a, append m' n)
```

3.3 Inferring a generic lifting

We have shown a specific ornamentation append of add. However, instead of producing such an ornamentation directly, we first generate a generic lifting of add that is abstracted over all possible instantiations and patches, and only then specialize it to some specific ornamentation by passing the encoding and decoding functions as arguments, as well as a set of *patches* describing how to generate the additional data. All liftings that follow the syntactic structure of the original function can be obtained by instantiating the same generic lifting.

Let us consider this process in more details on our running example add, which we remind below.

```
let rec add = fun m n → match m with
  | Z → n
  | S m' → S(add m' n)
```

There are two places where add can be generalized: the pattern matching on m, and the construction S(add m' n) in the successor branch. We do not generalize the base case because we want to preserve the syntactical structure of add. The ornamentation constraints are analyzed using a form of elaborating type inference, by simultaneously inferring ornaments and inserting the corresponding code transformations: we infer that m can be replaced by any ornament nat_ty of naturals, which will be given by a pair of functions m_{proj} and m_{cstr} to destruct nat_ty into a nat_skel and construct a nat_ty from a nat_skel, respectively; we also infer that n and the result must be the same ornament of naturals, hence given by the other pair of functions n_{proj} and n_{cstr}. We thus obtain a description of all possible *syntactic* ornaments of the base function, *i.e.* those ornaments that preserve the structure of the original code:

```
let add_gen = fun m_proj m_cstr n_proj n_cstr p_1 ⇏
  let rec add_gen' = fun m n → match m_proj # m with
    | Z' → n
    | S' m' → n_cstr # S'(add_gen' m' n) # (p_1 # add_gen' # m # m' # n)
  in add_gen'
```

Notice that since m is only destructured and n is only constructed, m_{cstr} and n_{proj} are unused in this example, but we keep them as extra parameters for regularity of the encoding.

Finally, the patch p_1 describes how to obtain the extra information from the environment, namely add_gen, m, n, m', when rebuilding a new value of the ornamented type. While m_{proj}, m_{cstr}, n_{proj}, n_{cstr} parameters will be automatically instantiated, the code for patches will have to be user-provided. The generalized function abstracts over all possible ornaments, and must now be instantiated by some specific ornaments.

For a trivial example, we may decide to ornament nothing, *i.e.* just lift nat to itself using the *identity ornament* on nat, which amounts to passing to add_gen the following trivial functions:

```
let proj_nat_nat = fun x ⇏                    let cstr_nat_nat = fun x () ⇏
  match x with Z → Z' | S x → S' x              match x with Z' → Z | S' x → S x
```

There is no information added, so we may use this unit_patch for p_1 (the information returned will be ignored anyway):

$$\textbf{let}\ \mathsf{unit\_patch} = \textbf{fun}\ \_\ \_\ \_\ \_ \ \#\!\!\Rightarrow ()$$
$$\textbf{let}\ \mathsf{add}_1 = \mathsf{add\_gen}\ \#\ \mathsf{proj\_nat\_nat}\ \#\ \mathsf{cstr\_nat\_nat}\ \#\ \mathsf{proj\_nat\_nat}\ \#\ \mathsf{cstr\_nat\_nat}\ \#\ \mathsf{unit\_patch}$$

As expected, meta-reducing add_1 and simplifying the result returns the original program add.

Returning to the append function, we may instantiate the generic lifting add_gen with the ornament between natural numbers and lists with the following patch:

$$\textbf{let}\ \mathsf{append\_patch} = \textbf{fun}\ \_\ \mathsf{m}\ \_\ \_ \ \#\!\!\Rightarrow \textbf{match}\ \mathsf{m}\ \textbf{with}\ \mathsf{Cons}(\mathsf{x}, \_) \to \mathsf{x}$$
$$\textbf{let}\ \mathsf{append}_5 = \mathsf{add\_gen}\ \#\ \mathsf{proj\_nat\_list}\ \#\ \mathsf{cstr\_nat\_list}\ \#\ \mathsf{proj\_nat\_list}\ \#\ \mathsf{cstr\_nat\_list}\ \#\ \mathsf{append\_patch}$$

Meta-reduction of append_5 gives append_2. which can then be simplified to append, as explained above.

Besides sharing the same generic code for different ornaments of the base type, the generic code also helps relate the original code and the ornamented one: we use a parametricity result to prove that add_1 and append_5 are related by ornamentation. We then show they can be simplified into add and append, respectively, hence respecting the (observational) equivalence on both sides. This in turn proves that add and append are in an ornamentation relation.

The generic lifting is not exposed as is to the user because it is not convenient to use directly. Positional arguments are not practical, because one must reference the generic term to understand the role of each argument. We can solve this problem by attaching the arguments to program locations and exposing the correspondence in the user interface. For example, in the lifting of add to append shown in the previous section, the location #1 corresponds to the argument p_1.

Patches can be automatically inferred in some cases: some patches are trivial such as the unit patch in the lifting of add to itself, and some other patches disappear because they are located in a dead branch.

3.4 Local lifting and ornament specifications

A lifting definition comes with an *ornament signature* which is propagated during the elaboration to choose the appropriate ornaments and liftings of other types appearing in the definition. This process will be described in §9. However, this mechanism is not always sufficient for specifying all ornaments. In particular, it cannot describe the ornament of types of subexpressions used for auxiliary computations that do not appear in the type of the whole expression. In such cases, the elaboration is incomplete or fails and additional information must be provided as additional lifting rules. We may tell the elaboration to choose natlist whenever an unspecified ornament of nat is needed. A common situation is to use the identity ornament by default (if no other rule applies).

During elaboration, required liftings are chosen in the environment of already existing ones. Even when the ornament type is fully determined, a lifting may be required at some type while none or several[5] are available. In such situations, lifting information must also be provided as additional rules.

4 Meta ML

As explained above (§3), we elaborate programs into an extended meta-language *m*ML that extends ML with dependent types and has separate meta-abstractions and meta-applications. We describe *m*ML as an extension of ML in two steps: we first enrich the language with equality constraints in typing judgments, obtaining an intermediate language *e*ML. We then add meta operations to obtain *m*ML. Our design is carefully crafted so that programs that have an *m*ML typing containing only *e*ML types can be meta-reduced to *e*ML (Theorem 5.38). Moreover, under additional conditions, these can be simplified into ML programs (§7). It is also an important

[5]Indeed, there may be two lifting of the same function with the same ornament but different patches.

$$\textsc{EnvE}\ \frac{}{\vdash \emptyset} \qquad \textsc{EnvVar}\ \frac{\vdash \Gamma \quad \Gamma \vdash \tau : \mathsf{Sch} \quad x \# \Gamma}{\vdash \Gamma, x : \tau} \qquad \textsc{EnvTVar}\ \frac{\vdash \Gamma \quad \Gamma \vdash \kappa : \mathsf{wf} \quad \alpha \# \Gamma}{\vdash \Gamma, \alpha : \kappa} \qquad \textsc{K-Var}\ \frac{\alpha : \mathsf{Typ} \in \Gamma}{\Gamma \vdash \alpha : \mathsf{Typ}} \qquad \textsc{K-Base}\ \frac{\zeta : (\mathsf{Typ})^i \to \mathsf{Typ} \quad (\Gamma \vdash \tau_i : \mathsf{Typ})^i}{\Gamma \vdash \zeta\ (\tau_i)_i : \mathsf{Typ}}$$

$$\textsc{K-Arr}\ \frac{\Gamma \vdash \tau_1 : \mathsf{Typ} \quad \Gamma \vdash \tau_2 : \mathsf{Typ}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathsf{Typ}} \qquad \textsc{K-SubTyp}\ \frac{\Gamma \vdash \tau : \mathsf{Typ}}{\Gamma \vdash \tau : \mathsf{Sch}} \qquad \textsc{K-All}\ \frac{\Gamma, \alpha : \kappa \vdash \tau : \mathsf{Sch}}{\Gamma \vdash \forall \alpha : \mathsf{Typ}\ \tau : \mathsf{Sch}}$$

Fig. 1. Kinding rules for ML

$$\begin{aligned}
\kappa &::= \mathsf{Typ} \mid \mathsf{Sch} \\
\tau, \sigma &::= \alpha \mid \tau \to \tau \mid \zeta\ \overline{\tau} \mid \forall(\alpha : \mathsf{Typ})\ \tau \\
\Gamma &::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha : \mathsf{Typ} \\
\zeta &::= \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{list} \mid \ldots
\end{aligned}
\qquad
\begin{aligned}
a, b &::= x \mid \mathsf{let}\ x = a\ \mathsf{in}\ a \mid \mathsf{fix}\ (x : \tau)\ x.\ a \mid a\ a \\
&\quad\mid \Lambda(\alpha : \mathsf{Typ}).\ u \mid a\ \tau \mid d\ \overline{\tau}\ \overline{a} \mid \mathsf{match}\ a\ \mathsf{with}\ \overline{P \to a} \\
P &::= d\ \overline{\tau}\ \overline{x} \\
v &::= d\ \overline{\tau}\ \overline{v} \mid \mathsf{fix}\ (x : \tau)\ x.\ a \\
u &::= x \mid d\ \overline{\tau}\ \overline{u} \mid \mathsf{fix}\ (x : \tau)\ x.\ a \mid u\ \tau \mid \Lambda(\alpha : \kappa).\ u \\
&\quad\mid \mathsf{let}\ x = u\ \mathsf{in}\ u \mid \mathsf{match}\ u\ \mathsf{with}\ \overline{P \to u}
\end{aligned}$$

Fig. 2. Syntax of ML

$$\textsc{Var}\ \frac{\vdash \Gamma \quad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textsc{TAbs}\ \frac{\Gamma, \alpha : \mathsf{Typ} \vdash u : \sigma}{\Gamma \vdash \Lambda(\alpha : \mathsf{Typ}).\ u : \forall(\alpha : \mathsf{Typ})\ \sigma} \qquad \textsc{TApp}\ \frac{\Gamma \vdash \tau : \mathsf{Typ} \quad \Gamma \vdash a : \forall(\alpha : \mathsf{Typ})\ \sigma}{\Gamma \vdash a\ \tau : \sigma[\alpha \leftarrow \tau]}$$

$$\textsc{Fix}\ \frac{\Gamma, x : \tau_1 \to \tau_2, y : \tau_1 \vdash a : \tau_2}{\Gamma \vdash \mathsf{fix}\ (x : \tau_1 \to \tau_2)\ y.\ a : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash b : \tau_1 \quad \Gamma \vdash a : \tau_1 \to \tau_2}{\Gamma \vdash a\ b : \tau_2}$$

$$\textsc{Let-Mono}\ \frac{\Gamma \vdash \tau' : \mathsf{Typ} \quad \Gamma \vdash a : \tau' \quad \Gamma, x : \tau', x =_{\tau'} a \vdash b : \tau}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau} \qquad \textsc{Let-Poly}\ \frac{\Gamma \vdash \sigma : \mathsf{Sch} \quad \Gamma \vdash u : \sigma \quad \Gamma, x : \sigma, x =_{\sigma} u \vdash b : \tau}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ b : \tau}$$

$$\textsc{Con}\ \frac{\vdash d : \forall(\alpha_j : \mathsf{Typ})^j\ (\tau_i)^i \to \tau \quad (\Gamma \vdash \tau_j : \mathsf{Typ})^j \quad (\Gamma \vdash a_i : \tau_i[\alpha_j \leftarrow \tau_j]^j)^i}{\Gamma \vdash d(\tau_j)^j (a_i)^i : \tau[\alpha_j \leftarrow \tau_j]^j}$$

$$\textsc{Match}\ \frac{\begin{array}{c}\Gamma \vdash \tau : \mathsf{Sch} \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \\ \Gamma \vdash a : \zeta\ (\tau_k)^k \quad (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, a =_{\zeta\ (\tau_k)^k} d_i(\tau_{ij})^k (x_{ij})^j \vdash b_i : \tau)^i\end{array}}{\Gamma \vdash \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_{ik})^k (x_{ij})^j \to b_i)^i : \tau}$$

Fig. 3. Typing rules of ML (and *e*ML in gray)

aspect of our design that *m*ML is only used as an intermediate to implement the generic lifting and the elaboration process and that lifted programs eventually falls back in the ML subset.

$$E \;::=\; [] \mid E\ a \mid v\ E \mid d(v, ..\, v, E, a, ..\, a) \mid \Lambda(\alpha : \mathsf{Typ}).\ E \mid E\ \tau \mid \mathsf{match}\ E\ \mathsf{with}\ \overline{P \to a} \mid \mathsf{let}\ x = E\ \mathsf{in}\ a$$

$$\begin{array}{rcl}
(\mathsf{fix}\ (x : \tau)\ y.\ a)\ v & \longrightarrow^h_\beta & a[x \leftarrow \mathsf{fix}\ (x : \tau)\ y.\ a, y \leftarrow v] \\
(\Lambda(\alpha : \mathsf{Typ}).\ v)\ \tau & \longrightarrow^h_\beta & v[\alpha \leftarrow \tau] \\
\mathsf{let}\ x = v\ \mathsf{in}\ a & \longrightarrow^h_\beta & a[x \leftarrow v] \\
\mathsf{match}\ d_j\ \overline{\tau_j}\ (v_i)^i\ \mathsf{with}\ (d_j\ \overline{\tau_j}\ (x_{ji})^i \to a_j)^j & \longrightarrow^h_\beta & a_j[x_{ij} \leftarrow v_i]^i
\end{array}$$

$$\textsc{Context-Beta}\quad \frac{a \longrightarrow^h_\beta b}{E[a] \longrightarrow_\beta E[b]}$$

Fig. 4. Reduction rules of ML

Notation

We write $(Q_i)^{i \in I}$ for a tuple $(Q_1, ..\, Q_n)$. We often omit the set I in which i ranges and just write $(Q_i)^i$, using different indices i, j, and k for ranging over different sets I, J, and K; We also write $\overline{Q}$ if we do not have to explicitly mention the components Q_i. In particular, $\overline{Q}$ stands for $(Q, ..\, Q)$ in syntax definitions. We write $Q[z_i \leftarrow Q_i]^{i \in I}$, or $Q[z_i \leftarrow Q_i]^i$ for short, for the simultaneous substitution of z_i by Q_i for all i in I.

4.1 ML

We consider an explicitly typed version of ML. In practice, the user writes programs with implicit types that are elaborated into the explicit language, but we leave out type inference here for sake of simplicity[6]. The programmer's language is core ML with recursion and datatypes. Its syntax is described in Figure 2, ignoring the gray which is not part of the ML definition. To prepare for extensions, we slightly depart from traditional presentations. Instead of defining type schemes as a generalization of monomorphic types, we do the converse and introduce monotypes as a restriction of type schemes. The reason to do so is to be able to see both ML and *e*ML as sublanguages of *m*ML—the most expressive of the three. We use kinds to distinguish between the types of the different languages: for ML we only need a kind Typ, to classify the monomorphic types, and its superkind Sch, to classify type schemes. Still, type schemes are not first-class, since polymorphic type variables range only over monomorphic types, *i.e.* those of kind Typ.

We assume given a set of type constructors, written ζ. Each type constructor has a fixed signature of the form $(\mathsf{Typ}, ..\, \mathsf{Typ}) \Rightarrow \mathsf{Typ}$. We require that type expressions respect the kinds of type constructors and type constructors are always fully applied.

The grammar of types is given on the left-hand side of Figure 2. Well formedness of types and type schemes are asserted by judgments $\Gamma \vdash \tau : \mathsf{Typ}$ and $\Gamma \vdash \tau : \mathsf{Sch}$, defined on Figure 6.

We assume given a set of data constructors. Each data constructor d comes with a type signature, which is a closed type-scheme of the form $\forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\ (\alpha_i)^i$. For technical reasons, we assume that all datatypes contain at least one value (note that function types always contain as a value a function that takes an argument and never terminates). This assumption could be relaxed, at the cost of a more complex presentation. Pattern matching is restricted to complete, shallow patterns. Instead of having special notation for recursive functions, functions are always defined recursively, using the construction $\mathsf{fix}\ (f : \tau_1 \to \tau_2)\ x.\ a$. This avoids having two different syntactic forms for values of function type. We still use the standard notation $\lambda(x : \tau_1).\ a$ for non-recursive functions, but we just see it as a shorthand for $\mathsf{fix}\ (f : \tau_1 \to \tau_2)\ x.\ a$ where f does not appear free in a and τ_2 is the function return type.

The language is equipped with a weak left-to-right, call-by-value small-step reduction semantics. The evaluation contexts E and the reduction rules are given in Figure 4. This reduction is written $\longrightarrow_\beta$, and the corresponding

[6]Notice that the issue of type inference is orthogonal, since the generic lifting is obtained by instrumentation of type inference.

$$\begin{array}{rcl}
\mathsf{let}\ x = u\ \mathsf{in}\ b & \longrightarrow^h_\iota & b[x \leftarrow u] \\
(\Lambda(\alpha : \mathsf{Typ}).\, u)\ \tau & \longrightarrow^h_\iota & u[\alpha \leftarrow \tau] \\
\mathsf{match}\ d_j\,\overline{\tau_j}\,(u_i)^i\ \mathsf{with}\ (d_j\,\overline{\tau_j}\,(x_{ji})^i \to \tau_j)^{j\in J} & \longrightarrow^h_\iota & \tau_j[x_{ji} \leftarrow u_i]^i \\
\mathsf{match}\ d_j\,\overline{\tau_j}\,(u_i)^i\ \mathsf{with}\ (d_j\,\overline{\tau_j}\,(x_{ji})^i \to a_j)^{j\in J} & \longrightarrow^h_\iota & a_j[x_{ji} \leftarrow u_i]^i
\end{array}
\qquad
\textsc{Context-Iota}\ \frac{a \longrightarrow^h_\iota b}{C[a] \longrightarrow_\iota C[b]}$$

Fig. 5. New reduction rules of *e*ML

head-reduction is written $\longrightarrow^h_\beta$. Reduction must proceed under *type* abstractions, so that we have a type-erasing semantics.

Typing environments Γ contain term variables $x : \tau$ and type variables α : Typ. Well-formedness rules for types and environments are given in figures 1 and 3. We use the convention that type environments do not map the same variable twice. We write $z \# \Gamma$ to mean that z is fresh for Γ, *i.e.* it is neither in the domain nor in the image of Γ. Kinding rules are straightforward. Rule K-SubTyp says that any type of the kind Typ, *i.e.* a simple type, can also be considered as a type of the kind Sch, *i.e.* a type scheme. The typing rules are just the explicitly typed version of the ML typing rules. Typing judgments are of the form $\Gamma \vdash a : \tau$ where $\Gamma \vdash \tau$: Sch. Although we do not have references, we still have a form of value restriction: Rule Let-Poly restricts polymorphic binding to a class of *non-expansive terms u*, defined on Figure 2, that extends values with type abstraction, application, pattern matching, and binding of non-expansive terms—whose reduction always terminate. Binding of an expansive term is still allowed (and is heavily used in the elaboration), but its typing is monomorphic as described by Rule Let-Mono.

4.2 Adding term equalities

The intermediate language *e*ML extends ML with *term equalities* and *type-level matches*. Type-level matches may be reduced using term equalities accumulated along pattern matching branches. We describe the syntax and semantics of *e*ML below, but do not discuss its metatheory, as it is a sublanguage of *m*ML, whose meta-theoretical properties will be studied in the following sections.

The syntax of *e*ML terms is the same as that of ML terms, except for the syntax of types, which now includes a pattern matching construct that matches on values and returns types. The new kinding and typing rules are given on Figure 6. We classify type pattern matching in Sch to prevent it from appearing deep inside types. Typing contexts are extended with type equalities, which will be accumulated along pattern matching branches:

$$\tau\ ::=\ \ldots \mid \mathsf{match}\ a\ \mathsf{with}\ \overline{P \to \tau} \qquad\qquad \Gamma\ ::=\ \ldots \mid \Gamma, a =_\tau b$$

A let binding introduces an equality in the typing context witnessing that the new variable is equal to its definition, while we are typechecking the body (rules Let-eML-Mono and Let-eML-Poly); similarly, both type-level and term-level pattern matching introduce equalities witnessing the branch under selection (rules K-Match and Match-eML). Type-level pattern matching is not introduced by syntax-directed typing rules. Instead, it is implicitly introduced through the conversion rule Conv. It allows replacing one type with another in a typing judgment as long as the types can be proved equal, as expressed by an equality judgment $\Gamma \vdash \tau_1 \simeq \tau_2$ defined on Figure 7.

We define the judgment generically, as equality on kinds and terms will intervene later: we use the metavariable X to stand for either a term or a type (and later a kind), and correspondingly, Y stands for respectively a type, a kind (and later the sort of well-formed kinds). Equality is an equivalence relation (C-Refl, C-Sym, C-Trans) on well-typed terms and well-kinded types. Rule C-Red-Iota allows the elimination of type-level matches through the reduction $\longrightarrow_\iota$, defined on Figure 5, but also term-level matches, let bindings, type abstraction and type application. Since it is used for equality proofs rather than computation, and in possibly open terms, it is not

$$\textsc{Conv}\quad \frac{\Gamma \vdash \tau_1 \simeq \tau_2 \qquad \Gamma \vdash a : \tau_1}{\Gamma \vdash a : \tau_2} \qquad\qquad \textsc{EnvEq}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau : \mathsf{Sch} \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash b : \tau}{\vdash \Gamma, a =_\tau b}$$

$$\textsc{K-Match}\quad \frac{\begin{array}{c}\Gamma \vdash a : \zeta\ (\tau_k)^k \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, a =_{\zeta\ (\tau_k)^k} d_i(\tau_{ik})^k (x_{ij})^j \vdash \tau'_i : \mathsf{Sch})^i\end{array}}{\Gamma \vdash \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_{ik})^k (x_{ij})^j \to \tau'_i)^i : \mathsf{Sch}}$$

$$\textsc{Let-eML-Mono}\quad \frac{\begin{array}{c}\Gamma \vdash \tau : \mathsf{Typ} \qquad \Gamma \vdash a : \tau \\ \Gamma, x : \tau, x =_\tau a \vdash b : \tau'\end{array}}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau'}$$

$$\textsc{Let-eML-Poly}\quad \frac{\begin{array}{c}\Gamma \vdash \tau : \mathsf{Sch} \qquad \Gamma \vdash u : \tau \\ \Gamma, x : \tau, x =_\tau u \vdash b : \tau'\end{array}}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ b : \tau'}$$

$$\textsc{Match-eML}\quad \frac{\begin{array}{c}\Gamma \vdash \tau : \mathsf{Sch} \qquad \Gamma \vdash a : \zeta\ (\tau_k)^k \qquad (d_i : \forall(\alpha_k : \mathsf{Typ})^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \\ (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, a =_{\zeta\ (\tau_k)^k} d_i(\tau_{ij})^k (x_{ij})^j \vdash b_i : \tau)^i\end{array}}{\Gamma \vdash \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_{ij})^k (x_{ij})^j \to b_i)^i : \tau}$$

Fig. 6. New typing rules for *e*ML

$$\textsc{C-Refl}\quad \frac{\Gamma \vdash X : Y}{\Gamma \vdash X \simeq X} \qquad \textsc{C-Sym}\quad \frac{\Gamma \vdash X_1 \simeq X_2}{\Gamma \vdash X_2 \simeq X_1} \qquad \textsc{C-Trans}\quad \frac{\Gamma \vdash X_1 \simeq X_2 \qquad \Gamma \vdash X_2 \simeq X_3}{\Gamma \vdash X_1 \simeq X_3}$$

$$\textsc{C-Context}\quad \frac{\Gamma \vdash C[\Gamma' \vdash X_1 : Y'] : Y \qquad \Gamma \vdash X_1 \simeq X_2}{\Gamma \vdash C[X_1] \simeq C[X_2]}$$

$$\textsc{C-Red-Iota}\quad \frac{X_1 \longrightarrow_\iota X_2 \qquad \Gamma \vdash X_1 : Y_1}{\Gamma \vdash X_1 \simeq X_2} \qquad\qquad \textsc{C-Eq}\quad \frac{(u_1 =_\tau u_2) \in \Gamma'}{\Gamma \vdash u_1 \simeq u_2}$$

$$\textsc{C-Split}\quad \frac{\Gamma \vdash u : \zeta\ (\alpha_k)^k \qquad (d_i : \forall(\alpha_k)^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \qquad (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, u = d_i(\tau_{ij})^j (x_{ij})^j \vdash X_1 \simeq X_2)^i}{\Gamma \vdash X_1 \simeq X_2}$$

Fig. 7. Equality judgment for *e*ML

restricted to evaluation contexts but can be performed in an arbitrary context C and uses a call-by-non-expansive term strategy. It does not include reduction of term abstractions, so as to be terminating. The equalities introduced in the context are used through the rule C-Eq. This rule is limited to equalities between non-expansive terms . Conversely, C-Split allows case-splitting on a non-expansive term of a datatype, checking the equality in each branch under the additional equality learned from the branch selection.

Finally, we allow a strong form of congruence (C-Context): if two terms can be proved equal, they can be substituted in any context. The rule is stated using a general *context typing*: we note $\Gamma \vdash C[\Gamma' \vdash X : Y'] : Y$ if there is a derivation of $\Gamma \vdash C[X] : Y$ such that the subderivation concerning X is $\Gamma' \vdash X : Y'$. The context Γ' will hold all equalities and definitions in the branch leading up to X. This means that, when proving an equivalence under a branch, we can use the equalities introduced by this branch. Moreover, when C is a term contexts C, we may write C to mean that C expects a non-expansive term and C expects any term.

Rule C-Context could have been replaced by one congruence rule for each syntactic construct of the language, but this would have been more verbose, and would require adding new equality rules when we extend *e*ML to *m*ML. Rule C-Context enhances the power of the equality. In particular, it allows case splitting on a variable bound by an abstraction. For instance, we can show that terms $\lambda(x : \mathsf{bool}).\ x$ and $\lambda(x : \mathsf{bool}).\ \mathsf{match}\ x\ \mathsf{with}\ \mathsf{True} \to \mathsf{True} \mid \mathsf{False} \to \mathsf{False}$ are equal, by reasoning under the context $\lambda(x : \mathsf{bool}).\ []$ and case-splitting on x. This allows expressing a number of program transformations, among which let extrusion and expansion, eta-expansion, *etc.*

$$
\begin{array}{rcl}
\kappa & ::= & \dots \mid \mathsf{Met} \mid \tau \to \kappa \mid \forall(\alpha : \kappa)\,\kappa \\
\tau, \sigma & ::= & \dots \mid \forall^\sharp(\alpha : \kappa).\,\tau \mid \Pi(x : \tau).\,\tau \mid \Pi(\diamond : a =_\tau a).\,\tau \mid \Lambda^\sharp(\alpha : \kappa).\,\tau \mid \tau \sharp \tau \mid \lambda^\sharp(x : \tau).\,\tau \mid \tau \sharp a \\
a, b & ::= & \dots \mid \lambda^\sharp(x : \tau).\,a \mid a \sharp u \mid \Lambda^\sharp(\alpha : \kappa).\,a \mid a \sharp \tau \mid \lambda^\sharp(\diamond : a =_\tau a).\,a \mid a \sharp \diamond \\
u & ::= & \dots \mid \lambda^\sharp(x : \tau).\,a \mid \Lambda^\sharp(\alpha : \kappa).\,a \mid \lambda^\sharp(\diamond : a =_\tau a).\,a
\end{array}
$$

Fig. 8. Syntax of *m*ML

as equalities. This help with the ornamentation: the pre- and post-processing on the terms will often preserve equality (for example in §7), and thus many other useful properties (for example, they can to be put in the same contexts, and are interchangeable for the logical relation we define in §6).

Under an incoherent context, we can prove equality between any two types: if the environment contains incoherent equalities like $d_1\,\overline{\tau_1}\,\overline{a_1} = d_2\,\overline{\tau_2}\,\overline{a_2}$, we can prove equality of any two types σ_1 and σ_2 as follows: consider the two types σ'_i equal to match $d_i\,\overline{\tau_i}\,\overline{a_i}$ with $d_1\,\overline{\tau_1}\,\overline{a_1} \to \sigma_1 \mid d_2\,\overline{\tau_2}\,\overline{a_2} \to \sigma_2$. By C-Context and C-Eq, they are equal. But one reduces to σ_1 and the other to σ_2. Thus, the code in provably unreachable branches need not be well typed. When writing *e*ML programs, such branches can be simply ignored, for example by replacing their content with () or any other expression. This contrasts with ML, where one needs to add a term that fails at runtime, such as assert false.

Restricting equalities to be used only between non-expansive terms is necessary to get *subject reduction* in *e*ML: since reduction of beta-redexes is disable when testing for equality, we only allow using equalities between terms that will never be affected by reduction of beta-redexes. Consider for example the following term:

```
match (λ(x : unit). True) () with
  | True →   match (λ(x : unit). True) () with True → () | False → 1 + True
  | False → ()
```

If we allowed the use of equalities between expansive terms, it would correctly typecheck: we could prove from the equalities $(\lambda(x : \mathsf{unit}).\,\mathsf{True})\ () = \mathsf{True}$ and $(\lambda(x : \mathsf{unit}).\,\mathsf{True})\ () = \mathsf{False}$ that the branch containing $1 + \mathsf{True}$ is absurd. But, after one step, the first occurrence of $(\lambda(x : \mathsf{unit}).\,\mathsf{True})\ ()$ reduces to True, and we can no longer prove the incoherence without reducing the application in the second occurrence. Thus we need to forbid equalities between terms containing application in a position where it may be evaluated.

We only limit equalities at the usage point. In *m*ML, this allows putting some equalities in the context, even if it is not known at introduction time that they will reduce (by meta-reduction) to non-expansive terms. The code that uses the equality must still be aware of the non-expansiveness of the terms.

We also restrict case-splitting to non-expansive terms. Since they terminate, this greatly simplifies the metatheory of *e*ML.

Forbidding the reduction of application in $\longrightarrow_\iota$ makes $\longrightarrow_\iota$ terminate (see Lemma 5.40). This allows the transformation from *e*ML to ML to proceed easily: in fact, the transformation can be adapted into a typechecking algorithm for *e*ML.

Note that full reduction would be unsound in *e*ML: under an incoherent context, it is possible to type expressions such as True True, *i.e.* progress would not hold if this were in an evaluation context. However, full reduction is not part of the dynamic semantics of *e*ML, but only used in its static semantics to reason about equality. It is then unsurprising—and harmless that progress does not hold under an incoherent context.

4.3 Adding meta-abstractions

The language *m*ML is *e*ML extended with meta-abstractions and meta-applications, with two goals in mind: first, we need to abstract over all the elements that appear in a context so that they can be passed to patches; second,

$$\begin{array}{rcl} (\lambda^\sharp(x:\tau).\,a)\,\sharp\,u & \longrightarrow^h_\sharp & a[x \leftarrow u] \\ (\Lambda^\sharp(\alpha:\kappa).\,a)\,\sharp\,\tau & \longrightarrow^h_\sharp & a[\alpha \leftarrow \tau] \\ (\lambda^\sharp(\diamond : b_1 =_\tau b_2).\,a)\,\sharp\,\diamond & \longrightarrow^h_\sharp & a \end{array} \qquad \begin{array}{rcl} (\lambda^\sharp(x:\tau').\,\tau)\,\sharp\,u & \longrightarrow^h_\sharp & \tau[x \leftarrow u] \\ (\Lambda^\sharp(\alpha:\kappa).\,\tau)\,\sharp\,\tau' & \longrightarrow^h_\sharp & \tau[\alpha \leftarrow \tau'] \end{array} \qquad \frac{a \longrightarrow^h_\sharp b}{C[a] \longrightarrow_\sharp C[b]}\ \textsc{Context-Meta}$$

Fig. 9. The $\longrightarrow_\sharp$ reduction for *m*ML

$$\textsc{S-Type}\ \ \Gamma \vdash \mathsf{Typ} : \mathsf{wf} \qquad \textsc{S-Scheme}\ \ \Gamma \vdash \mathsf{Sch} : \mathsf{wf} \qquad \textsc{S-Meta}\ \ \Gamma \vdash \mathsf{Met} : \mathsf{wf}$$

$$\textsc{S-VArr}\ \frac{\Gamma \vdash \tau : \mathsf{Met} \quad \Gamma \vdash \kappa : \mathsf{wf}}{\Gamma \vdash \tau \rightarrow^{\ell} \kappa : \mathsf{wf}} \qquad \textsc{S-TArr}\ \frac{\Gamma \vdash \kappa_1 : \mathsf{wf} \quad \Gamma, \alpha : \kappa_1 \vdash \kappa_2 : \mathsf{wf}}{\Gamma \vdash \forall(\alpha : \kappa_1)\ \kappa_2 : \mathsf{wf}}$$

Fig. 10. Well-formedness rules for *m*ML

$$\textsc{K-Conv}\ \frac{\Gamma \vdash \tau_1 : \kappa \quad \Gamma \vdash \kappa \simeq \kappa'}{\Gamma \vdash \tau_1 : \kappa'} \qquad \textsc{K-SubEqu}\ \frac{\Gamma \vdash \tau : \mathsf{Sch}}{\Gamma \vdash \tau : \mathsf{Met}} \qquad \textsc{K-Pi}\ \frac{\Gamma \vdash \tau_1 : \mathsf{Met} \quad \Gamma, x^{\ell} : \tau_1 \vdash \tau_2 : \mathsf{Met}}{\Gamma \vdash \Pi(x^{\ell} : \tau_1).\,\tau_2 : \mathsf{Met}} \qquad \textsc{K-Forall-Meta}\ \frac{\Gamma, \alpha : \kappa \vdash \tau : \mathsf{Met} \quad \Gamma \vdash \kappa : \mathsf{wf}}{\Gamma \vdash \forall^\sharp(\alpha : \kappa).\,\tau : \mathsf{Met}}$$

$$\textsc{K-Pi-Eq}\ \frac{\Gamma \vdash a : \tau' \quad \Gamma \vdash b : \tau' \quad \Gamma \vdash \tau' : \mathsf{Sch} \quad \Gamma, (a =_{\tau'} b) \vdash \tau : \mathsf{Met}}{\Gamma \vdash \Pi(\diamond : a =_{\tau'} b).\,\tau : \mathsf{Met}} \qquad \textsc{K-TLam}\ \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \Lambda^\sharp(\alpha : \kappa_1).\,\tau : \forall(\alpha : \kappa_1)\ \kappa_2}$$

$$\textsc{K-TApp}\ \frac{\Gamma \vdash \tau_1 : \forall(\alpha : \kappa_a)\ \kappa_b \quad \Gamma \vdash \tau_2 : \kappa_a}{\Gamma \vdash \tau_1 \,\sharp\, \tau_2 : \kappa_b[\alpha \leftarrow \tau_2]} \qquad \textsc{K-VLam}\ \frac{\Gamma \vdash \tau_1 : \mathsf{Met} \quad \Gamma, x : \tau_1 \vdash \tau_2 : \kappa_2}{\Gamma \vdash \lambda^\sharp(x : \tau_1).\,\tau_2 : \tau_1 \rightarrow \kappa_2} \qquad \textsc{K-VApp}\ \frac{\Gamma \vdash \tau_1 : \tau_2 \rightarrow \kappa_2 \quad \Gamma \vdash a : \tau_2}{\Gamma \vdash \tau_1 \,\sharp\, a : \kappa_2}$$

Fig. 11. Kinding rules for *m*ML

$$\textsc{TAbs-Meta}\ \frac{\Gamma, \alpha : \kappa \vdash a : \tau}{\Gamma \vdash \Lambda^\sharp(\alpha : \kappa).\,a : \forall^\sharp(\alpha : \kappa).\,\tau} \qquad \textsc{TApp-Meta}\ \frac{\Gamma \vdash a : \forall^\sharp(\alpha : \kappa).\,\tau_1 \quad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash a \,\sharp\, \tau_2 : \tau_1[\alpha \leftarrow \tau_2]} \qquad \textsc{EApp}\ \frac{\Gamma \vdash a_1 \simeq a_2 \quad \Gamma \vdash b : \Pi(\diamond : a_1 =_{\tau'} a_2).\,\tau}{\Gamma \vdash b \,\sharp\, \diamond : \tau}$$

$$\textsc{EAbs}\ \frac{\begin{array}{c}\Gamma \vdash \tau : \mathsf{Sch} \quad \Gamma \vdash a_1 : \tau \\ \Gamma \vdash a_2 : \tau \quad \Gamma, (a_1 =_\tau a_2) \vdash b : \tau'\end{array}}{\Gamma \vdash \lambda^\sharp(\diamond : a_1 =_\tau a_2).\,b : \Pi(\diamond : a_1 =_\tau a_2).\,\tau'} \qquad \textsc{Abs-Meta}\ \frac{\Gamma \vdash \tau_1 : \mathsf{Met} \quad \Gamma, x : \tau_1 \vdash a : \tau_2}{\Gamma \vdash \lambda^\sharp(x : \tau_1).\,a : \Pi(x : \tau_1).\,\tau_2} \qquad \textsc{App-Meta}\ \frac{\Gamma \vdash a : \Pi(x : \tau_1).\,\tau_2 \quad \Gamma \vdash u : \tau_1}{\Gamma \vdash a \,\sharp\, u : \tau_2[x \leftarrow u]}$$

$$\textsc{C-Eq}\ \frac{(a_1 =_\tau a_2) \in \Gamma \quad a_1 \longrightarrow^*_\sharp u_1 \quad a_2 \longrightarrow^*_\sharp u_2}{\Gamma \vdash u_1 \simeq u_2} \qquad \textsc{C-Red-Meta}\ \frac{X_1 \longrightarrow^*_\sharp X_2 \quad \Gamma \vdash X_1 : Y}{\Gamma \vdash X_1 \simeq X_2}$$

Fig. 12. Typing and equality rules for *m*ML

we need a form of stratification so that a well-typed *m*ML term whose type and typing context are in *e*ML can always be reduced to a term that can be typed in *e*ML, *i.e.* without any meta-operations. The program can still be read and understood as if *e*ML and *m*ML reduction were interleaved, *i.e.* as if the encoding and decodings of ornaments were called at runtime, but may all happen at ornamentation time.

The syntax of *m*ML is described in Figure 8. We only describe the differences with *e*ML. Terms are extended with meta-abstractions and the corresponding meta-applications on non-expansive terms, types, and equalities, while types are extended with meta-abstractions and meta-applications on non-expansive terms and types. Both meta abstractions and meta applications are marked with $\sharp$ to distinguish them from ML abstractions and applications.

The restriction of meta-applications to the non-expansive subset of terms is to ensure that non-expansive terms are closed under meta-reductionas both the value restriction in ML and the treatment of equalities in *e*ML rely on the stability of non-expansive terms by substitution. A non-expansive term should remain expansive after substitution. Therefore, we may only allow substitution by non-expansive terms. In particular, arguments of redexes in Figure 9 must be non-expansive. To ensure that meta-redexes can still always be reduced before other redexes, we restrict all arguments of meta-applications in the grammar of *m*ML to be non-expansive. To allow some *higher-order meta-programming* (as simple as taking ornament encoding and decoding functions as parameters), we add meta-abstractions, but not meta-applications, to the class of non-expansive terms u. The reason is that we want non-expansive terms to be stable by reduction, but the reduction of a meta-redexes could reveal an ML redex. A simple way to forbid meta-redexes in non-expansive terms is to forbid meta-application.

Equalities are unnamed in environments, but we use the notation $\diamond$ to witness the presence of an equality in both abstractions $\Pi(\diamond : a =_\tau a).\, \tau$ and $\lambda^\sharp(\diamond : a =_\tau a).\, \tau$ and applications $\tau \sharp \diamond$.

The meta-reduction, written $\longrightarrow_\sharp$, is defined on Figure 9. It is a strong reduction, allowed in arbitrary contexts C. The corresponding head-reduction is written $\longrightarrow^h_\sharp$.

The introduction and elimination rules for the new term-level abstractions are given on Figure 12. The new kinding rules for type-level abstraction and application are given on Figure 11. We introduce a kind Met, superkind of Sch (Rule K-SubEqu), to classify the types of meta abstractions. This enforces a phase distinction where meta constructions cannot be bound or returned by *e*ML code. The grammar of kinds is complex enough to warrant its own *sorting* judgment, noted $\Gamma \vdash \kappa : \mathsf{wf}$ and defined on Figure 10.

We must revisit equality. Kinds can now contain types, that can be converted using Rule K-Conv. The equality judgment is enriched with closure by meta-reduction (Rule C-Red-Meta). To prevent meta-reduction from blocking equalities, Rule C-Eq is extended to consider equalities up to meta-reduction. The stratification ensures that a type-level pattern matching cannot return a meta type. This prevents conversion from affecting the meta part of a type. Thus, the meta-reduction of well-typed program does not get stuck, even under arbitrary contexts—in particular under incoherent branches.

5 The metatheory of *m*ML

In this section we present the results on the metatheory of *m*ML that we later use to prove the correctness of our encoding of ornaments.

We write $\longrightarrow$ for the union of $\longrightarrow_\beta$, $\longrightarrow_\iota$ and $\longrightarrow_\sharp$ and $\longrightarrow^*$ its transitive closure. The calculus is confluent.

Theorem 5.1 (Confluence). *Any combination of the reduction relations $\longrightarrow_\iota$, $\longrightarrow_\beta$, $\longrightarrow_\sharp$ is confluent.*

Below we show that meta-reduction can always be performed first—hence at compilation time.

5.1 A temporary definition of equality

The rules for equality given previously omit some hypotheses that are useful when subject reduction is not yet proved. To guarantee that both sides of an equality are well-typed, we need to replace C-Red-Meta with C-Red-Meta', C-Red-Iota with C-Red-Iota' and C-Context with rule C-Context', given on Figure 13. Admissibility of C-Context will be a consequence of Lemma 5.17, and admissibility of C-Red-Meta and C-Red-Iota will be consequences of subject reduction. Since the original rules are less constrained, they are also complete with respect to the rules used in this section. Thus, once the proofs are done we will be able to use the original version.

$$\textsc{C-Context'}\quad \frac{\Gamma \vdash C[\Gamma' \vdash X_1 : Y'] : Y \qquad \Gamma \vdash C[\Gamma' \vdash X_2 : Y'] : Y \qquad \Gamma \vdash X_1 \simeq X_2}{\Gamma \vdash C[X_1] \simeq C[X_2]}$$

$$\textsc{C-Red-Iota'}\quad \frac{X_1 \longrightarrow_\iota X_2 \qquad \Gamma \vdash X_1 : Y_1 \qquad \Gamma \vdash X_2 : Y_2}{\Gamma \vdash X_1 \simeq X_2} \qquad \textsc{C-Red-Meta'}\quad \frac{X_1 \longrightarrow_\sharp^* X_2 \qquad \Gamma \vdash X_1 : Y_1 \qquad \Gamma \vdash X_2 : Y_2}{\Gamma \vdash X_1 \simeq X_2}$$

Fig. 13. Stricter rules for equality

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\langle\!\langle \mathsf{Typ} \rangle\!\rangle & = & \{\mathcal{N}_\mathrm{a}\} & \langle\!\langle \forall(\alpha : \kappa_1)\ \kappa_2 \rangle\!\rangle & = & \langle\!\langle \kappa_1 \rangle\!\rangle \to \langle\!\langle \kappa_2 \rangle\!\rangle \\
\langle\!\langle \mathsf{Sch} \rangle\!\rangle & = & \{\mathcal{N}_\mathrm{a}\} & \langle\!\langle \tau \to^\ell \kappa \rangle\!\rangle & = & \mathbf{1} \to \langle\!\langle \kappa \rangle\!\rangle \\
\langle\!\langle \mathsf{Met} \rangle\!\rangle & = & \mathcal{C}_\mathrm{a} & \langle\!\langle (a_1 =_\tau a_2) \to \kappa \rangle\!\rangle & = & \mathbf{1} \to \langle\!\langle \kappa \rangle\!\rangle
\end{array}$$

Fig. 14. Interpretation of kinds as sets of interpretations

5.2 Strong normalization for $\longrightarrow_\sharp$

Our goal in this section is to prove that meta-reduction and type reduction are strongly normalizing. The notations used in this proof are only used here, and will be re-used for other purposes later in this article.

Theorem 5.2 (Strong normalization for meta-reduction). *The reduction $\longrightarrow_\sharp$ is strongly normalizing.*

As usual, the proof uses *reducibility sets.*

Definition 5.3 (Reducibility set). A set $\mathcal{S}$ of terms is called a *reducibility set* if it respects the properties C1-3 below. We write $\mathcal{C}_\mathrm{a}$ the set of reducibility sets of terms.

C1 every term $a \in \mathcal{S}$ is strongly normalizing;
C2 if $a \in \mathcal{S}$ and $a \longrightarrow_\sharp a'$ then $a' \in \mathcal{S}$;
C3 if a is not a meta-abstraction, and for all a' such that $a \longrightarrow_\sharp a'$, $a' \in \mathcal{S}$ then $a \in \mathcal{S}$.

Similarly, replacing terms with types and kinds, we obtain a version of the properties C1-3 for sets of types and sets of kinds. A set of types or kinds is called a reducibility set if it respects those properties, and we write $\mathcal{C}_\mathrm{t}$ the set of reducbility sets of types, and $\mathcal{C}_\mathrm{k}$ the set of reducibility sets of kinds.

Let $\mathcal{N}_\mathrm{a}$ be the set of all strongly normalizing terms, $\mathcal{N}_\mathrm{t}$ the set of all strongly normalizing types, and $\mathcal{N}_\mathrm{k}$ the set of all strongly normalizing kinds.

Lemma 5.4. *$\mathcal{N}_\mathrm{a}$, $\mathcal{N}_\mathrm{t}$, and $\mathcal{N}_\mathrm{k}$ are reducibility sets.*

Proof. The properties C1-3 are immediate from the definition. □

Definition 5.5 (Interpretation of types and kinds). We define an interpretation $\langle\!\langle \kappa \rangle\!\rangle$ of kinds as sets of possible interpretations of types, with $\mathbf{1}$ the set with one element $\bullet$. The interpretation is given on Figure 14

On Figure 15, we also define an interpretation $[\![\kappa]\!]_\rho$ of kinds as sets of types and an interpretation $[\![\tau]\!]_\rho$ of a type τ under an assignment ρ of reducibility sets to type variables by mutual induction

Definition 5.6 (Type context). We will write $\rho \vDash \Gamma$ if for all $(\alpha : \kappa) \in \Gamma$, $\rho(\alpha) \in \langle\!\langle \kappa \rangle\!\rangle$.

Lemma 5.7 (Equal kinds have the same interpretation). *If $\Gamma \vdash \kappa_1 \simeq \kappa_2$, then $\langle\!\langle \kappa_1 \rangle\!\rangle = \langle\!\langle \kappa_2 \rangle\!\rangle$.*

Proof. By induction on the derivation of the judgment $\Gamma \vdash \kappa_1 \simeq \kappa_2$. We can assume that all reductions in the rules C-Red-Iota' and C-Red-Meta' are head reductions (otherwise we simply need to compose with C-Context.

$$\begin{array}{rcl}
[\![\mathsf{Typ}]\!]_\rho \;=\; [\![\mathsf{Sch}]\!]_\rho & = & [\![\mathsf{Met}]\!]_\rho \;=\; \mathcal{N}_t \\
[\![\forall(\alpha : \kappa_1)\, \kappa_2]\!]_\rho & = & \{\tau \in \mathcal{N}_t \mid \forall \tau' \in [\![\kappa_1]\!]_\rho, \tau \sharp \tau' \in [\![\kappa_2]\!]_{\rho[\alpha \leftarrow [\![\tau']\!]_\rho]}\} \\
[\![\tau \to \kappa]\!]_\rho & = & \{\tau \in \mathcal{N}_t \mid \forall u \in [\![\tau]\!]_\rho, \tau \sharp u \in [\![\kappa]\!]_\rho\} \\
[\![(a_1 =_\tau a_2) \to \kappa]\!]_\rho & = & \{\tau \in \mathcal{N}_t \mid \tau \sharp \diamond \in [\![\kappa]\!]_\rho\} \\
\\
[\![\alpha]\!]_\rho & = & \rho(\alpha) \\
[\![\tau_1 \to \tau_2]\!]_\rho \;=\; [\![\zeta\, \overline{\tau}]\!]_\rho & = & [\![\mathsf{match}\ a\ \mathsf{with}\ \ldots]\!]_\rho \;=\; [\![\forall(\alpha : \mathsf{Typ})\ \ldots]\!]_\rho \;=\; \mathcal{N}_a \\
[\![\Pi(x : \tau_1).\, \tau_2]\!]_\rho & = & \{a \in \mathcal{N}_a \mid \forall u \in [\![\tau_1]\!]_\rho, a \sharp u \in [\![\tau_2]\!]_\rho\} \\
[\![\Pi(\diamond : a_1 =_\tau a_2).\, \tau']\!]_\rho & = & \{a \in \mathcal{N}_a \mid a \sharp \diamond \in [\![\tau']\!]_\rho\} \\
[\![\forall^\sharp(\alpha : \kappa).\, \tau]\!]_\rho & = & \{a \in \mathcal{N}_a \mid \forall \tau' \in [\![\kappa]\!]_\rho, \forall S \in \langle\!\langle \kappa \rangle\!\rangle, a \sharp \tau' \in [\![\tau]\!]_{\rho[\alpha \leftarrow S]}\} \\
[\![\lambda^\sharp(x : \tau').\, \tau]\!]_\rho & = & [\![\lambda^\sharp(\diamond : a_1 =_{\tau'} a_2).\, \tau]\!]_\rho \;=\; \lambda \bullet .\, [\![\tau]\!]_\rho \\
[\![\Lambda^\sharp(\alpha : \kappa).\, \tau]\!]_\rho & = & \lambda S_\alpha \in \langle\!\langle \kappa \rangle\!\rangle.\, [\![\tau]\!]_{\rho[\alpha \leftarrow S_\alpha]} \\
[\![\tau_1 \sharp \tau_2]\!]_\rho & = & [\![\tau_1]\!]_\rho\ [\![\tau_2]\!]_\rho \\
[\![\tau \sharp u]\!]_\rho & = & [\![\tau \sharp \diamond]\!]_\rho \;=\; [\![\tau]\!]_\rho\ \bullet
\end{array}$$

Fig. 15. Interpretation of kinds and types as sets of types and terms

- Reflexivity, symmetry and transitivity translate trivially to equalities.
- There is no head-reduction on kind, and the rule C-Eq does not apply either.
- For C-Split, use the fact that every datatype is inhabited, and conclude from applying the induction hypothesis to any of the cases.
- For C-Context, proceed by induction on the context. If the context is empty, use the induction hypothesis. Otherwise, note that the interpretation of a kind only depends on the interpretation of its (direct) subkinds, and the interpretation of the direct subkinds are equal either because they are identical, or by induction on the context. □

Lemma 5.8 (Interpretation of kinds and types). *Assume $\rho \vDash \Gamma$. Then:*

- *If $\Gamma \vdash \kappa : \mathsf{wf}$, then $[\![\kappa]\!]_\rho$ is well-defined, and $[\![\kappa]\!]_\rho \in \mathcal{C}_t$.*
- *If $\Gamma \vdash \tau : \kappa$, then $[\![\tau]\!]_\rho$ is well-defined, and we have $[\![\tau]\!]_\rho \in \langle\!\langle \kappa \rangle\!\rangle$.*

Proof. By simultaneaous induction on the sorting and kinding derivations. The case of all syntax-directed rules whose output is interpreted as $\mathcal{N}_a$ or $\mathcal{N}_t$ is follows by Lemma 5.4. The variable rule K-Var is handled by using the definition of $\rho \vDash \Gamma$. For abstraction, abstract, add the interpretation to the context and interpret. For application, use the type of $[\![\kappa]\!]_\rho$ for functions. For K-Conv, use Lemma 5.7 to deduce that the interpretation of the kinds are the same. For the subkinding rules (K-SubTyp, K-SubSch, K-SubEqu), use the fact that $\langle\!\langle \mathsf{Typ} \rangle\!\rangle = \langle\!\langle \mathsf{Sch} \rangle\!\rangle = \langle\!\langle \mathsf{Sch} \rangle\!\rangle \subseteq \langle\!\langle \mathsf{Met} \rangle\!\rangle$.

The rules S-VArr, S-TArr, S-EArr, K-Forall, K-Pi, K-Pi-Eq are similar. We only give the proof for K-Pi: Assume S_1 and S_2 are reducibility sets. We will prove C1-3 for $S = \{a \in \mathcal{N}_a \mid \forall u \in S_1, a \sharp u \in S_2\}$.

C1 S is a subset of $\mathcal{N}_a$.

C2 Consider $a \in S$ and a' such that $a \longrightarrow_\sharp a'$. For a given $u \in S_1$, $a \sharp u \in S_2 \longrightarrow_\sharp a' \sharp u$. Thus, $a' \sharp u \in S_2$ by C2 for S_2. Then, $a' \in S$.

C3 Consider a, not an abstraction, such that if $a \longrightarrow_\sharp a'$, $a' \in S$. For $u \in S_1$, we'll prove $a \sharp u \in S_2$. Since a is not an abstraction, $a \sharp u$ reduces either to $a' \sharp u$ with $a \longrightarrow_\sharp a'$, or $a \sharp u'$ with $u \longrightarrow_\sharp u'$. In the first case, $a' \in S$ by hypothesis and $u \in S_1$, so $a' \sharp u \in S_2$. In the second case, $u' \in S_1$ by C2, so $a \sharp u' \in S_2$. By C3 for S_2, because $a \sharp u$ is not an abstraction, $a \sharp u \in S_2$. □

We need the following substitution lemma:

Lemma 5.9 (Substitution).
For all τ, κ, τ', α, and ρ, we have both $[\![\tau]\!]_{\rho[\alpha\leftarrow[\![\tau']\!]_\rho]} = [\![\tau[\alpha \leftarrow \tau']]\!]_\rho$ and $[\![\kappa]\!]_{\rho[\alpha\leftarrow[\![\tau']\!]_\rho]} = [\![\kappa[\alpha \leftarrow \tau']]\!]_\rho$.

Proof. By induction on types and kinds. □

We then need to prove that conversion is sound with respect to the relation. We start by proving soundness of reduction:

Lemma 5.10 (Soundness of reduction). *Let $\longrightarrow$ stand for $\longrightarrow_\iota \cup \longrightarrow_\sharp$. Assume that $\rho \vDash \Gamma$, and τ, τ', κ, κ' are well-kinded (or well-formed) in Γ. Then $[\![\tau]\!]_\rho = [\![\tau']\!]_\rho$ whenever $\tau \longrightarrow \tau'$ and $[\![\kappa]\!]_\rho = [\![\kappa']\!]_\rho$ whenever $\kappa \longrightarrow \kappa'$.*

Proof. By induction on the contexts. The only interesting context is the hole []. Consider the different kinds of head-reduction on types (the induction hypothesis is not concerned with terms, and there is no head-reduction on kinds).

- The cases of all meta-reductions are similar. Consider $(\Lambda^\sharp(\alpha : \kappa).\ \tau) \sharp\, \tau' \longrightarrow^h_\sharp \tau[\alpha \leftarrow \tau']$. The interpretation of the left-hand side is $(\lambda S_\alpha \in \langle\!\langle \kappa \rangle\!\rangle.\ [\![\tau]\!]_{\rho[\alpha\leftarrow S_\alpha]})\ [\![\tau']\!]_\rho = [\![\tau]\!]_{\rho[\alpha\leftarrow[\![\tau']\!]_\rho]}$ and the interpretation of the right-hand side is $[\![\tau[\alpha \leftarrow \tau']]\!]_\rho = [\![\tau]\!]_{\rho[\alpha\leftarrow[\![\tau']\!]_\rho]}$ by substitution (Lemma 5.9).
- In the case of match-reduction, the arguments of the meta-reduction have kind Sch, thus by Lemma 5.8, their interpretation is $\mathcal{N}_\mathrm{a}$. □

Lemma 5.11 (Soundness of conversion). *If $\rho \vDash \Gamma$ and $\Gamma \vdash \tau_1 \simeq \tau_2$, then $[\![\tau_1]\!]_\rho = [\![\tau_2]\!]_\rho$. If $\Gamma \vdash \kappa_1 \simeq \kappa_2$, then $[\![\kappa_1]\!]_\rho = [\![\kappa_2]\!]_\rho$*

Proof. By induction on the equality judgment.

- The rules C-Refl, C-Sym and C-Trans respect the property (by reflexivity, symmetry, transitivity of equality).
- The equalities are not used, thus C-Split does not affect the interpretation (just consider one of the sub-proofs).
- The rule C-Eq does not apply to types and kinds.
- For reductions (C-Red-Iota', C-Red-Meta'), use the previous lemma.
- For C-Context', proceed by induction on the context. The interpretation of a type/kind depends only on the interpretation of its subterms. □

Now we can prove the fundamental lemma:

Lemma 5.12 (Fundamental lemma). *We say $\rho, \gamma \vDash \Gamma$ if $\rho \vDash \Gamma$ and for all $(x, \tau) \in \Gamma$, $\gamma(x) \in [\![\tau]\!]_\rho$. Suppose $\rho, \gamma \vDash \Gamma$. Then:*

- *If $\Gamma \vdash \kappa : \mathrm{wf}$, then $\gamma(\kappa) \in \mathcal{N}_\mathrm{k}$.*
- *If $\Gamma \vdash \tau : \kappa$, then $\gamma(\tau) \in [\![\kappa]\!]_\rho$.*
- *If $\Gamma \vdash a : \tau$, then $\gamma(a) \in [\![\tau]\!]_\rho$.*

Proof. By mutual induction on typing, kinding, and well-formedness derivations. We will examine a few representative rules:

- If the last rule is a conversion, use soundness of conversion.
- If the last rule is App. Assume $\rho, \gamma \vDash \Gamma$, and consider two terms a and b such that $\gamma(a) \in [\![\tau_1 \to \tau_2]\!]_\rho$ and $\gamma(b) \in [\![\tau_1]\!]_\rho$. We need to show: $\gamma(a\ b) = \gamma(a)\ \gamma(b) \in [\![\tau_2]\!]_\rho = \mathcal{N}_\mathrm{a}$ (because τ_2 is necessarily of kind Typ). Consider then a' and b' normal forms of $\gamma(a)$ and $\gamma(b)$. $a'\ b'$ is a normal form of $\gamma(a\ b)$. Thus, $\gamma(a\ b) \in \mathcal{N}_\mathrm{a}$.

- If the last rule is a meta application App-Meta

$$\frac{\Gamma \vdash a : \Pi(x : \tau_1).\, \tau_2 \qquad \Gamma \vdash b : \tau_1}{\Gamma \vdash a \sharp b : \tau_2[x \leftarrow b]}$$

  Consider $\rho, \gamma \vDash \Gamma$. Then, by induction hypothesis, we have: $\gamma(a) \in [\![\Pi(x : \tau_1).\, \tau_2]\!]_\rho$, and $\gamma(b) \in [\![\tau_1]\!]_\rho$. We thus have: $\gamma(a \sharp b) = \gamma(a) \sharp \gamma(b) \in [\![\tau_2]\!]_\rho$. But the interpretation of types does not depend on terms: $[\![\tau_2]\!]_\rho = [\![\tau_2[x \leftarrow b]]\!]_\rho$. It follows that $\gamma(a \sharp b) \in [\![\tau_2[x \leftarrow b]]\!]_\rho$.
- If the last rule is a meta abstraction Abs-Meta:

$$\frac{\Gamma, x : \tau_1 \vdash a : \tau_2}{\Gamma \vdash \lambda^\sharp(x : \tau_1).\, a : \Pi(x : \tau_1).\, \tau_2}$$

  Consider $\rho, \gamma \vDash \Gamma$. Consider $b \in [\![\tau_1]\!]_\rho$. We need to prove that $\gamma(\lambda^\sharp(x : \tau_1).\, a) \sharp b = (\lambda^\sharp(x : \tau_1).\, \gamma(a)) \sharp b \in [\![\tau_2]\!]_\rho)$. Let us use C3: consider all possible reductions. We will proceed by induction on the reduction of $\gamma(A) = A'$, with the hypothesis $\forall\, (B \in [\![\tau_2]\!]_\rho)\ A'[x \leftarrow B]$ (true for $\gamma(A)$ and conserved by reduction), and on the reduction of B ($B \in [\![\tau_1]\!]_\rho$ is conserved by reduction).
  - We can only reduce the type τ_1' a finite number of types by induction hypothesis. It is discarded after reduction of the head redex.
  - If $A' \longrightarrow_\sharp A''$, $(\lambda^\sharp(x : \tau_1).\, a') \sharp b \longrightarrow_\sharp (\lambda^\sharp(x : \tau_1).\, a'') \sharp b$, and we continue by induction.
  - If $B \longrightarrow_\sharp B'$, $(\lambda^\sharp(x : \tau_1).\, a') \sharp b \longrightarrow_\sharp (\lambda^\sharp(x : \tau_1).\, a') \sharp b'$, and we continue by induction.
  - If we reduce the head redex, $(\lambda^\sharp(x : \tau_1).\, a') \sharp b \longrightarrow_\sharp a'[x \leftarrow b]$. But by hypothesis, $a'[x \leftarrow b] \in [\![\tau_2]\!]_\rho$. □

We can now prove the main result of this section:

Proof. [Proof of Theorem 5.2] Consider a kind, type, or term X that is well-typed in a context Γ. We can take the identity substitution $\gamma(x) = x$ for all $x \in \Gamma$ and apply the fundamental lemma. All interpretations are subsets of $\mathcal{N}_a$, thus $X \in \mathcal{N}_a$. □

5.3 Contexts, substitution and weakening

We define a *weakening* judgment $\Gamma_1 \rhd \Gamma_2$ for typing environments that also includes conversion on the types and kinds in the environment.

WEnv-Empty
$$\frac{}{\emptyset \rhd \emptyset}$$

WEnv-Weaken-Var
$$\frac{\Gamma_1 \rhd \Gamma_2}{\Gamma_1 \rhd \Gamma_2, x^\ell : \tau}$$

WEnv-Conv-Var
$$\frac{\Gamma_1 \rhd \Gamma_2 \qquad \Gamma_2 \vdash \tau_1 \simeq \tau_2}{\Gamma_1, x^\ell : \tau_1 \rhd \Gamma_2, x^\ell : \tau_2}$$

WEnv-Weaken-TVar
$$\frac{\Gamma_1 \rhd \Gamma_2}{\Gamma_1 \rhd \Gamma_2, \alpha^\ell : \kappa}$$

WEnv-Conv-TVar
$$\frac{\Gamma_1 \rhd \Gamma_2 \qquad \Gamma_2 \vdash \kappa_1 \simeq \kappa_2}{\Gamma_1, \alpha : \kappa_1 \rhd \Gamma_2, \alpha : \kappa_2}$$

WEnv-Weaken-Eq
$$\frac{\Gamma_1 \rhd \Gamma_2}{\Gamma_1 \rhd \Gamma_2, (a =_\tau b)}$$

WEnv-Conv-Eq
$$\frac{\Gamma_1 \rhd \Gamma_2 \qquad \Gamma_2 \vdash \tau_1 \simeq \tau_2 \qquad \Gamma_2 \vdash a_1 \simeq a_2 \qquad \Gamma_2 \vdash b_1 \simeq b_2}{\Gamma_1, (a_1 =_{\tau_1} b_1) \rhd \Gamma_2, (a_2 =_{\tau_2} b_2)}$$

Lemma 5.13. *Weakening is reflexive and transitive: for all well-formed environments* Γ_1, Γ_2 *and* Γ_3, *we have:*

- $\Gamma_1 \rhd \Gamma_1$ *and*
- *if* $\Gamma_1 \rhd \Gamma_2$ *and* $\Gamma_2 \rhd \Gamma_3$, *then* $\Gamma_1 \rhd \Gamma_3$.

Proof. Reflexivity is proved by induction on $\vdash \Gamma_1$. Transitivity is proved by induction on the two weakening judgments. □

Lemma 5.14 (Weakening and conversion). *Let Γ_1, Γ_2, Γ_1', Γ_2' be well-formed contexts. Suppose $\Gamma_1 \rhd \Gamma_2$ and $\Gamma_2' \rhd \Gamma_1'$. Then:*

- *If $\Gamma_1 \vdash X : Y$, then $\Gamma_2 \vdash X : Y$.*
- *If $\Gamma_1 \vdash X_1 \simeq X_2$, then $\Gamma_2 \vdash X_1 \simeq X_2$.*
- *If $\Gamma_1 \vdash C[\Gamma_1' \vdash X : Y'] : Y$, then $\Gamma_2 \vdash C[\Gamma_2' \vdash X : Y'] : Y$.*

Proof. Proceed by mutual induction on the typing, kinding, sorting, and equality judgment. All rules grow the context only by adding elements at the end, and the elements added will be the same in both contexts, thus preserving the weakening relation. Then we can use the induction hypothesis on subderivations.

Then, we have to consider the rules that read from the context: they are Var, K-Var and C-Eq. For these rules, proceed by induction on the weakening derivation. Consider the case of Var on a variable x. Most rules do not influence variables. There will be no weakening on x because the term types in the stronger context and variables are supposed distinct. The variable x types in the context by hypothesis, so we cannot reach WEnv-Empty. The renaming case is WEnv-Conv-Var. Suppose $\Gamma_1 = \Gamma_1', x : \tau_1$, $\Gamma_2 = \Gamma_2', x : \tau_2$ and $\Gamma_2 \vdash \tau_1 \simeq \tau_2$. Then, we can obtain a derivation of $\Gamma_2 \vdash x : \tau_1$ by using Var, getting a type τ_2 and converting. □

Lemma 5.15 (Substitution preserves typing).
Suppose $\Gamma \vdash u : \tau$. Then,

- *if $\Gamma, x : \tau, \Gamma' \vdash X : Y$, then $\Gamma, \Gamma'[x \leftarrow u] \vdash X[x \leftarrow u] : Y[x \leftarrow u]$;*
- *if $\Gamma, x : \tau, \Gamma' \vdash X_1 \simeq X_2$, then $\Gamma, \Gamma'[x \leftarrow u] \vdash X_1[x \leftarrow u] \simeq X_2[x \leftarrow u]$.*

Suppose $\Gamma \vdash \tau : \kappa$. Then,

- *if $\Gamma, \alpha : \kappa, \Gamma' \vdash X : Y$, then $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash X[\alpha \leftarrow \tau] : Y[\alpha \leftarrow \tau]$;*
- *if $\Gamma, \alpha : \kappa, \Gamma' \vdash X_1 \simeq X_2$, then $\Gamma, \Gamma'[\alpha \leftarrow \tau] \vdash X_1[\alpha \leftarrow \tau] \simeq X_2[\alpha \leftarrow \tau]$.*

Proof. By mutual induction. We use weakening to grow the context on the typing/kinding judgment of the substituted term/type. □

Lemma 5.16 (Substituting equal terms preserves equality).

- *Assume $\Gamma \vdash \tau_1 \simeq \tau_2$ and $\Gamma, \alpha : \kappa \vdash X : Y$ and $\Gamma \vdash \tau_i : \kappa$. Then, $\Gamma \vdash X[\alpha \leftarrow \tau_1] \simeq X[\alpha \leftarrow \tau_2]$.*
- *Assume $\Gamma \vdash u_1 \simeq u_2$ and $\Gamma, x : \tau \vdash X : Y$ and $\Gamma \vdash u_i : \tau$. Then, $\Gamma \vdash X[x \leftarrow u_1] \simeq X[x \leftarrow u_2]$.*

Lemma 5.17 (Substituting equal terms preserves typing). *Assume $\Gamma \vdash C[\Gamma' \vdash X_1 : Y'] : Y$ and $\Gamma' \vdash X_1 \simeq X_2$. Then, $\Gamma \vdash C[\Gamma' \vdash X_2 : Y'] : Y$.*

Proof. We prove these two results by mutual induction on, respectively, the typing derivation $\Gamma, \alpha : \kappa \vdash X : Y$ and the typing derivation $\Gamma \vdash C[\Gamma' \vdash X_1 : Y'] : Y$.

For the first lemma, for each construct, prove equality of the subterms, and use congruence and transitivity of the equality. Use weakening on the equality if there are introductions. Use the second lemma to get the required typing hypotheses.

For the second lemma, the interesting cases are the dependent rules, where a term or type in term-position in a premise of a rule appears either in the context of another premise, or in type-position in the conclusion. When a term or type appears in the context, we use context conversion. The other type of dependency uses substitution in the result, which is handled by the first lemma (Lemma 5.16) □

Note that these lemmas imply that C-Context is admissible.

In order to prove subject reduction, we will need to prove that restricting the rule C-Eq to non-expansive terms only is enough to preserve types, even when applying the reduction $\longrightarrow_\beta$: this reduction should not affect any equality that is actually used in the typing derivation.

Definition 5.18 (Always expansive term). A term a is said to be always expansive if it does not reduce by $\longrightarrow_\sharp$ to a non-expansive term.

Lemma 5.19 (Always expansive redexes). *Let a be of the form $b_1\ b_2$. Then a is always expansive.*

Proof. Meta-reduction does not change the shape of the term. □

Lemma 5.20 (Useless equalities). *Let a_1 or a_2 be always expansive, and suppose $\Gamma \vdash a_i : \tau$. Then $\Gamma, (a_1 =_\tau a_2) \vdash X : Y$ if and only if $\Gamma \vdash X : Y$.*

Proof. The "only if" direction is a direct consequence of weakening. For the other direction, proceed by induction on the typing derivation. The only interesting rule is C-Eq. But by definition, an equality containing an always expansive term is not usable in equalities. □

Lemma 5.21 (Non-dependent contexts for always expansive terms). *Consider an evaluation context E and an always expansive term a such that $\Gamma \vdash E[\Gamma' \vdash a : \tau] : Y$. Then, if $\Gamma' \vdash a' : \tau$, we also have $\Gamma \vdash E[\Gamma' \vdash a' : \tau] : Y$*

Proof. We prove simultaneously that putting an always expansive term in an evaluation context gives an always expansive term, and that the context is not dependent. The case of the hole is immediate. We will examine the case of Let-Poly, which show the important ideas: consider a non-expansive, and $a_1 = \mathsf{let}\ x = a\ \mathsf{in}\ b$. a_1 is always expansive: any meta-reduction will be to something of the form $\mathsf{let}\ x = a_2\ \mathsf{in}\ b_2$ with $a \longrightarrow_\sharp^* a_2$, but a is always expansive, so a_2 is expansive, thus $\mathsf{let}\ x = a_2\ \mathsf{in}\ b_2$ is expansive too. It also admits the same types: suppose we have a derivation $\Gamma, x : \tau, (x =_\tau a) \vdash b : \tau'$. Then by Lemma 5.20, $\Gamma, x : \tau \vdash b : \tau'$ and thus by weakening $\Gamma, x : \tau, (x =_\tau a') \vdash b : \tau'$. The hypotheses of the rule Let-Poly are preserved, so the conclusion is too: $\mathsf{let}\ x = a\ \mathsf{in}\ b$ and $\mathsf{let}\ x = a'\ \mathsf{in}\ b$ have the same type. □

5.4 Analysis of conversions and subject reduction

To prove subject reduction, we need results allowing us to split a conversion between a compound type or kind into a conversion into this subtypes. The easiest way to do it is to proceed in a stratified way. We extract a subreduction $\longrightarrow_\sharp^t$ of $\longrightarrow_\sharp$ that only contains the reductions on types, and $\longrightarrow_\sharp^a$ that only contains the reductions on terms. Then, we consider the reductions in order: first $\longrightarrow_\sharp^t$, then $\longrightarrow_\sharp^a$, then $\longrightarrow_\beta$ and $\longrightarrow_\iota$.

The following lemma is easily derived from the new definition of equality (it requires subject reduction otherwise):

Lemma 5.22 (Equalities are between well-typed things). *Let Γ be a well-formed context. Suppose $\Gamma \vdash X_1 \simeq X_2$. Then, there exists Y_1, Y_2 such that $\Gamma \vdash X_1 : Y_1$ and $\Gamma \vdash X_2 : Y_2$.*

Proof. By induction on a derivation. This is true for C-Red-Meta', C-Red-Iota', C-Context', C-Eq and C-Refl. For C-Sym and C-Trans, apply the induction hypothesis on the subderivations. □

We define a decomposition of kinds into a *head* and a tuple of *tails*. The meta-variable h stands for a head. The decomposition is unique up to renaming.

$$\mathsf{Typ} \blacktriangleright \mathsf{Typ} \circ () \qquad \mathsf{Sch} \blacktriangleright \mathsf{Sch} \circ () \qquad \mathsf{Met} \blacktriangleright \mathsf{Met} \circ () \qquad \tau \to^\ell \kappa \blacktriangleright \_ \to^\ell_{\mathsf{tk}} \_ \circ (\tau, \kappa)$$
$$(a =_\tau b) \to \kappa \blacktriangleright \_ \to_{\mathsf{ek}} \_ \circ (a, b, \tau, \kappa) \qquad \forall(\alpha : \kappa)\ \kappa' \blacktriangleright \forall(\alpha : \_)\ \_ \circ (\kappa, \kappa')$$

Lemma 5.23 (Analysis of conversions, kind-level). *Suppose $\Gamma \vdash \kappa \simeq \kappa'$. Then, κ' and κ' decompose as $\kappa \blacktriangleright h \circ (X_i)^i$ and $\kappa \blacktriangleright h' \circ (X_i')^i$, with $h = h'$ and $X_i = X_i'$.*

Proof. By induction on a derivation of $\Gamma \vdash \kappa \simeq \kappa'$. We can suppose without loss of generality that all reductions are head-reductions by splitting a reduction into a C-Context and the actual head-reduction.

- For C-Refl, we have $\kappa = \kappa'$. Moreover, all kinds decompose, thus κ has a decomposition $\kappa \blacktriangleright h \circ (X_i)^i$. By hypothesis, $\vdash \Gamma \kappa$. Invert this derivation to find that the X_i are well-kinded or well-sorted. Thus, we can conclude by reflexivity: $\Gamma \vdash X_i \simeq X_i$.
- For C-Sym and C-Trans, apply analysis of conversions to the subbranch(es), then use C-Sym or C-Trans to combine the subderivations on the decompositions.
- For C-Split, apply the lemma in each branch. There exists at least one branch, from where we get equality of the heads. For equality of the tails, apply C-Split to combine the subderivations.
- The rules C-Red-Meta' and C-Red-Iota' do not apply because there is no head-reduction on kinds.
- For rule C-Context, either the context is empty and we can apply the induction hypothesis, or the context is non-empty. In this case, the heads are necessarily equal. We can extract one layer from the context. Then, for the tail where the hole is, we can apply C-Context with the subcontext. For the other tails, by inverting the kinding derivation we can find that they are well sorted. Thus we can apply C-Refl to get equality.

□

We can then prove subject reduction for the type-level meta-reduction.We will use the following inversion lemma:

Lemma 5.24 (Inversion for type-level meta reduction). *Consider an environment* Γ.

- *If* $\Gamma \vdash \lambda^\sharp(x : \tau_1').\ \tau_2 : \tau_1 \rightarrow \kappa$, *then* $\Gamma, x : \tau_1 \vdash \tau_2 : \kappa$.
- *If* $\Gamma \vdash \Lambda^\sharp(\alpha : \kappa_1').\ \tau : \forall(\alpha : \kappa_1)\ \kappa_2$, *then* $\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2$.
- *If* $\Gamma \vdash \lambda^\sharp(\diamond : a_1' =_{\tau'} a_2').\ \tau'' : (\diamond : a_1 =_\tau a_2) \rightarrow \kappa$, *then* $\Gamma, (a_1 =_\tau a_2) \vdash \tau'' : \kappa$.

Proof. We will study the first case, the two other cases are similar. Suppose the last rule is not K-Conv. Then, it is a syntax directed rule, so it must be K-VLam, and we have $\Gamma, x : \tau_1 \vdash \tau_2 : \kappa$. Otherwise, we can collect by induction all applications of K-Conv leading to the application of K-VLam. We obtain (by the previous case) a derivation of $\Gamma, x : \tau_1'' \vdash \tau_2 : \kappa'$, with (combining all conversions using transitivity) an equality $\Gamma \vdash \tau_1 \rightarrow \kappa \simeq \tau_1'' \rightarrow \kappa'$. By the previous lemma (Lemma 5.23), we have $\Gamma \vdash \tau_1 \simeq \tau_1''$ and $\Gamma \vdash \kappa \simeq \kappa'$. □

Lemma 5.25 (Subject reduction, type-level meta reduction). *Suppose* $X \longrightarrow_\sharp^t X'$ *and* $\Gamma \vdash X : Y$. *Then,* $\Gamma \vdash X' : Y$.

Proof. The reduction is a head-reduction $\tau \longrightarrow_\sharp^t \tau'$ in a context C. If we can prove subject reduction for $\tau \longrightarrow_\sharp^t \tau'$, we can conclude by Lemma 5.17, because the reduction implies $\Gamma \vdash \tau \simeq \tau'$.

Let us prove subject reduction for the head-reduction: suppose $\tau \longrightarrow_\sharp^t \tau'$ and $\Gamma \vdash \tau : \kappa$. We want to prove $\Gamma \vdash \tau' : \kappa$.

Consider a derivation of $\Gamma \vdash \tau : \kappa$ whose last rule is not a conversion. It is thus a syntax-directed rule. We will consider as an example the head-reduction from $\tau = (\lambda^\sharp(x : \tau_1).\ \tau_2) \sharp\, u$ to $\tau' = \tau_2[x \leftarrow u]$.

Since the last rule of $\Gamma \vdash (\lambda^\sharp(x : \tau_1).\ \tau_2) \sharp\, u : \kappa$ is syntax-directed, we can invert it and obtain $\Gamma \vdash u : \tau_1$ and $\Gamma \vdash \lambda^\sharp(x : \tau_1).\ \tau_2 : \tau_1 \rightarrow \kappa$, We apply inversion (Lemma 5.24) and obtain $\Gamma, x : \tau_1 \vdash \tau_2 : \kappa$. Finally, by substitution (Lemma 5.15), we obtain $\Gamma \vdash \tau_2[x \leftarrow u] : \kappa$, *i.e.* $\Gamma \vdash \tau' : \kappa$. □

This is sufficient to prove the following lemma:

Lemma 5.26 (Normal derivations, type-level meta reduction). *Suppose* $\Gamma \vdash X_1 \simeq X_2$, *where* X_1 *and* X_2 *are normal terms, types or kinds for* $\longrightarrow_\sharp$. *Then, there exists a derivation of* $\Gamma \vdash^n X_1 \simeq X_2$, *where* $\Gamma \vdash^n X_1 \simeq X_2$ *is a limited version of* $\Gamma \vdash X_1 \simeq X_2$ *where the rule* C-Red-Meta *is limited to* $\longrightarrow_\sharp^a$. *More precisely, this judgment is defined from the following rules:*

$$\textsc{C-Refl}\ \frac{\Gamma \vdash X : Y}{\Gamma \vdash^n X \simeq X} \qquad \textsc{C-Sym}\ \frac{\Gamma \vdash^n X_1 \simeq X_2}{\Gamma \vdash^n X_2 \simeq X_1} \qquad \textsc{C-Trans}\ \frac{\Gamma \vdash^n X_1 \simeq X_2 \qquad \Gamma \vdash^n X_2 \simeq X_3}{\Gamma \vdash^n X_1 \simeq X_3}$$

$$\textsc{C-Context}\ \frac{\Gamma \vdash C[\Gamma' \vdash X_1 : Y'] : Y \qquad \Gamma \vdash^n X_1 \simeq X_2}{\Gamma \vdash^n C[X_1] \simeq C[X_2]} \qquad \textsc{C-Red-Iota'}\ \frac{X_1 \longrightarrow_\iota X_2 \qquad \Gamma \vdash X_1 : Y_1 \qquad \Gamma \vdash X_2 : Y_2}{\Gamma \vdash^n X_1 \simeq X_2}$$

$$\textsc{C-Red-Meta'}\ \frac{X_1 \longrightarrow^a_\sharp X_2 \qquad \Gamma \vdash X_1 : Y_1 \qquad \Gamma \vdash X_2 : Y_2}{\Gamma \vdash^n X_1 \simeq X_2} \qquad \textsc{C-Eq}\ \frac{a_1 \longrightarrow^*_\sharp u_1 \qquad a_2 \longrightarrow^*_\sharp u_2 \qquad (a_1 =_\tau a_2) \in \Gamma}{\Gamma \vdash^n u_1 \simeq u_2}$$

$$\textsc{C-Split}\ \frac{\Gamma \vdash u : \zeta\ (\alpha_k)^k \qquad (d_i : \forall(\alpha_k)^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \qquad (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, u = d_i(\tau_{ij})^j(x_{ij})^j \vdash^n X_1 \simeq X_2)^i}{\Gamma \vdash^n X_1 \simeq X_2}$$

Proof. We prove a stronger result: suppose $\Gamma \vdash X_1 \simeq X_2$, and X_1', X_2' are the $\longrightarrow^t_\sharp$ normal forms of X_1 and X_2. Then, for all context C such that $\Gamma \vdash X_i : Y_i$ and $\Gamma' \vdash C[\Gamma \vdash X_i : Y_i] : Y_i'$, if X_i' are the normal forms of $C[X_i]$, then $\Gamma \vdash^n X_1' \simeq X_2'$. In this proof, we'll say "normal forms" without further qualification for $\longrightarrow^t_\sharp$ normal forms.

We proceed by induction on the derivation. We assume all reductions are head-reductions.

- The property is symmetric, so it is preserved by C-Sym.
- For C-Refl: by subject reduction, if a term is well-typed, its normal form is well-typed too.
- For C-Trans, we get the result by unicity of the normal form.
- For C-Context, we fuse the contexts and use the induction hypothesis.
- For C-Red-Meta', if the reduction is a type-level meta reduction, it becomes a C-Refl on the (well-typed) normal form.
- For the other rules, we follow the same pattern. We first normalize the context to a multi-context, represented by a term with a free variable x (or α): $C[x]$ has a normal form X_c. We also normalize X_1 and X_2 to X_1'' and X_2'', and prove $\Gamma \vdash X_1'' \simeq X_2''$.
  - For C-Eq, we can completely normalize the terms.
  - For C-Red-Meta' and C-Red-Iota', head-reduction and normalization commute: if X_1 head-reduces to X_2, then X_1'' head-reduces to X_2''.

  We now have to prove that the normal form of $C[X_i]$ is $X_c[x \leftarrow X_i]$.
  - It is immediate if the hole is a term: no type-level meta-reduction rule depends on the shape of a term.
  - For type-level iota reduction, we use a typing argument: X_1'' is a type-level match, thus has kind Sch, thus X_2'' has kind Sch by subject reduction. Then, X_2'' cannot be a type-level abstraction, because by Lemma 5.23 the kinds of type-level abstractions are not convertible to Sch.

  Finally, we can conclude by C-Context. □

We prove a decomposition result on meta conversions: types and kinds that start with a *meta head* keep their heads, and their *tails* stay related. In order to prove this, we extend the decoding into a head and tails to types. Note that not all types have a head: applications and variables, for example, have no head yet, but they can gain one after reduction.

$$\forall^\sharp(\alpha:\kappa).\,\tau \blacktriangleright \forall^\sharp(\alpha:\_).\,\_\circ(\kappa,\tau) \qquad \Pi(x:\tau).\,\tau' \blacktriangleright \Pi(x:\_).\,\_\circ(\tau,\tau')$$
$$\Pi(\diamond:b =_\tau b').\,a \blacktriangleright \Pi(\diamond:\_ =_\_ \_).\,\_\circ(b,b',\tau,a) \qquad \forall(\alpha:\mathsf{Typ})\,\tau \blacktriangleright \forall(\alpha:\mathsf{Typ})\,\_\circ(\tau)$$
$$\tau_1 \to \tau_2 \blacktriangleright \_ \to_{\mathsf{tt}} \_\circ(\tau_1,\tau_2) \qquad \zeta\,(\tau_i)^i \blacktriangleright \zeta\,\_\circ(\tau_i)^i$$

The meta-heads are all heads that can not be generated by ML reduction in well-kinded types, *i.e.* all except $\forall(\alpha:\mathsf{Typ})\,\_$, $\_ \to_{\mathsf{tt}} \_$ and $\zeta\,\_$

The head-decomposition of types and kinds is preserved by reduction:

Lemma 5.27 (Reduction preserves head-decomposition). *Consider a type or kind X that decomposes as $X \blacktriangleright h\circ(X_i)^i$. Then, if $X \longrightarrow_\sharp^* X'$, X' decomposes as $X' \blacktriangleright h\circ(X_i')^i$, and for all i, $X_i \longrightarrow_\sharp^* X_i'$.*

Proof. The head never reduces. □

From this we can prove a generic result of separation and projection:

Lemma 5.28 (Equality preserves the head). *Consider a type or kind X that decomposes as $X \blacktriangleright h\circ(X_i)^i$, and X' that decomposes as $X' \blacktriangleright h'\circ(X_j')^j$. Then, if h or h' is a meta head, $\Gamma \vdash X \simeq X'$, $h = h'$ and for all i, $\Gamma \vdash X_i \simeq X_i'$.*

Proof. We can start by $\longrightarrow_\sharp^t$-normalizing both sides. The heads stay the same, and the tails are equivalent. Then consider (using Lemma 5.26) a normal derivation of the result. In the following, we assume X and X' are $\longrightarrow_\sharp^t$-normal and the derivation is normalized.

We then proceed by induction on the size of the derivation $\Gamma \vdash^n X \simeq X'$, proving a strengthened result: suppose X decomposes as $X \blacktriangleright h\circ(X_i)^i$, and either $\Gamma \vdash^n X \simeq X'$ or $\Gamma \vdash^n X' \simeq X$. Then, $X' \blacktriangleright h\circ(X_i')^i$, with $\Gamma \vdash X_i \simeq X_i'$.

- There is no difficulty with the rules C-Refl, C-Sym, C-Trans, C-Split.
- For C-Context on the empty context, we apply the induction hypothesis. Otherwise, the head stays the same, and the equality is applied in one of the tails.
- C-Eq does not apply on types.
- Since both X and X' are types, instances of C-Red-Meta' distribute in the tails. That is also the case for instances of C-Red-Iota' that do not reduce the head directly.
- For $\longrightarrow_\iota^h$: X has a head, so the reduction is necessarily $X' \longrightarrow_\iota^h X$. X and X' cannot be kinds, so they are types τ and τ'. We have $\tau' = \mathsf{match}\ d_j(u_i)^i$ with $(d_j(x_{ij})^i \to \tau_j)^{j\in J}$ and $\tau = \tau_j[x_{ij} \leftarrow u_i]^i$. The term τ_j has the same head as τ. Moreover, inverting the last syntactic rule of a kinding derivation for τ', we obtain $\Gamma \vdash \tau_j : \mathsf{Sch}$. We want to show that this is impossible. Consider the last syntactic rule of this derivation. It is of the form $\Gamma \vdash \tau_j : \kappa$, with $\kappa \neq \mathsf{Sch}$. Moreover, we have $\Gamma \vdash \kappa \simeq \mathsf{Sch}$. But this is absurd by Lemma 5.23. □

Then, we get subject reduction for term-level part of $\longrightarrow_\sharp$. We first prove an inversion lemma:

Lemma 5.29 (Inversion, meta, term level). *Consider an environment Γ.*

- *If $\Gamma \vdash \lambda^\sharp(x:\tau_1').\,a : \Pi(x:\tau_1).\,\tau_2$, then $\Gamma, x:\tau_1 \vdash a:\tau_2$.*
- *If $\Gamma \vdash \Lambda^\sharp(\alpha:\kappa').\,a : \forall^\sharp(\alpha:\kappa).\,\tau$, then $\Gamma, \alpha:\kappa \vdash a:\tau$.*
- *If $\Gamma \vdash \lambda^\sharp(\diamond: b_1' =_{\tau'} b_2').\,a : \Pi(\diamond: b_1 =_\tau b_2).\,\tau''$, then $\Gamma, (b_1 =_\tau b_2) \vdash a:\tau''$.*

Proof. Similar to the proof of Lemma 5.24. □

Lemma 5.30 (Subject reduction for $\longrightarrow_\sharp$). *Let Γ be a well-formed context. Suppose $X \longrightarrow_\sharp X'$. Then, if $\Gamma \vdash X : Y$, $\Gamma \vdash X' : Y$.*

Proof. Add the term-level part to the proof of Lemma 5.25 □

We can normalize further the conversions between two $\longrightarrow_\sharp$ normal forms:

Lemma 5.31 (Normal derivations, type-level meta reduction). *Suppose $\Gamma \vdash X_1 \simeq X_2$, where X_1 and X_2 are normal terms, types or kinds for $\longrightarrow_\sharp$. Then, there exists a derivation of $\Gamma \vdash^n X_1 \simeq X_2$, where $\Gamma \vdash^n X_1 \simeq X_2$ is a limited version of $\Gamma \vdash X_1 \simeq X_2$ where the rule C-Red-Meta is limited to $\longrightarrow_\sharp^a$. More precisely, this judgment is defined from the following rules:*

$$\text{C-Refl}\ \frac{\Gamma \vdash X : Y}{\Gamma \vdash^n X \simeq X} \qquad \text{C-Sym}\ \frac{\Gamma \vdash^n X_1 \simeq X_2}{\Gamma \vdash^n X_2 \simeq X_1} \qquad \text{C-Trans}\ \frac{\Gamma \vdash^n X_1 \simeq X_2 \qquad \Gamma \vdash^n X_2 \simeq X_3}{\Gamma \vdash^n X_1 \simeq X_3}$$

$$\text{C-Red-Iota'}\ \frac{X_1 \longrightarrow_\iota X_2 \qquad \Gamma \vdash X_1 : Y_1 \qquad \Gamma \vdash X_2 : Y_2}{\Gamma \vdash^n X_1 \simeq X_2} \qquad \text{C-Red-Meta'}\ \frac{X_1 \longrightarrow_\sharp^a X_2 \qquad \Gamma \vdash X_1 : Y_1 \qquad \Gamma \vdash X_2 : Y_2}{\Gamma \vdash^n X_1 \simeq X_2}$$

$$\text{C-Context}\ \frac{\Gamma \vdash C[\Gamma' \vdash X_1 : Y'] : Y \qquad \Gamma \vdash^n X_1 \simeq X_2}{\Gamma \vdash^n C[X_1] \simeq C[X_2]} \qquad \text{C-Eq}\ \frac{a_1 \longrightarrow_\sharp^* u_1 \qquad a_2 \longrightarrow_\sharp^* u_2 \qquad (a_1 =_\tau a_2) \in \Gamma}{\Gamma \vdash^n u_1 \simeq u_2}$$

$$\text{C-Split}\ \frac{(d_i : \forall(\alpha_k)_k\ (\tau_{ij})_j \to \zeta\ (\alpha_k)_k)_i \qquad (\Gamma, (x_{ij} : \tau_{ij}[(\alpha_k \leftarrow \tau_k)_k])_j, (u = d_i(x_{ij})) \vdash^n X_1 \simeq X_2)_i \qquad \Gamma \vdash u : \zeta\ (\tau_k)_k}{\Gamma \vdash^n X_1 \simeq X_2}$$

Proof. Similar to Lemma 5.26. □

We can now prove a projection result for *e*ML. The corresponding separation result is more complex and will be proved separately.

Lemma 5.32 (Projection for *e*ML). *Consider X and X' decomposing as $X \blacktriangleright h \circ (X_i)^i$ and $X' \blacktriangleright h \circ (X'_i)^i$. Suppose $\Gamma \vdash X \simeq X'$. Then, $\Gamma \vdash X_i \simeq X'_i$.*

Proof. The result for non-ML heads is already implied by the previous lemma. We can suppose that X and X' are types τ and τ' and are normal for $\longrightarrow_\sharp$, and that we have a normal derivation of $\Gamma \vdash \tau \simeq \tau'$.

We derive a stronger result: we define a function $\mathsf{tails}(h; \tau)$ that returns the tails of a type, assuming it has a given *e*ML head. We use Any_a to stand for any well-typed term, Any_t for a well-kinded type, and Any_k for a well-sorted kind (for example, $\mathsf{Any}_\mathsf{a} = \lambda(x : \mathsf{Any}_\mathsf{t}).\ x$, $\mathsf{Any}_\mathsf{t} = \forall(\alpha : \mathsf{Typ})\ \alpha$ and $\mathsf{Any}_\mathsf{k} = \mathsf{Typ}$).

$$\begin{array}{rcll} \mathsf{tails}(h; \tau) & = & (X_i)^i & \text{if } \tau \blacktriangleright h \circ (X_i)^i \\ \mathsf{tails}(h; \mathsf{match}\ a\ \mathsf{with}\ (P_i \to \tau_i)^i) & = & (\mathsf{match}\ a\ \mathsf{with}\ (P_i \to X_{ij})^i)^j & \text{if } \mathsf{tails}(h; \tau_i) = (X_{ij})^j \\ \mathsf{tails}(h; \tau) & = & (\mathsf{Any})^i & \end{array}$$

Note that the action of taking the tail commutes with substitution of terms: $\mathsf{tails}(h; \tau[x \leftarrow u]) = \mathsf{tails}(h; \tau)[x \leftarrow u]$. Moreover, if τ is well-typed, its tails $\mathsf{tails}(h; \tau)$ are well-typed or kinded (by inversion of the last syntax-directed rule of a kinding of τ).

Then, we show by induction on a normal derivation that whenever $\Gamma \vdash \tau \simeq \tau'$, for any ML head h, $\Gamma \vdash \mathsf{tails}(h; \tau)_i \simeq \mathsf{tails}(h; \tau')_i$. This is sufficient, because $\mathsf{tails}(h; \tau)_i = X_i$ and $\mathsf{tails}(h; \tau')_i = X'_i$. We suppose that the reductions are head-reductions, and that all applications of C-Context use only a shallow context.

- For C-Refl, invert the typing derivation to ensure that the tails are well-typed.
- There is no difficulty for C-Sym and C-Trans.
- For C-Split: prove the equality in each branch and merge using C-Split.

$$D_t ::= [] \sharp a \mid [] \sharp u \mid [] \sharp \tau \mid [] \sharp \diamond$$
$$D_v ::= [] \sharp a \mid [] \sharp u \mid [] \sharp \tau \mid [] \sharp \diamond \mid []\ a \mid []\ \tau \mid \text{match } [] \text{ with } \overline{P \to a}$$

$$c_t ::= \lambda^\sharp(x : \tau).\ \tau \mid \lambda^\sharp(x : \tau).\ \tau \mid \Lambda^\sharp(\alpha : \kappa).\ \tau \mid \lambda^\sharp(\diamond : a =_\tau a).\ \tau$$
$$c_v ::= \lambda^\sharp(x : \tau).\ a \mid \lambda^\sharp(x : \tau).\ a \mid \Lambda^\sharp(\alpha : \kappa).\ a \mid \lambda^\sharp(\diamond : a =_\tau a).\ a \mid \lambda(x : \tau).\ a \mid \text{fix } (x : \tau)\ x.\ a \mid \Lambda(\alpha : \mathsf{Typ}).\ a \mid d(\overline{a})$$

Fig. 16. Destructors and constructors

- The rule C-Eq does not apply in a typing context.
- For C-Red-Iota', the only possible head-reduction is a reduction of a type-level match: suppose we have $\tau = \text{match } d_j(u_i)^i \text{ with } (d_k(x_{ki})^i \to \tau_k)^k$ and $\tau' = \tau_j[x_{ji} \leftarrow u_i]^i$. Then, compute the tail: $\text{tails}(h; \tau)_l = \text{match } d_j(u_i)^i \text{ with } (d_k(x_{ki})^i \to X_{kl})^k$ where $X_{kl} = \text{tails}(h; \tau_k)_l$ and $\text{tails}(h; \tau') = \text{tails}(h; \tau_j)[x_{ji} \leftarrow u_i]^i$. The tails are well-typed, and reduce to one another, thus we can conclude by C-Red-Iota'
- For C-Context, consider the different cases:
  - If the context is of the form $C = \text{match } C' \text{ with } (P_i \to \tau_i)^i$ and is applied to X_1 and X_2, we have $\Gamma \vdash C'[X_1] = C'[X_2] \simeq$. Then we can substitute in the tails.
  - If the context is of the form $C = \text{match } a \text{ with } (P_i \to \tau_i)^i \mid P_j \to C'$, we can use the induction hypothesis: the tails of the case where the hole is are equal, so we can substitute in the global tails.
  - All other contexts distribute immediately in the tails. □

We obtain subject reduction.

Theorem 5.33 (Subject reduction). *Suppose Γ is well-formed, $X \longrightarrow X'$ and $\Gamma \vdash X : Y$. Then, $\Gamma \vdash X' : Y$.*

Proof. We need to prove subject reduction for $\longrightarrow_\iota$ and $\longrightarrow_\beta$. We prove this for head-reduction as in the other subject reduction results (see Lemma 5.25). For $\longrightarrow_\iota$, we can use the same technique as in the other proofs since C-Red-Iota' allows injecting $\longrightarrow_\iota$ in the equality.

For $\longrightarrow_\beta$, there are two cases. If we reduce an application, the evaluation context E is not dependent according to Lemma 5.21. In the other cases, the reduction is actually a $\longrightarrow_\iota$ reduction. □

Theorem 5.34 (Equal things have the same types, kinds, and sorts). *Consider a context Γ. Suppose $\Gamma \vdash X_1 \simeq X_2$. Then, for all Y, $\Gamma \vdash X_1 : Y$ if and only if $\Gamma \vdash X_2 : Y$.*

Proof. By induction on a derivation. This is immediate for reflexivity, transitivity and symmetry. Reduction preserves types by subject reduction. Substitution preserves types too. □

We can now use the simplified version of equality (C-Eq, C-Red-Iota, C-Red-Meta).

5.5 Soundness for $\longrightarrow_\sharp$

We now show that meta reductions are sound in any environment, and ML reductions are sound in the empty environment.

We define (see Figure 16) constructors c_t and c_v at the level of types and terms, and destructor contexts, or simply destructors, D_t and D_v for types and terms. Some destructors destruct terms but return types.

Moreover, we defined the predicate meta on constructors and destructors that do not belong to *e*ML (hence, use a meta-construction at the toplevel)

Theorem 5.35 (Soundness, meta). *Let Γ be an environment. Then:*

- *If $\Gamma \vdash D_t[c_t] : Y$, then $D_t[c_t] \longrightarrow^h$.*
- *If $\Gamma \vdash D_v[c_v] : Y$, then $D_v[c_v] \longrightarrow^h$.*

Proof. By case analysis on the destructor. We consider the case $D_t = [] \sharp \tau$. Consider the various cases for c_t: by Lemma 5.28, the only possible case is $c_t = \Lambda^\sharp(\alpha : \kappa).\ \tau'$. Then, $D_t[c_t]$ reduces. □

5.6 Reducing *m*ML to *e*ML

We will now show that any *m*ML term that can be typed in an environment without any meta construct normalizes by $\longrightarrow_\sharp$ to an *e*ML term of the same type. It does not suffice to normalize the term and check that it does not contain any *m*ML syntactic construct and conclude by subject reduction: we have to show the existence of an *e*ML typing derivation of the term.

Definition 5.36 (Meta-free context). A meta-free context is a context where the types of all (term) variables have kind Sch, and all type variables have kind Typ. A term is said to be meta-closed if it admits a typing under a meta-free context. A term is said to be *e*ML-typed if moreover its type has kind Sch. A type is said to be *e*ML-kinded if moreover it has kind Sch (or one of its subkinds).

Theorem 5.37 (Classification of meta-normal forms). *Consider a normal, meta-closed term or type. Then, it is an eML term or type, or it is not eML-typed (or eML-kinded) and starts with a meta abstraction.*

Proof. By induction on the typing or kinding derivation. Consider the last rule of a derivation:

- If it is a kind conversion K-Conv, by Lemma 5.23, it is a trivial conversion.
- If it is a type conversion, by Theorem 5.34, the kind of the type is preserved.
- If it is a construct in ML syntax: the subderivations on terms and types are also in meta-closed environments and *e*ML-typed or *e*ML-kinded, and we apply the induction hypothesis.
- If it is a meta-abstraction, it is not *e*ML-typed or *e*ML-kinded because of non-confusion of kinds.
- If it is a meta-application: let us consider the case of term-level meta type-application. The other cases are similar. We have $a = b \sharp \tau$. b is typeable in a meta-closed context but is not *e*ML-typed. Thus, it is a meta-abstraction. By soundness, a reduces, thus is not a normal form. □

We prove that all *m*ML derivations on *e*ML syntax that can be derived in *m*ML can also be derived in *e*ML. The difficulty comes from equalities: transitivity allows us to make *m*ML terms appear in the derivation; these must be reduced to *e*ML while maintaining a valid typing derivation.

Theorem 5.38 (*e*ML terms type in *e*ML). *In eML, consider an environment Γ; terms (*resp. *types, kinds) X, X_1, and X_2; and a type (*resp. *kind, sort) Y. Then:*

- *If $\vdash \Gamma$ in mML, then there is a derivation of $\vdash \Gamma$ in eML.*
- *If $\Gamma \vdash X : Y$ in mML, then there is a derivation of $\Gamma \vdash X : Y$ in eML.*
- *If $\Gamma \vdash X_1 \simeq X_2$ in mML, then there is a derivation of $\Gamma \vdash X_1 \simeq X_2$ in eML.*

Proof. By mutual induction.

We need to strengthen the induction for the typing derivations: we prove that, for any *m*ML type, kind or sort Y', if $\Gamma \vdash X : Y'$, Y' reduces to Y in *e*ML and $\Gamma \vdash X : Y$. Then notice that the conversions only happen between normal *e*ML terms, so we can apply the results on equalities.

For equalities, normalize the derivations as in Lemma 5.31, but simultaneously transform the typing derivations into *e*ML derivations (this must be done simultaneously otherwise we cannot control the size of the new derivations). □

The main result of this section follows.

Theorem 5.39 (Reduction from *m*ML to *e*ML). *If $\Gamma \vdash a : \tau$ and $\Gamma \vdash \tau :$ Sch are eML judgments that are derivable in mML, then there exists a reduction $a \longrightarrow_\sharp a'$ such that $\Gamma \vdash a' : \tau$ holds in eML.*

Note that this implies that *e*ML also admits subject reduction.

Proof. The well-typed term a normalizes by $\longrightarrow_\sharp$ to an irreducible term a'. By subject reduction, $\Gamma \vdash a' : \tau$. By classification of values, it is an *e*ML term. By Theorem 5.38, there is a derivation of $\Gamma \vdash a' : \tau$ in *e*ML. □

5.7 Soundness, via a logical relation for $\longrightarrow_\iota$

We prove that $\longrightarrow_\iota$ is normalizing, on all terms (including ill-typed terms):

Lemma 5.40 (Nomalization for $\longrightarrow_\iota$). *The reduction $\longrightarrow_\iota$ is strongly normalizing.*

Proof. We say that a term, type or kind is *good* if it admits no infinite reduction sequence, and if it reduces to $\Lambda(\alpha : \mathsf{Typ}).\, u$, for all good types τ, $u[\alpha \leftarrow \tau]$ is good. Goodness is stable by reduction.

We will prove the following property by induction on a term, type or kind X: suppose γ associates type and term variables to good terms and types. Then, $\gamma(X)$ is good. Let us consider the different cases:

- If X is a variable, $\gamma(X)$ is good by hypothesis.
- If $X = \Lambda(\alpha : \mathsf{Typ}).\, u$: by induction hypothesis, $\gamma(u)$ is good for all γ, thus $\gamma(X)$ admits no infinite reduction sequence. Moreover, if X reduces to $\Lambda(\alpha : \mathsf{Typ}).\, u'$, $u'[\alpha \leftarrow \tau]$ can be obtained by reduction from $(\gamma[\alpha \leftarrow \tau])(u)$, and thus is good.
- Suppose no head-reduction occurs from $\gamma(X)$. Then, since the subterms normalize, $\gamma(X)$ normalizes. We will now only consider the terms where head-reduction could occur.
- If $X = a\ \tau$, suppose $\gamma(X)$ reduces to a term that head-reduces. Then, this term is $X' = (\Lambda(\alpha : \mathsf{Typ}).\, u)\ \tau$, that reduces to $u[\alpha \leftarrow \tau]$. Since a is good, the result is good too.
- If $X = \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ has a head-reduction, it is from $\mathsf{let}\ x = u\ \mathsf{in}\ a_2'$ to $a_2'[x \leftarrow u]$, where $\gamma(a_1)$ reduces to u and $\gamma(a_2)$ reduces to a_2'. The term u is good, thus $(\gamma[x \leftarrow u])a_2$ is good and reduces to $a_2'[x \leftarrow u]$.
- Similarly for type and term-level pattern matching.

□

We then prove soundness of the $\longrightarrow_\iota$ reduction. This is done via a logical relation (essentially, implementing an evaluator for the non-expansive terms of *e*ML with the reduction $\longrightarrow_\iota$). This then allows proving (syntactically) that all conversions in the empty environment are between types having the same head (up to reduction). Let us note $u \longrightarrow_\iota^! v$ if all reduction paths from u terminate at v.

We start by defining a unary logical relation specialized to $\longrightarrow_\iota$ on Figure 17. It includes an interpretation $\mathbf{V}[\tau]_\gamma$ of values of type τ, an interpretation $\mathbf{E}[\tau]_\gamma$ of terms as normalizing to the appropriate values, an interpretation $\mathbf{G}[\gamma]\tau$ of the typing environments as environments associating variables to non-expansive terms. We also define binary interpretations (although they are interpreted in an unary environment). $\mathbf{EqE}[\tau]_\gamma$ of equality at type τ, via an interpretation $\mathbf{EqV}[\tau]_\gamma$ of equality for values. We will omit the typing and equality conditions in the definitions. The unary interpretation only contains terms that normalize, while the binary interpretation contains both pairs of terms that normalize by $\longrightarrow_\iota$, and pairs of terms stuck on a beta-reduction step. We write $\vdash \gamma : \Gamma$ to mean that γ is a well-typed environment that models Γ, and $\vdash \gamma_1 \simeq \gamma_2$ if all components of γ_1 and γ_2 are equal.

The definition of the interpretations is well-founded, by induction on types, and, for datatypes, by induction on the values (because the values appearing inside datatype constructors are necessarily values of a datatype, or functions from terms).

Definition 5.41 (Valid environment). An environment γ is valid if for all α such that $\gamma(\alpha) = (\tau, S, R)$,

- For all $u \in S$, $\emptyset \vdash u : \tau$.
- The restriction of R to S is an equivalence relation.

Lemma 5.42 (Definition of the interpretations). *The interpretations are defined for wall-kinded terms and well-formed environments:*

$$
\begin{array}{lcl}
\mathbf{G}[\Gamma] & \subseteq & \{\gamma \mid \ \vdash \gamma : \Gamma\} \\
\mathbf{G}[\Gamma, x : \tau] & = & \{\gamma[x \leftarrow u] \mid \gamma \in \mathbf{G}[\Gamma] \wedge u \in \mathbf{E}[\tau]_\gamma\} \\
\mathbf{G}[\Gamma, \alpha : \mathsf{Typ}] & = & \{\gamma[\alpha \leftarrow (\gamma(\tau), \mathbf{E}[\tau]_\gamma, \mathbf{EqE}[\tau]_\gamma)] \mid \gamma \in \mathbf{G}[\Gamma]\} \\
\mathbf{G}[\Gamma, (a_1 =_\tau a_2)] & = & \{\gamma \in \mathbf{G}[\Gamma] \mid a_1, a_2 \text{ non-expansive } \implies (\gamma(a_1), \gamma(a_2)) \in \mathbf{EqE}[\tau]_\gamma\} \\
\\
\mathbf{E}[\tau]_\gamma & \subseteq & \{a \mid \emptyset \vdash a : \gamma(\tau)\} \\
\mathbf{E}[\tau]_\gamma & = & \{u \mid \exists\,(v)\ u \longrightarrow_\iota^! v \wedge v \in \mathbf{V}[\tau]_\gamma\} \\
\\
\mathbf{V}[\tau]_\gamma & \subseteq & \{v \mid \emptyset \vdash v : \gamma(\tau)\} \\
\mathbf{V}[\alpha]_\gamma & = & \gamma(\alpha) \\
\mathbf{V}[\tau_1 \to \tau_2]_\gamma & = & \{\mathsf{fix}\ (x : \tau_1' \to \tau_2')\ y.\ a\} \\
\mathbf{V}[\forall(\alpha : \mathsf{Typ})\ \tau]_\gamma & = & \{(\Lambda(\alpha : \mathsf{Typ}).\ v) \mid \forall\,(\emptyset \vdash \tau' : \mathsf{Typ})\ v[\alpha \leftarrow \tau'] \in \mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\tau', \mathbf{E}[\tau']_\gamma, \mathbf{EqE}[\tau']_\gamma)]}\} \\
\mathbf{V}[\zeta\ (\tau_i)^i]_\gamma & = & \{(d(v_j)^j) \mid (d : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\ (\alpha_i)^i) \wedge \forall\,(j)\ v_j \in \mathcal{V}_k[\tau_j[\alpha_i \leftarrow \tau_i]^i]_\gamma\} \\
\end{array}
$$

$$
\mathbf{V}[\mathsf{match}\ a\ \mathsf{with}\ (d_i(x_{ij})^j \to \tau_i)^i]_\gamma = \begin{cases} \mathbf{V}[\tau_j]_{\gamma[x_{ij} \leftarrow v_j]^j} & \text{if } \gamma(a) \longrightarrow_\iota^! d_i(v_j)^j \\ \emptyset & \text{otherwise} \end{cases}
$$

$$
\begin{array}{lcl}
\mathbf{EqG}[\Gamma] & \subseteq & \{(\gamma_1, \gamma_2) \mid \ \vdash \gamma_1 \simeq \gamma_2\} \\
\mathbf{EqG}[\Gamma, x : \tau] & = & \{(\gamma_1[x \leftarrow u_1], \gamma_2[x \leftarrow u_2]) \mid (\gamma_1, \gamma_2) \in \mathbf{EqG}[\Gamma] \wedge (u_1, u_2) \in \mathbf{EqE}[\tau]_{\gamma_1}\} \\
\mathbf{EqG}[\Gamma, \alpha : \mathsf{Typ}] & = & \left\{ \left(\begin{array}{l} \gamma[\alpha \leftarrow (\gamma_1(\tau_1), \mathbf{E}[\tau_1]_{\gamma_1}, \mathbf{EqE}[\tau_2]_{\gamma_1})], \\ \gamma[\alpha \leftarrow (\gamma_2(\tau_2), \mathbf{E}[\tau_1]_{\gamma_2}, \mathbf{EqE}[\tau_2]_{\gamma_2})] \end{array} \right) \middle| \begin{array}{ll} & (\gamma_1, \gamma_2) \in \mathbf{EqG}[\Gamma] \\ \wedge & \mathbf{E}[\tau_1]_{\gamma_1} = \mathbf{E}[\tau_2]_{\gamma_2} \\ \wedge & \mathbf{EqE}[\tau_1]_{\gamma_1} = \mathbf{EqE}[\tau_2]_{\gamma_2} \end{array} \right\} \\
\mathbf{EqG}[\Gamma, (a_1 =_\tau a_2)] & = & \left\{ (\gamma_1, \gamma_2) \in \mathbf{G}[\Gamma] \middle| \begin{array}{l} a_1, a_2 \text{ non-expansive } \implies \\ \qquad (\gamma_1(a_1), \gamma_1(a_2)) \in \mathbf{EqE}[\tau]_{\gamma_1} \wedge (\gamma_2(a_1), \gamma_2(a_2)) \in \mathbf{EqE}[\tau]_{\gamma_2} \end{array} \right\} \\
\mathbf{EqE}[\tau]_\gamma & \subseteq & \{(a_1, a_2) \mid \emptyset \vdash a_1 \simeq a_2\} \\
\mathbf{EqE}[\tau]_\gamma & = & \left\{ (u_1, u_2) \middle| \begin{array}{ll} & (\exists\,(v_1\ v_2)\ u_1 \longrightarrow_\iota^! v_1 \wedge u_2 \longrightarrow_\iota^! v_2 \wedge (v_1, v_2) \in \mathbf{EqV}[\tau]_\gamma) \\ \vee & (\forall\,(v_1\ v_2)\ \neg(u_1 \longrightarrow_\iota^! v_1) \wedge \neg(u_2 \longrightarrow_\iota^! v_2)) \end{array} \right\} \\
\mathbf{EqV}[\tau]_\gamma & \subseteq & \{(v_1, v_2) \mid \emptyset \vdash v_1 \simeq v_2\} \\
\mathbf{EqV}[\alpha]_\gamma & = & \gamma(\alpha) \\
\mathbf{EqV}[\tau \to \tau']_\gamma & = & \{(\mathsf{fix}\ (x : \tau_1 \to \tau_1')\ y.\ a_1, \mathsf{fix}\ (x : \tau_2 \to \tau_2')\ y.\ a_2)\} \\
\mathbf{EqV}[\forall(\alpha : \mathsf{Typ})\ \tau]_\gamma & = & \{(\Lambda(\alpha : \mathsf{Typ}).\ v_1, \Lambda(\alpha : \mathsf{Typ}).\ v_2) \mid \forall\,(\emptyset \vdash \tau' : \mathsf{Typ})\ (v_1[\alpha \leftarrow \tau'], v_2[\alpha \leftarrow \tau']) \in \mathbf{V}[\tau]_{\gamma[\alpha \leftarrow \mathbf{EqE}[\tau']_\gamma]}\} \\
\mathbf{EqV}[\zeta\ (\tau_i)^i]_\gamma & = & \{(d(v_j)^j, d(w_j)^j) \mid d : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\ (\alpha_i)^i \wedge ((v_j, w_j) \in \mathbf{EqV}[\tau_j[\alpha_i \leftarrow \tau_i]^i]_\gamma)^j\} \\
\end{array}
$$

$$
\mathbf{EqV}[\mathsf{match}\ a\ \mathsf{with}\ (d_i(x_{ij})^j \to \tau_i)^i]_\gamma = \begin{cases} \mathbf{EqV}[\tau_j]_{\gamma[x_{ij} \leftarrow v_j]^j} & \text{if } \gamma(a) \longrightarrow_\iota^! d_i(v_j)^j \\ \emptyset & \text{otherwise} \end{cases}
$$

Fig. 17. Logical relation for $\longrightarrow_\iota$

- *If* $\vdash \Gamma$, $\mathbf{G}[\Gamma]$ *is well-defined and all its elements are valid, and* $\mathbf{EqG}[\Gamma]$ *is well-defined and reflexive on* $\mathbf{G}[\Gamma]$.
- *If* $\Gamma \vdash \tau : \mathsf{Sch}$, *for all* $\gamma \in \mathbf{G}[\Gamma]$, $\mathbf{E}[\tau]_\gamma$ *is defined, all its elements are non-expansive terms of type* $\gamma(\tau)$, *and* $\mathbf{EqE}[\tau]_\gamma$ *is an equivalence relation on* $\mathbf{E}[\tau]_\gamma$.

Proof. By mutual induction on the kinding and well-formed derivations. □

As usual, we need to prove a substitution result:

Lemma 5.43 (Substitution). *Consider a valid environment* γ. *Then,*

- $\mathbf{E}[\tau[x \leftarrow u]]_\gamma = \mathbf{E}[\tau]_{\gamma[x \leftarrow u]}$
- $\mathbf{EqE}[\tau[x \leftarrow u]]_\gamma = \mathbf{EqE}[\tau]_{\gamma[x \leftarrow u]}$
- $\mathbf{E}[\tau[\alpha \leftarrow \tau']]_\gamma = \mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\gamma(\tau'), \mathbf{E}[\tau']_\gamma, \mathbf{EqE}[\tau']_\gamma)]}$

- $\mathbf{EqE}[\tau[x \leftarrow \tau']]_\gamma = \mathbf{EqE}[\tau]_{\gamma[\alpha \leftarrow (\gamma(\tau'), \mathbf{E}[\tau']_\gamma, \mathbf{EqE}[\tau']_\gamma)]}$

Proof. By induction on τ. □

We also need to prove that reducing a value in the environment does not change the interpretation:

Lemma 5.44 (Reduction in the environment). *Consider a valid environment γ, and $u \longrightarrow_\iota u'$. Then,*

- $\mathbf{E}[\tau]_{\gamma[x \leftarrow u]} = \mathbf{E}[\tau]_{\gamma[x \leftarrow u']}$
- $\mathbf{EqE}[\tau]_{\gamma[x \leftarrow u]} = \mathbf{EqE}[\tau]_{\gamma[x \leftarrow u']}$

Proof. By induction on τ. The term variables in γ are used for substituting into types for the typing side-conditions. Since $\emptyset \vdash u \simeq u'$, the typing side-conditions stay true by Lemma 5.16. They also occur in the interpretation of pattern matching. But, for all terms a, $(\gamma[x \leftarrow u])(a)$ and $(\gamma[x \leftarrow u'])(a)$ normalize to the same term. □

Lemma 5.45 (Equality in the environment). *Consider a valid environment γ and τ_1, S, R such that $\gamma[\alpha \leftarrow (\tau_1, S, R)]$ is valid. Suppose $\emptyset \vdash \tau_1 \simeq \tau_2$. Then,*

- *$\gamma[\alpha \leftarrow (\tau_2, S, R)]$ is valid;*
- *for all types τ, $\mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\tau_1, S, R)]} = \mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\tau_2, S, R)]}$;*
- *for all types τ, $\mathbf{EqE}[\tau]_{\gamma[\alpha \leftarrow (\tau_1, S, R)]} = \mathbf{EqE}[\tau]_{\gamma[\alpha \leftarrow (\tau_2, S, R)]}$.*

Proof. By induction. The type τ_1 occurs only in the conclusion of typing derivations. Use conversion and $\emptyset \vdash \tau_1 \simeq \tau_2$ to get the same derivations with τ_2. □

Lemma 5.46 (Reduction preserves interpretation). *Suppose $\tau \longrightarrow_\iota \tau'$. Then,*

- *$\mathbf{E}[\tau]_\gamma = \mathbf{E}[\tau']_\gamma$;*
- *$\mathbf{EqE}[\tau]_\gamma = \mathbf{EqE}[\tau']_\gamma$.*

Proof. We will show the lemma for $\mathbf{E}[\tau]_\gamma$. We eliminate the context of the reduction by induction. If we pass to a reduction between terms, it is in the argument of a type-level match. Then, the two terms normalize to the same term. The only type-level reduction is the reduction of the type-level match:

$$\mathsf{match}\ d_j\ \overline{\tau_j}\ (u_i)^i\ \mathsf{with}\ (d_j\ \overline{\tau_j}\ (x_{ji})^i \to \tau_j)^{j \in J} \longrightarrow_\iota^h \tau_j[x_{ji} \leftarrow u_i]^i$$

Let us show that the two types have the same interpretation. We have $d_j\ \overline{\tau_j}\ (u_i)^i \longrightarrow_\iota^! d_j\ \overline{\tau_j}\ (v_i)^i$, where $u_i \longrightarrow_\iota^! v_i$. Thus, the interpretation of the left-hand side is $\mathbf{E}[\tau]_{\gamma[x_{ji} \leftarrow v_i]^i} = \mathbf{E}[\tau]_{\gamma[x_{ji} \leftarrow u_i]^i}$ by Lemma 5.44. By substitution (Lemma 5.43), $\mathbf{E}[\tau]_{\gamma[x_{ji} \leftarrow u_i]^i} = \mathbf{E}[\tau[x_{ji} \leftarrow u_i]^i]_\gamma$. This is the interpretation of the right-hand side. □

Lemma 5.47 (Evaluation for $\longrightarrow_\iota$). *Suppose $\vdash \Gamma$. Then:*

- *if $\Gamma \vdash u : \tau$, then for all $\gamma \in \mathbf{G}[\Gamma]$, $\gamma(u) \in \mathbf{E}[\tau]_\gamma$;*
- *if $\Gamma \vdash \tau_1 \simeq \tau_2$, then for all $(\gamma_1, \gamma_2) \in \mathbf{EqG}[\Gamma]$, $\mathbf{E}[\tau_1]_\gamma = \mathbf{E}[\tau_2]_\gamma$ and $\mathbf{EqE}[\tau_1]_{\gamma_1} = \mathbf{EqE}[\tau_2]_{\gamma_2}$;*
- *if $\Gamma \vdash a_1 \simeq a_2$ and $\Gamma \vdash a_1 : \tau$, then for all $(\gamma_1, \gamma_2) \in \mathbf{EqG}[\Gamma]$, $(\gamma_1(a_1), \gamma_2(a_2)) \in \mathbf{EqE}[\tau]_{\gamma_1}$;*
- *if $\Gamma \vdash \tau : \mathsf{Sch}$, then for all $(\gamma_1, \gamma_2) \in \mathbf{EqG}[\Gamma]$, $\mathbf{EqE}[\tau]_{\gamma_1} = \mathbf{EqE}[\tau]_{\gamma_2}$;*
- *if $\Gamma \vdash a : \tau$, then for all $(\gamma_1, \gamma_2) \in \mathbf{EqG}[\Gamma]$, $(\gamma_1(a), \gamma_2(a)) \in \mathbf{EqE}[\tau]_{\gamma_1}$.*

Proof. We prove these results by mutual induction on the derivations.
For the result on typing derivations, let us consider the different rules:

- For Var on $x : \tau$: by hypothesis, $\gamma(x) \in \mathbf{E}[\tau]_\gamma$.
- For Conv: use the second lemma to show the interpretations of the two types are the same.
- For Fix: any two well-typed abstractions are linked at an arrow type.
- The rule App cannot occur as the first rule in the typing of a non-expansive term.

- For TAbs:

$$\textsc{TAbs}\quad \frac{\Gamma, \alpha : \mathsf{Typ} \vdash u : \tau}{\Gamma \vdash \Lambda(\alpha : \mathsf{Typ}).\, u : \forall(\alpha : \mathsf{Typ})\ \tau}$$

  Consider $\gamma \in \mathbf{G}[\Gamma]$ and $\emptyset \vdash \tau' : \mathsf{Typ}$. $\Lambda(\alpha : \mathsf{Typ}).\, u$ normalizes to $\Lambda(\alpha : \mathsf{Typ}).\, v$ with $u \longrightarrow_\iota^! v$. By induction hypothesis, $(\gamma[\alpha \leftarrow \tau'])(u) \in \mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\tau', \mathbf{E}[\tau']_\gamma, \mathbf{EqE}[\tau']_\gamma)]}$. Moreover, it reduces to $(\gamma[\alpha \leftarrow \tau'])(v) = \gamma(v)[\alpha \leftarrow \tau']$. Thus, $\gamma(v)[\alpha \leftarrow \tau'] \in \mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\tau', \mathbf{E}[\tau']_\gamma, \mathbf{EqE}[\tau']_\gamma)]}$.
- For TApp:

$$\textsc{TApp}\quad \frac{\Gamma \vdash \tau' : \mathsf{Typ} \qquad \Gamma \vdash u : \forall(\alpha : \mathsf{Typ})\ \tau}{\Gamma \vdash u\ \tau' : \tau[\alpha \leftarrow \tau']}$$

  Consider $\gamma \in \mathbf{G}[\Gamma]$. There exists τ'' such that $\gamma(\tau') \longrightarrow_\iota^! \tau''$. Then, $\emptyset \vdash \tau'' : \mathsf{Typ}$. There exists v such that $\gamma(u) \longrightarrow_\iota^! v$. By inductive hypothesis, $v \in \mathbf{V}[\forall(\alpha : \mathsf{Typ})\ \tau]_\gamma$. Thus, there exists v' such that $v = \Lambda(\alpha : \mathsf{Typ}).\, v'$, and $v'[\alpha \leftarrow \tau''] \in \mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\tau'', \mathbf{E}[\tau'']_\gamma, \mathbf{EqE}[\tau'']_\gamma)]} = \mathbf{E}[\tau[\alpha \leftarrow \tau'']]_\gamma = \mathbf{E}[\tau[\alpha \leftarrow \tau']]_\gamma$ by Lemmas 5.43 and 5.46. We need to prove $\gamma(u\ \tau') \in \mathbf{E}[\tau[\alpha \leftarrow \tau']]_\gamma$. But we have $\gamma(u\ \tau') \longrightarrow_\iota^* (\Lambda(\alpha : \mathsf{Typ}).\, v')\ \tau'' \longrightarrow_\iota v'[\alpha \leftarrow \tau'']$.
- The other cases are similar.

For the result on equalities between types, by induction on a derivation. We suppose that the context rule is always used with a shallow context.

- For C-Sym and C-Trans, use the induction hypothesis and symmetry/transitivity of equality.
- For C-Refl, use reflexivity on types.
- For C-Context: apply the induction hypothesis on the modified subterm if it is a type, and reflexivity on the other subterms.
- Otherwise, it is the argument of a type-level pattern matching. We have $\Gamma \vdash a_1 : \zeta\ (\tau_i)^i$, $\Gamma \vdash a_2 : \zeta\ (\tau_i)^i$, and $\Gamma \vdash a_1 \simeq a_2$. By the third result, $(\gamma_1(a_1), \gamma_2(a_2)) \in \mathbf{EqE}[\zeta\ (\tau_i)^i]_{\gamma_1}$. If both do not normalize to a value, the two interpretations of the pattern matching are empty. Otherwise they normalize to $d(v_{1j})^j$ and $d(v_{2j})^j$ such that $(v_{1j}, v_{2j}) \in \mathbf{EqE}[\tau_j]_{\gamma_1}$ where the τ_j are the types of the arguments. Thus, $(\gamma_1[x_j \leftarrow v_{1j}]^j, \gamma_2[x_j \leftarrow v_{2j}]^j) \in \mathbf{EqG}[\Gamma, (x_j : \tau_j)^j]$, and we can interpret the selected branch in these environments.
- For C-Split on a term $\Gamma \vdash u : \tau'$, use reflexivity on types to prove that $(\gamma_1(u), \gamma_2(u)) \in \mathbf{EqE}[\tau']_{\gamma_1}$. Moreover, $\gamma_1(u) \in \mathbf{E}[\tau']_{\gamma_1}$, thus the terms normalize to a value. Then, select the case corresponding to the constructor, construct an environment as in the previous case, and use the induction hypothesis.
- For C-Red-Iota, suppose we have a head-reduction (otherwise, use C-Context). Then, it is the reduction of a pattern matching. Proceed as in C-Split.
- The rule C-Eq does not apply on types.

For the result on equalities between terms:

- The cases of C-Sym, C-Trans and C-Split are similar to the same cases on types.
- For C-Refl, use reflexivity on terms.
- For C-Red-Iota, proceed as in C-Red-Iota for types.
- For C-Eq, consider the two equal terms u_1, u_2. From γ_1 we get $(\gamma_1(u_1), \gamma_1(u_2)) \in \mathbf{EqE}[\tau]_{\gamma_1}$. Moreover, by reflexivity on u_2, $(\gamma_1(u_2), \gamma_2(u_2)) \in \mathbf{EqE}[\tau]_{\gamma_1}$. We conclude by transitivity of $\mathbf{EqE}[\tau]_{\gamma_1}$.
- For C-Context, examine the different typing rules as in the first result, and use reflexivity when needed.

For the reflexivity results, examine the different typing rules as in the first result. Take special care for variables. The interpretations of the type variables are the same. For term variables appearing in terms, they are bound to related terms. Finally, for type-level pattern matching, the interpretations of the term in the environments are related by reflexivity. □

$$\begin{array}{rcl} E & ::= & \ldots \mid E \sharp u \mid E \sharp \tau \mid E \sharp \diamond \\ v & ::= & \ldots \mid \lambda^\sharp(x : \tau).\ a \mid \Lambda^\sharp(\alpha : \kappa).\ a \mid \lambda^\sharp(\diamond : a =_\tau a).\ a \end{array} \qquad \frac{a \longrightarrow^h_\sharp b}{E[a] \mapsto E[b]}\ \textsc{Ctx-Det-Meta} \qquad \frac{a \longrightarrow^h_\beta b}{E[a] \mapsto E[b]}\ \textsc{Ctx-Beta-Meta}$$

Fig. 18. Deterministic reduction $\mapsto$ for *m*ML

The following result is then a direct consequence:

Lemma 5.48 (Separation for $\longrightarrow_\iota$). *Suppose $\emptyset \vdash \tau_1 \simeq \tau_2$. Then if τ_1 and τ_2 have a head, it is the same.*

Proof. Apply the previous result with the empty environment. The interpretations of types with distinct heads are distinct. □

Theorem 5.49 (Soundness, empty environment).

- *If $\emptyset \vdash D_t[c_t] : Y$, then $D_t[c_t] \longrightarrow^h$.*
- *If $\emptyset \vdash D_v[c_v] : Y$, then $D_v[c_v] \longrightarrow^h$.*

Proof. Similar to Theorem 5.35, but use the separation theorem for empty environments. □

6 A step-indexed logical relation on *m*ML

To give a semantics to ornaments and establish the correctness of elaboration, we define a step-indexed logical relation on *m*ML. We later give a definition of ornamentation using the logical relation. Instead of defining a relation compatible with the strong and non-deterministic reduction on *m*ML, we choose a more standard presentation and define a deterministic subset of the reduction. We also ignore all reductions on types. This relation will be used to prove soundness for *e*ML, and that *e*ML programs can be transformed into equivalent ML programs.

6.1 A deterministic reduction

We never need to evaluate types for reduction to proceed to a value. Thus, our reduction will ignore the types appearing in terms (and the terms appearing in these types, *etc.*) and only reduce the term part that actually computes. We define a subreduction $\mapsto$ of $\longrightarrow$ by restricting the reduction $\longrightarrow_\sharp$ to a call-by-name left-to-right strategy. The deterministic meta-reduction is defined as applying only in ML evaluation contexts. We extend the ML reduction to also occur on the left-hand side of meta-applications. This does not change the metatheory of the language, as such terms are necessarily ill-typed, but it allows us to use the same evaluation contexts for ML reduction and meta reduction. The values are also extended to contain meta-abstractions. By a similar argument, these values are not passed to any *e*ML construct because they have type Met, which is not allowed in *e*ML. With these changes, we get a deterministic meta-reduction $\mapsto$ defined as the ML and meta head-reductions for terms, applied under the extended ML evaluation contexts E. We write $\mapsto^h$ the associated head-reduction, *i.e.* the union of all head-reduction of terms.

The reduction $\mapsto$ admits the usual properties: it is deterministic (and thus confluent), and the values are irreducible.

Lemma 6.1. *Values v are irreducible for $\mapsto$.*

Proof. We show a slightly more general result: if a value v is of the form $E[a]$, then a not a value v'. This is enough, as values do not head-reduce.

Proceed by induction on v. Assume v is $E[a]$. If E is the empty context, a is v which is a value. Otherwise, only constructors can appear as the root of both an evaluation context and a value. Then, the context E is of the form $d(\overline{v}', E', \overline{b})$, and $v = d(\overline{v}'')$. Thus, $E'[a]$ is a value, and we use the induction hypothesis to show that a is a value. □

Lemma 6.2 (Determinism). *Consider a term a. Then, there exists at most one term a' such that $a \mapsto a'$.*

Proof. We show, by induction on the term, that there exists at most one decomposition $a = E[b]$, with b head-reducible. This suffices because head-reduction is deterministic. We have shown in the previous proof that values don't admit such a decomposition.

Consider the different cases for a.

- a is an abstraction or a fixed point: it does not decompose further since there is no appropriate non-empty evaluation context.
- a is $d(a)_i^i$: either it is a value, or we can decompose the sequence $(a)_i^i = (b)_j^j, b, (v)_k^k$, with b not a value. Necessarily, $E = d((b)_j^j, E', (v)_k^k)$ (because E cannot be empty: constructors do not reduce). But then, b admits a unique decomposition as $E'[b']$.
- a is let $x = a_1$ in a_2: If a_1 is a value, then it does not decompose, and the only possible context is the empty context []. Otherwise, it admits at most one decomposition $E'[b]$, and a admits only the decomposition let $x = E'[b]$ in a_2.
- a is $a_1\ a_2$: If a_2 is not a value, a does not head-reduce and the only possible decomposition of E is $a_1\ E'$. Since a_2 decomposes uniquely, E' is unique, thus E is unique. Otherwise a_2 is a value v, If a_1 is not a lambda, a does not head-reduce, and the only possible decomposition of E is $E'\ v$. By induction, E' is unique. If both are values, the only possible evaluation context is the empty context.
- Cases for all others applications are similar, except that there is no context allowing the reduction of the right-hand side of the application, thus it is not necessary to check whether it is a value.

□

We can now link the deterministic reduction $\mapsto$ and the full reduction $\longrightarrow$.

We first note that ML reduction is impossible on values:

Lemma 6.3. *If $v \longrightarrow a$, then we actually have $v \longrightarrow_\sharp a$ and a is a value.*

Proof. The ML reduction $\longrightarrow_\beta$ is included in $\mapsto$, and values do not reduce for $\mapsto$. By induction on the values, we can prove that redexes only occur under abstractions. Thus, the term remains a value after reduction. □

We write $\longrightarrow_\lambda$ the reduction that only reduces ML term abstractions. It is also the subset of $\longrightarrow_\beta$ reductions that are not included in $\longrightarrow_\iota$.

Lemma 6.4 (Commutations for well-typed terms). *Meta-reductions can always be done first, moreover, β-reductions and ι-reductions commute. More precisely, assume X_1 is a well-typed term:*

- *If $X_1 \longrightarrow_\iota X_2 \longrightarrow_\sharp X_3$, then $X_1 \longrightarrow_\sharp X_4 \longrightarrow_\iota^* X_3$ for some X_4.*
- *If $X_1 \longrightarrow_\lambda X_2 \longrightarrow_\sharp X_3$, then $X_1 \longrightarrow_\sharp^* X_4 \longrightarrow_\lambda X_3$ for some X_4.*
- *If $X_1 \longrightarrow_\lambda X_2$ and $X_1 \longrightarrow_\iota X_3$, then $X_2 \longrightarrow_\iota^* X_4$ and $X_3 \longrightarrow_\lambda X_4$ for some X_4.*

Proof. For the first two commutations: a typing argument prevents $\longrightarrow_\iota$ and $\longrightarrow_\lambda$ from creating meta-redexes. For the third property: note that $\longrightarrow_\iota$ cannot duplicate a $\longrightarrow_\lambda$ redex in an evaluation context. □

Lemma 6.5 (Normalization for $\longrightarrow_\iota$ and $\longrightarrow_\sharp$). *The union of $\longrightarrow_\iota$ and $\longrightarrow_\sharp$ is strongly normalizing on well-typed terms.*

Proof. Consider a term X. By König's lemma (since by Theorem 5.2, there is a finite number of possible reductions from one term), all possible $\longrightarrow_\sharp$ reduction sequences from X are of length at most k for some k. Consider an infinite $(\longrightarrow_\iota \cup \longrightarrow_\sharp)$ reduction sequence from X. It has k $\longrightarrow_\sharp$ reductions or less. Otherwise, we could use Lemma 6.4 to put $k+1$ $\longrightarrow_\sharp$ reductions at the start of the sequence. Thus, after the last $\longrightarrow_\sharp$ reduction, there is an infinite sequence of $\longrightarrow_\iota$ reduction. But this is impossible by Lemma 5.40. □

Lemma 6.6 (Weak normalization implies strong normalization). *Consider a term a. Suppose there exists a terminating reduction path from a. Then, all reduction sequences are finite.*

Proof. We show that all reduction sequences from a have the same number of $\longrightarrow_\lambda$ reductions. We can, without loss of generality (by Lemma 6.4) assume that a is meta-normal and that the reduction path is meta-normal.

Consider a normalizing reduction sequence from a. We are interested in the $\longrightarrow_\lambda$ reductions. Thus, we decompose the sequence as $a = a_0 \longrightarrow_\iota^* a_0' \longrightarrow_\lambda a_1 \longrightarrow_\iota^* a_1' \ldots \longrightarrow_\lambda a_n \longrightarrow_\iota^* a_n'$. Consider a longer reduction sequence from a. We can decompose it as a finite sequence $a = b_0 \longrightarrow_\iota^* b_0' \longrightarrow_\lambda b_1 \ldots$.

Let us show by induction that for all $i \le n$, there exists c_i such that $a_i' \longrightarrow_\iota^* c_i$ and $b_i' \longrightarrow_\iota^* c_i$. This is true for 0. Suppose it is true for i. Then, we can use Lemma 6.4 to transport the reductions at the next step, and conclude by confluence. Finally, $a_n' = c_n$ since a_n' is irreducible. But, since b_n' reduces by $\longrightarrow_\lambda$, c_n reduces by $\longrightarrow_\lambda$. □

This suffices to show that the reductions coincide, up to reduction under abstractions:

Lemma 6.7 (Equivalence of the deterministic reduction).

- *If $a \mapsto v$ and a normalizes by $\longrightarrow$ to a', then $v \longrightarrow^* a'$.*
- *If $a \longrightarrow^* v$, then $a \mapsto v'$ for some v'. More precisely:*
  - *If $a \longrightarrow^* d(v_i)^i$, then $a \mapsto^* d(v_i')^i$ and for all i, $v_i' \longrightarrow_\sharp^* v_i$.*
  - *If $a \longrightarrow^* \mathsf{fix}\ (x : \tau_1)\ y.\ b$, then $a \mapsto^* \mathsf{fix}\ (x : \tau_1)\ y.\ b$ and $b' \longrightarrow_\sharp^* b$.*
  - *If $a \longrightarrow^* \lambda^\sharp(x : \tau).\ b$, then $a \mapsto^* \lambda^\sharp(x : \tau').\ b'$ and $b' \longrightarrow_\sharp^* b$.*
  - *If $a \longrightarrow^* \Lambda^\sharp(\alpha : \kappa).\ b$, then $a \mapsto^* \Lambda^\sharp(\alpha : \kappa').\ b'$ and $b' \longrightarrow_\sharp^* b$.*
  - *If $a \longrightarrow^* \lambda^\sharp(\diamond : a_1 =_\tau a_2).\ b$, then $a \mapsto^* \lambda^\sharp(\diamond : a_1' =_\tau' a_2').\ b'$ and $b' \longrightarrow_\sharp^* b$.*

Proof. The first result is rephrasing of Lemma 6.6. Consider the second result. We start by proving that whenever $a \mapsto v'$, v' has the correct form. This is a consequence of confluence, and the fact that head constructors are preserved by reduction.

Then, we only need to prove that the deterministic reduction does not get stuck when the full reduction does not: suppose $a \longrightarrow v$, then either a is a value or $a \mapsto$. We will proceed by structural induction on a.

- If $a = x$, a does not reduce.
- Consider $a = \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$. The let binding cannot be the root of a value, so $\longrightarrow$ will reduce it at some point: there exists $a_1 \longrightarrow^* v_1$. By induction hypothesis, a_1 reduces or is a value. If it is a value, a head-reduces, and otherwise the subterm a_1 reduces.
- Suppose a is an abstraction. Then it is a value.
- Suppose a is an application. We will only consider the case $a = a_1\ a_2$, the cases of the other applications are similar. A value cannot start with an application. Thus, the application will be reduced at some points. Then, there exists τ, b and w such that $a_1 \longrightarrow^* \lambda(x : \tau).\ b$ and $a_2 \longrightarrow^* w$. Suppose a_2 is not already a value. Then, by induction hypothesis it reduces, so a reduces by $\mapsto$. Otherwise, suppose a_1 is not already a value. Then, by induction hypothesis it reduces by $\mapsto$, and a reduces. Otherwise, we have $a = (\lambda(x : \tau).\ b)\ v$, and a is head-reducible.

$$
\begin{array}{rl}
(\mathsf{fix}\ (x : \tau)\ y.\ a)\ v & \mapsto_1^h a[x \leftarrow \mathsf{fix}\ (x : \tau)\ y.\ a, y \leftarrow v] \\
(\Lambda(\alpha : \mathsf{Typ}).\ v)\ \tau & \mapsto_0^h v[\alpha \leftarrow \tau] \\
\mathsf{let}\ x = v\ \mathsf{in}\ a & \mapsto_0^h a[x \leftarrow v] \\
\begin{array}{l}\mathsf{match}\ d_j\ \overline{\tau_j}\ (v_i)^i\ \mathsf{with} \\ \quad (d_j\ \overline{\tau_j}\ (x_{ji})^i \to a_j)^j\end{array} & \mapsto_0^h a_j[x_{ij} \leftarrow v_i]^i \\
(\lambda^\sharp(x : \tau).\ a) \sharp u & \mapsto_0^h a[x \leftarrow u] \\
(\Lambda^\sharp(\alpha : \kappa).\ a) \sharp \tau & \mapsto_0^h a[\alpha \leftarrow \tau] \\
(\lambda^\sharp(\diamond : b_1 =_\tau b_2).\ a) \sharp \diamond & \mapsto_0^h a
\end{array}
$$

$$
\textsc{Context}\ \frac{a \mapsto_i^h b}{E[a] \mapsto_i E[b]}
\qquad
\textsc{Identity}\ \frac{}{a \mapsto_0 a}
\qquad
\textsc{Composition}\ \frac{a_1 \mapsto_i a_2 \qquad a_2 \mapsto_j a_3}{a_1 \mapsto_{i+j} a_3}
$$

Fig. 19. The counting reduction $\mapsto_i$

- Consider $a = d(a_i)^i$. It reduces by $\longrightarrow$ to a value that is necessarily of the form $d(v_i)^i$, with $a_i \longrightarrow^* v_i$. If all a_i are values, a is a value. Otherwise, consider the last index i such that a_i is not a value. Then, by induction hypothesis, it reduces by $\mapsto$. Thus, a reduces by $\mapsto$.
- Consider $a = \mathsf{match}\ b\ \mathsf{with}\ (d_j\ \overline{\tau_j}\ (x_{ji})^i \to a_j)^j$. A value cannot start with a pattern matching, so $\longrightarrow$ will reduce it at some point. Thus, there exists d and $(a_i)^i$ such that $b \longrightarrow^* d(a_i)^i$. Thus, there exists $(a'_i)^i$ such that $b \mapsto^* d(a'_i)^i$. If the reduction takes one step or more, a reduce under the pattern matching. Otherwise, the pattern matching itself reduces.

□

6.2 Counting steps

We define an indexed version of this reduction as follows: the beta-reduction and the expansion of fixed points in ML take one step each, and all other reductions take 0 steps. Then, $\mapsto_i$ is the reduction of cost i, *i.e.* the composition of i one-step reductions and an arbitrary number of zero-step reductions. The full definition of the indexed reduction is given on Figure 19. Since $\mapsto_0$ is a subset of the union of $\longrightarrow_\iota$ and $\longrightarrow_\sharp$, it terminates.

6.3 Semantic types and the interpretation of kinds

We want to define a typed, binary, step-indexed logical relation. The (relational) types will be interpreted as pairs of (ground) types and a relation between them. This relation is step-indexed, *i.e.* it is defined as the limit of a sequence of refinement of the largest relation between these types. The type-level functions are interpreted as function between these representation, *i.e.* a pair of type-level functions for the left- and right-hand side and a function of step-indexed relations subject to a causality constraint.

The interpretation of kinds is parameterized by a pair of term environments for the left and right-hand side of the relation. The environments must be well-typed: we define a judgment $\vdash \gamma : \Gamma$ that checks that all bindings in γ have the right type or kind.

Then, we define by induction on kinds an interpretation $\mathcal{K}[\kappa]_{\gamma_1,\gamma_2}$, defined for all κ, γ_1, γ_2 such that there exists Γ such that $(\vdash \Gamma : \gamma_i)^i$ and $\Gamma \vdash \kappa : \mathsf{wf}$. The interpretation is a set of triples $(\tau_1, (S_j)^{j \le i}, \tau_2)$ such that $(\emptyset \vdash \tau_i : \gamma_i(\kappa))^i$. In the interpretation of the base kinds Typ, Sch, Met, the S_j are a decreasing sequence of relations on values of the correct types. For higher-order constructs, the S_j are functions that map interpretations of one kind to interpretations of another kind. Equality between the interpretations is considered up to type equality.

Lemma 6.8. *The interpretation of kinds is well-defined.*

$$\textsc{Env-Empty}\quad \dfrac{}{\vdash \emptyset : \emptyset} \qquad \textsc{Env-TVar}\quad \dfrac{\emptyset \vdash \tau : \gamma(\kappa) \qquad \vdash \gamma : \Gamma}{\vdash \gamma[\alpha \leftarrow \tau] : \Gamma, \alpha : \kappa} \qquad \textsc{Env-Var-Term}\quad \dfrac{\emptyset \vdash a : \gamma(\tau) \qquad \vdash \gamma : \Gamma}{\vdash \gamma[x \leftarrow a] : \Gamma, x : \tau} \qquad \textsc{Env-Var-NonExp}\quad \dfrac{\emptyset \vdash u : \gamma(\tau) \qquad \vdash \gamma : \Gamma}{\vdash \gamma[x \leftarrow u] : \Gamma, x : \tau}$$

$$\textsc{Env-Eq}\quad \dfrac{\emptyset \vdash \gamma(a) \simeq \gamma(b) \qquad \vdash \gamma : \Gamma}{\vdash \gamma : \Gamma, (a =_\tau b)}$$

Fig. 20. The environment typing judgment $\vdash \gamma : \Gamma$

$$\mathcal{K}[\kappa \in \{\mathsf{Typ}, \mathsf{Sch}, \mathsf{Sch}, \mathsf{Met}\}]_{\gamma_1,\gamma_2} = \left\{ (\tau_1, (R_j)^{j \le i}, \tau_2) \;\middle|\; \begin{array}{ll} & \vdash \tau_1 : \kappa \wedge \vdash \tau_2 : \kappa \\ \wedge & \forall\, (j \le i)\, ((v_1, v_2) \in R_j)\ \vdash v_1 : \tau_1 \wedge \vdash v_2 : \tau_2 \\ \wedge & \forall\, (j \le k \le i)\ , R_j \supseteq R_k \end{array} \right\}$$

$$\mathcal{K}[\forall(\alpha : \kappa_1)\ \kappa_2]_{\gamma_1,\gamma_2} = \left\{ (\tau_1, (F_j)^{j \le i}, \tau_2) \;\middle|\; \begin{array}{ll} & \vdash \tau_1 : \forall(\alpha : \gamma_1(\kappa_1))\ \gamma_1(\kappa_2) \wedge \vdash \tau_2 : \forall(\alpha : \gamma_2(\kappa_1))\ \gamma_2(\kappa_2) \\ \wedge & \forall j \le i, (\tau_1', (S_k)^{k \le j}, \tau_2') \in \mathcal{K}[\kappa_1]_{\gamma_1,\gamma_2} \\ & \quad (\tau_1 \sharp \tau_1', (F_k(S_k))^{k \le j}, \tau_2 \sharp \tau_2') \in \mathcal{K}[\kappa_2]_{\gamma_1[\alpha \leftarrow \tau_1'], \gamma_2[\alpha \leftarrow \tau_2']} \end{array} \right\}$$

$$\mathcal{K}[\tau \to \kappa]_{\gamma_1,\gamma_2} = \left\{ (\tau_1, (f_j)^{j \le i}, \tau_2) \;\middle|\; \begin{array}{ll} & \vdash \tau_1 : \gamma_1(\tau) \to \gamma_1(\kappa) \wedge \vdash \tau_2 : \gamma_2(\tau) \to \gamma_2(\kappa) \\ \wedge & \forall\, (u_1, u_2)\ (\vdash u_1 : \gamma_1(\tau) \wedge \vdash u_2 : \gamma_2(\tau)) \\ & \quad \Longrightarrow\ (\tau_1 \sharp u_1, (f_j(u_1, u_2))^{j \le i}, \tau_2 \sharp u_2) \in \mathcal{K}[\kappa]_{\gamma_1,\gamma_2} \\ \wedge & \forall u_1, u_2, u_1', u_2', (\vdash u_1 \simeq u_1' \wedge \vdash u_2 \simeq u_2') \\ & \quad \Longrightarrow\ \forall j, f_j(u_1, u_2) = f_j(u_1', u_2') \end{array} \right\}$$

$$\mathcal{K}[(a =_\tau b) \to \kappa]_{\gamma_1,\gamma_2} = \begin{cases} \mathcal{K}[\kappa]_{\gamma_1,\gamma_2} & \text{if } \vdash \gamma_1(a) \simeq \gamma_1(b) \wedge \vdash \gamma_2(a) \simeq \gamma_2(b) \\ \{\bullet\} & \text{otherwise} \end{cases}$$

Fig. 21. Interpretation of kinds

Proof. We must prove that the types appearing in the triples are correctly kinded. This is guaranteed by the kinding conditions in each case. □

Lemma 6.9 (Equal kinds have equal interpretations). *Consider* $\vdash \gamma_1, \gamma_2 : \Gamma$. *Then, if* $\Gamma \vdash \kappa_1 \simeq \kappa_2$, $\mathcal{K}[\kappa_1]_{\gamma_1,\gamma_2} = \mathcal{K}[\kappa_2]_{\gamma_1,\gamma_2}$.

Proof. By induction on the kinds: Lemma 5.23 allows us to decompose the kinds and get equality between the parts. For the kinding conditions, note that if $\Gamma \vdash \kappa_1 \simeq \kappa_2$, then $\vdash \gamma_1(\kappa_1) \simeq \gamma_2(\kappa_2)$. □

6.4 The logical relation

We define a typed binary step-indexed logical relation on *m*ML equipped with $\mapsto$. The interpretation of types of terms $\mathcal{E}_k[\tau]_\gamma$ goes through an interpretation of types as a relation on values $\mathcal{V}_k[\tau]_\gamma$. These interpretations depend on an environment γ. The interpretation of a type of terms as values is an arbitrary relation between values. The interpretation of types of higher kind is a function from the interpretation of its arguments to the interpretation of its result. Typing environments Γ are interpreted as a set of environments γ that map types variables to either relations in (for arguments of kind Sch) or syntactic types (for higher-kinded types), and term variables to pairs of (related) terms. Equalities are interpreted as restricting the possible environments to those where the two terms can be proved equal using the typing rules.

Each step of the relation is a triple $(\tau_1, (R_j)^{j \le i}, \tau_2)$. For compactness, we will only specify the value of R_i and leave implicit the values of τ_1 and τ_2 (that are simply obtained by applying γ_1, γ_2 to the type τ).

$$
\begin{array}{lcl}
\mathcal{G}_k[\emptyset] & = & \{\emptyset\} \\
\mathcal{G}_k[\Gamma, x : \tau] & = & \{\gamma[x \leftarrow (u_1, u_2)] \mid (u_1, u_2) \in \mathcal{E}_k[\tau]_\gamma \wedge \gamma \in \mathcal{G}_k[\Gamma]\} \\
\mathcal{G}_k[\Gamma, x : \tau] & = & \{\gamma[x \leftarrow (a_1, a_2)] \mid (a_1, a_2) \in \mathcal{E}_k[\tau]_\gamma \wedge \gamma \in \mathcal{G}_k[\Gamma]\} \\
\mathcal{G}_k[\Gamma, \alpha : \kappa] & = & \{\gamma[\alpha \leftarrow (\tau_1, (R_j)^{j \le k}, \tau_2)] \mid (\tau_1, (R_j)^{j \le k}, \tau_2) \in \mathcal{K}[\kappa]_{\gamma_1, \gamma_2} \wedge \gamma \in \mathcal{G}_k[\Gamma]\} \\
\mathcal{G}_k[\Gamma, (a_1 =_\tau a_2)] & = & \{\gamma \in \mathcal{G}_k[\Gamma] \mid (\vdash \gamma_1(a_1) \simeq \gamma_1(a_2)) \wedge (\vdash \gamma_2(a_1) \simeq \gamma_2(a_2))\} \\
\mathcal{E}_k[\tau]_\gamma & = & \{(a_1, a_2) \mid \forall i,\ \forall v_2,\ a_2 \mapsto_i v_2 \implies \exists v_1,\ a_1 \mapsto^* v_1 \wedge (v_1, v_2) \in \mathcal{V}_{k-i}[\tau]_\gamma\} \\
\mathcal{V}_k[\alpha]_\gamma & = & \gamma(\alpha) \\
\mathcal{V}_k[\tau_1 \sharp \tau_2]_\gamma & = & \mathcal{V}_k[\tau_1]_\gamma\ \mathcal{V}_k[\tau_2]_\gamma \\
\mathcal{V}_k[\tau \sharp u]_\gamma & = & \mathcal{V}_k[\tau]_\gamma\ (\gamma_1(u), \gamma_2(u)) \\
\mathcal{V}_k[\tau \sharp \diamond]_\gamma & = & \mathcal{V}_k[\tau]_\gamma\ \bullet \\
\mathcal{V}_k[\Lambda^\sharp(\alpha : \kappa).\ \tau]_\gamma & = & \lambda(\mathcal{R} \in \mathcal{K}[\kappa]_{\gamma_1, \gamma_2}).\ \mathcal{V}_k[\tau]_{\gamma[\alpha \leftarrow \mathcal{R}]} \\
\mathcal{V}_k[\lambda^\sharp(x : \kappa).\ \tau]_\gamma & = & \lambda((u_1, u_2) \in \mathsf{Term} \times \mathsf{Term}).\ \mathcal{V}_k[\tau]_{\gamma[x \leftarrow (u_1, u_2)]} \\
\mathcal{V}_k[\lambda^\sharp(\diamond : a_1 =_{\tau_2} a_2).\ \tau_1]_\gamma & = & \lambda(\bullet \in \mathbf{1}).\ \mathcal{V}_k[\tau_1]_\gamma \\
\mathcal{V}_k[\tau_1 \to \tau_2]_\gamma & = & \left\{ \begin{pmatrix} \mathsf{fix}\ (x : \tau_1' \to \tau_2')\ y.\ a_1, \\ \mathsf{fix}\ (x : \tau_1'' \to \tau_2'')\ y.\ a_2 \end{pmatrix} \middle| \begin{array}{l} \forall\, (j < k)\ (v_1, v_2) \in \mathcal{V}_j[\tau_1]_\gamma \implies \\ \begin{pmatrix} a_1[x \leftarrow (\mathsf{fix}\ (x : \tau_1' \to \tau_2')\ y.\ a_1), y \leftarrow v_1], \\ a_1[x \leftarrow (\mathsf{fix}\ (x : \tau_1'' \to \tau_2'')\ y.\ a_2), y \leftarrow v_2] \end{pmatrix} \in \mathcal{E}_j[\tau_2]_\gamma \end{array} \right\} \\
\mathcal{V}_k[\Pi(x : \tau_1).\ \tau_2]_\gamma & = & \left\{ \begin{pmatrix} \lambda^\sharp(x : \tau_1').\ a_1, \\ \lambda^\sharp(x : \tau_1'').\ a_2 \end{pmatrix} \middle| \begin{array}{l} \forall\, (j \le k)\ (u_1, u_2) \in \mathcal{E}_j[\tau_1]_\gamma \implies \\ (a_1[x \leftarrow u_1], a_2[x \leftarrow u_2]) \in \mathcal{E}_j[\tau_2]_{\gamma[x \leftarrow (u_1, u_2)]} \end{array} \right\} \\
\mathcal{V}_k[\Pi(\diamond : b_1 =_\tau b_2).\ \tau']_\gamma & = & \left\{ \begin{pmatrix} \lambda^\sharp(\diamond : \ldots).\ a_1, \\ \lambda^\sharp(\diamond : \ldots).\ a_2 \end{pmatrix} \middle| \begin{pmatrix} \emptyset \vdash \gamma_1(a_1) \simeq \gamma_1(a_2) \\ \wedge \quad \emptyset \vdash \gamma_2(a_1) \simeq \gamma_2(a_2) \end{pmatrix} \implies (a_1, a_2) \in \mathcal{E}_k[\tau']_\gamma \right\} \\
\mathcal{V}_k[\forall(\alpha : \mathsf{Typ})\ \tau]_\gamma & = & \left\{ \begin{pmatrix} \Lambda(\alpha : \mathsf{Typ}).\ u_1, \\ \Lambda(\alpha : \mathsf{Typ}).\ u_2 \end{pmatrix} \middle| \begin{array}{l} \forall\, ((\tau_1, (R_j)^{j \le k}, \tau_2) \in \mathcal{K}[\kappa]_{\gamma_1, \gamma_2}) \\ (u_1, u_2) \in \mathcal{V}_k[\tau]_{\gamma[\alpha \leftarrow (\tau_1, (R_j)^{j \le k}, \tau_2)]} \end{array} \right\} \\
\mathcal{V}_k[\forall^\sharp(\alpha : \kappa).\ \tau]_\gamma & = & \left\{ \begin{pmatrix} \Lambda^\sharp(\alpha : \kappa_1).\ a_1, \\ \Lambda^\sharp(\alpha : \kappa_2).\ a_2 \end{pmatrix} \middle| \begin{array}{l} \forall\, ((\tau_1, (R_j)^{j \le k}, \tau_2) \in \mathcal{K}[\kappa]_{\gamma_1, \gamma_2}) \\ (a_1, a_2) \in \mathcal{E}_k[\tau]_{\gamma[\alpha \leftarrow (\tau_1, (R_j)^{j \le k}, \tau_2)]} \end{array} \right\} \\
\mathcal{V}_k[\zeta\ (\tau_i)^i]_\gamma & = & \left\{ (d(v_j)^j, d(w_j)^j) \middle| \begin{array}{l} (d : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\ (\alpha_i)^i) \\ \wedge \quad \forall\, (j)\ (v_j, w_j) \in \mathcal{V}_k[\tau_j[\alpha_i \leftarrow \tau_i]^i]_\gamma \end{array} \right\}
\end{array}
$$

$$
\mathcal{V}_k[\mathsf{match}\ a\ \mathsf{with}\ (d_i(x_{ij})^j \to \tau_i)^i]_\gamma = \begin{cases} \mathcal{V}_k[\tau_j]_{\gamma[x_{ij} \leftarrow (v_j, v_j')]^j} & \text{if } \gamma_1(a) \mapsto_0 d_i(v_j)^j \wedge \gamma_2(a) \mapsto_0 d_i(v_j')^j \\ \emptyset & \text{otherwise} \end{cases}
$$

Fig. 22. Definition of the logical relation

The relation $\mathcal{E}_k[\tau]_\gamma$ is defined so that the left-hand side term terminates whenever the the right-hand side term, *i.e.* the left-hand side program terminates *more often*. In particular, every program is related to the never-terminating program at any type. This is not a problem: if we need to consider termination, we can use the reverse relation, where the left and right side are exchanged. This is what we will do on ornaments: we will first show that, for arbitrary patches, the ornamented program is equivalent but terminates less, and then, assuming the patches terminate, we show that the base program and the lifted program are linked by both the normal and the reverse relation.

Let us justify that this definition is well-founded. The interpretation of kinds is defined by structural induction on the kind. The interpretation of contexts is defined by structural induction on the context, and is well-founded as long as the relation on terms and values is defined. The interpretation of values (and terms) is defined by induction, first on the indices, then on the structure of the type. The case of datatypes is particular: then, the interpretation is defined by induction on the term. Only datatypes and arrows can appear in the type of a field of a constructor, and arrows decrease the index. Thus, the definition is well-founded.

We now prove that well-formed contexts and well-kinded types have a defined interpretation.

Lemma 6.10 (Well-kinded types, well-formed contexts have an interpretation). *Let Γ be a context.*

- *Suppose $\vdash \Gamma$. Then, for all k, $\mathcal{G}_k[\Gamma]$ is defined.*
- *Suppose $\Gamma \vdash \tau : \kappa$. Then, for all k and for all $\gamma \in \mathcal{G}_k[\Gamma]$, $\mathcal{V}_k[\tau]_\gamma$ is defined, and $\mathcal{V}_k[\tau]_\gamma \in \mathcal{K}[\kappa]_{\gamma_1,\gamma_2}$.*
- *Suppose $\Gamma \vdash \tau : \mathrm{Met}$. Then, for all k and for all $\gamma \in \mathcal{G}_k[\Gamma]$, $\mathcal{E}_k[\tau]_\gamma$ is defined.*

Proof. By mutual induction on the kinding and well-formedness relations. Use the previous lemma for K-Conv. The interpretations of relational types are formed between base types of the right kinds. It remains to check that the interpretation of types of base kinds are decreasing with k. This can be shown by the same induction that guarantees the induction is well-founded: all definitions using interpretations in contravariant position are explicitly made decreasing by quantifying on the rank. □

Lemma 6.11 (Substitution commutes with interpretation). *For all environments γ, index k, we have:*

$$\begin{array}{lcl}\mathcal{V}_k[\tau[\alpha \leftarrow \tau']]_\gamma & = & \mathcal{V}_k[\tau]_{\gamma[\alpha \leftarrow \mathcal{V}_k[\tau']_\gamma]} \\ \mathcal{V}_k[\tau[x \leftarrow u]]_\gamma & = & \mathcal{V}_k[\tau]_{\gamma[x \leftarrow (\gamma_1(u), \gamma_2(u))]} \\ \mathcal{K}[\kappa[\alpha \leftarrow \tau']]_{\gamma_1,\gamma_2} & = & \mathcal{K}[\kappa]_{\gamma_1[\alpha \leftarrow \gamma_1(\tau')], \gamma_2[\alpha \leftarrow \gamma_2(\tau')]} \\ \mathcal{K}[\kappa[x \leftarrow u]]_{\gamma_1,\gamma_2} & = & \mathcal{K}[\kappa]_{\gamma_1[x \leftarrow \gamma_1(u)], \gamma_2[x \leftarrow \gamma_2(u)]}\end{array}$$

Proof. By structural induction on τ, κ. □

Lemma 6.12 (Fundamental lemma).

- *Suppose $\Gamma \vdash a : \tau$. Then, for all k, $\gamma \in \mathcal{G}_k[\Gamma]$, $(\gamma_1(a), \gamma_2(a)) \in \mathcal{E}_k[\tau]_\gamma$.*
- *Suppose $\Gamma \vdash \tau_1 \simeq \tau_2$. Then, for all k, $\gamma \in \mathcal{G}_k[\Gamma]$, $\mathcal{V}_k[\tau_1]_\gamma = \mathcal{V}_k[\tau_2]_\gamma$.*

Proof. By induction on the structure of the relation, and on the typing or equality derivations.
For equality proofs:

- Reflexivity, transitivity and symmetry are immediate.
- For head-reduction, the only possible reduction is application.
- For C-Context, proceed by induction on the context then apply the inductive hypothesis (for types), or in the case of match use the fact that equal terms have the same head-constructor in empty environments (Lemma 5.48).
- For C-Split on a term u: apply the induction hypothesis on $\Gamma \vdash u : \zeta\ (\tau_i)^i$. We have: $(\gamma_1(u), \gamma_2(u)) \in \mathcal{E}_k[\zeta\ (\tau_i)^i]_\gamma$. Since $\gamma_1(u)$ and $\gamma_2(u)$ are closed, non-expansive terms, they reduce in 0 steps to values $(v_1, v_2) \in \mathcal{V}_k[\zeta\ (\tau_i)^i]_\gamma$ (this is a consequence of reflexivity for the logical relation on $\longrightarrow_\iota$, after $\longrightarrow_\sharp$ normalization). In particular, they have the same head-constructor and the fields of the constructors are related. We can then add the fields and the equality to the context, and apply the inductive hypothesis on the appropriate constructor.

For typing derivations, we will only examine the cases of Var, Conv, Fix, and App.

- The Var rule is:

$$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

Consider $\gamma \in \mathcal{G}_k[\Gamma]$. By definition, $(\gamma_1(x), \gamma_2(x)) \in \mathcal{V}_k[\Gamma]_\gamma$. Thus, $(\gamma_1(x), \gamma_2(x)) \in \mathcal{E}_k[\Gamma]_\gamma$.
- Consider the Conv rule:

$$\dfrac{\Gamma \vdash \tau_1 \simeq \tau_2 \qquad \Gamma \vdash a : \tau_1}{\Gamma \vdash a : \tau_2}\ \textsc{Conv}$$

Let $\gamma \in \mathcal{G}_k[\Gamma]$. By inductive hypothesis, $(\gamma_1(a), \gamma_2(a)) \in \mathcal{E}_k[\tau_1]_\gamma$, and $\mathcal{V}_k[\tau_1]_\gamma = \mathcal{V}_k[\tau_2]_\gamma$. Thus, $(\gamma_1(a), \gamma_2(a)) \in \mathcal{E}_k[\tau_2]_\gamma = \mathcal{E}_k[\tau_1]_\gamma$.

- Consider the Fix rule:

$$\frac{\Gamma, x : \tau_1 \to \tau_2, y : \tau_1 \vdash a : \tau_2}{\Gamma \vdash \mathsf{fix}\,(x : \tau_1 \to \tau_2)\; y.\, a : \tau_1 \to \tau_2}\;\textsc{Fix}$$

Consider $\gamma \in \mathcal{G}_k[\Gamma]$. We want to prove $(\mathsf{fix}\,(x : \gamma_1(\tau_1) \to \gamma_1(\tau_2)) y.\gamma_1(a), \mathsf{fix}\,(x : \gamma_2(\tau_1) \to \gamma_2(\tau_2)) y.\gamma_2(a)) \in \mathcal{V}_k[\tau_1 \to \tau_2]_\gamma$. Consider $j < k$, and $(v_1, v_2) \in \mathcal{V}_j[\tau_1]_\gamma$. We need to show:
$(\gamma_1(a)[x \leftarrow \mathsf{fix}\,(x : \gamma_1(\tau_1) \to \gamma_1(\tau_2))\; y.\, \gamma_1(a), y \leftarrow v_1]\, \gamma_1(a)[x \leftarrow \mathsf{fix}\,(x : \gamma_1(\tau_1) \to \gamma_1(\tau_2))\; y.\, \gamma_1(a), y \leftarrow v_1])$ $\in \mathcal{V}_j[\tau_2]_\gamma$ Note that by weakening, $\gamma \in \mathcal{G}_j[\Gamma]$. Moreover, by induction hypothesis at rank $j < k$, $(\mathsf{fix}\,(x : \gamma_1(\tau_1) \to \gamma_1(\tau_2))\; y.\, \gamma_1(a), \mathsf{fix}\,(x : \gamma_2(\tau_1) \to \gamma_2(\tau_2))\; y.\, \gamma_2(a)) \in \mathcal{V}_j[\tau_1 \to \tau_2]_\gamma$. Consider
$\gamma' = \gamma[x \leftarrow (\mathsf{fix}\,(x : \gamma_1(\tau_1) \to \gamma_1(\tau_2))\; y.\, \gamma_1(a),$
$\mathsf{fix}\,(x : \gamma_2(\tau_1) \to \gamma_2(\tau_2))\; y.\, \gamma_2(a)), y \leftarrow (v_1, v_2)]$. Then, $\gamma' \in \mathcal{G}_j[\Gamma, x : \tau_1 \to \tau_2, y : \tau_1]$. Thus, by induction hypothesis at rank $j < k$, $(\gamma_1'(a), \gamma_2'(a)) =$
$(\gamma_1(a)[x \leftarrow \mathsf{fix}\,(x : \gamma_1(\tau_1) \to \gamma_1(\tau_2))\; y.\, \gamma_1(a), y \leftarrow v_1], \gamma_2(a)[x \leftarrow \mathsf{fix}\,(x : \gamma_2(\tau_1) \to \gamma_2(\tau_2))\; y.\, \gamma_2(a), y \leftarrow v_2])$ $\in \mathcal{V}_j[\tau_2]_{\gamma'} = \mathcal{V}_j[\tau_2]_\gamma$.

- Consider the App rule:

$$\frac{\Gamma \vdash b : \tau_1 \qquad \Gamma \vdash a : \tau_1 \to \tau_2}{\Gamma \vdash a\; b : \tau_2}\;\textsc{App}$$

Let $\gamma \in \mathcal{G}_k[\Gamma]$. Suppose $\gamma_2(a)\; \gamma_2(b) \mapsto_i v_2$. We want to show that there exists v_1 such that $\gamma_1(a)\; \gamma_1(b) \mapsto^* v_1$ and $(v_1, v_2) \in \mathcal{V}_{k-i}[\tau_2]_\gamma$.

Since $\gamma_2(a)$ reduces to a value, there exists w_2, w_2' such that $\gamma_2(a) \mapsto_{i_1} w_2$, $\gamma_2(b) \mapsto_{i_1} w_2'$. By induction hypothesis on a and b, there exists values w_1, w_1' such that $(w_1, w_2) \in \mathcal{V}_{k-i_1}[\tau_1 \to \tau_2]_\gamma$ and $(w_1', w_2') \in \mathcal{V}_{k-i_2}[\tau_1]_\gamma$. We can apply the first property at rank $k - i_1 - i_2 - 1$: there exists $a_1', a_2', \tau_1', \tau_1''$ such that $w_1 = \mathsf{fix}\,(x : \tau_1' \to \tau_2')\; y.\, a_1'$ and $w_2 = \mathsf{fix}\,(x : \tau_1'' \to \tau_2'')\; y.\, a_2'$, and also $(a_1'[x \leftarrow \ldots, y \leftarrow w_1'], a_2'[x \leftarrow \ldots, y \leftarrow w_2']) \in \mathcal{E}_{k-i_1-i_2-1}[\tau_2]_\gamma$. Then, we have: $a_2'[x \leftarrow \ldots, y \leftarrow w_2'] \mapsto_{i3} v_2$ with $i = i_1 + i_2 + i_3 + 1$, and $a_1'[x \leftarrow \ldots, y \leftarrow w_1'] \mapsto^* v_1$. Thus, $(v_1, v_2) \in \mathcal{V}_{k-i}[\tau_2]_\gamma$.

□

6.5 Closure by biorthogonality

We want our relation to be compatible with substitution. We build a closure of our relation: essentially, the relation at a type relates all programs that cannot be distinguished by a context that does not distinguish programs we defined to be equivalent. To be well-typed, our notion of context must be restricted to only allow equal programs to be substituted. This is enough to show that it embeds contextual equivalence and substitution.

We will assume that there exists a type unit with a single value (). Consider two closed terms a_1 and a_2. We note $a_1 \precsim a_2$ if and only if a_1 and a_2 both have type unit, and if a_2 reduces to (), a_1 reduces to () too.

We will consider the relation $\mathcal{E}[\tau]_\gamma$ without indices as the limit of $\mathcal{E}_k[\tau]_\gamma$.

We can then define a relation on contexts. This relation must take equality into accounts: two unequal terms cannot necessarily be put in the same context, because the context might be dependent on the term we put in. Thus, our relation on contexts only compares contexts at terms equal to a given term.

Definition 6.13 (Relation on contexts). We note $(C_1, C_2) \in C[\tau \mid a_1, a_2]_\gamma$ iff:

- $\emptyset \vdash C_1[\emptyset \vdash a_1 : \gamma_1(\tau)] : \mathsf{unit}$ and $\emptyset \vdash C_2[\emptyset \vdash a_2 : \gamma_2(\tau)] : \mathsf{unit}$
- for all a_1', a_2' such that $\emptyset \vdash a_i' : \gamma_i(\tau)$, $(\emptyset \vdash a_i \simeq a_i')$ and $(a_1', a_2') \in \mathcal{E}[\tau]_\gamma$, we have $C_1[a_1] \precsim C_2[a_2]$.

From this relation on context we can define a closure of the relation:

Definition 6.14 (Closure of the logical relation). We note $(a_1, a_2) \in \mathcal{E}^2[\tau]_\gamma$ iff:

- $\emptyset \vdash a_1 : \gamma_1(\tau)$ and $\emptyset \vdash a_2 : \gamma_2(\tau)$
- for all $(C_1, C_2) \in C[\tau \mid a_1, a_2]_\gamma$, $C_1[a_1] \precsim C_2[a_2]$.

We obtain a relation that includes the previous relation, and allows substitution, contextual equivalence, etc. We introduce a notation that includes quantification on environments:

Lemma 6.15 (Inclusion). *Suppose $(a_1, a_2) \in \mathcal{E}_k[\tau]_\gamma$. Then $(a_1, a_2) \in \mathcal{E}^2[\tau]_\gamma$.*

Proof. Expand the definitions. □

Lemma 6.16 (Inclusion in contextual equivalence). *Suppose that for all $\gamma \in \mathcal{G}_k[\Gamma]$, $(a_1, a_2) \in \mathcal{E}^2[\tau]_\gamma$. Then, for all contexts C such that $\emptyset \vdash C[\Gamma \vdash a_1 : \tau] : \mathtt{unit}$ and $\emptyset \vdash C[\Gamma \vdash a_2 : \tau] : \mathtt{unit}$, we have $C[a_1] \precsim C[a_2]$.*

Proof. By induction on the context. As in the proof of the fundamental lemma, each typing rule induces an equivalent deduction rule for the logical relation. For example, the Let-Poly rule becomes (assuming the typing conditions are met): if $(a_1, a_2) \in \mathcal{E}^2[\tau_0]_\gamma$, and for all $(v_1, v_2) \in \mathcal{E}[\tau_0]_\gamma$ such that $\emptyset \vdash a_i \simeq v_i$, we have $(b_1[x \leftarrow (v_1, v_2)], b_2[x \leftarrow (v_1, v_2)]) \in \mathcal{E}^2[\tau]_{\gamma[x \leftarrow (v_1, v_2)]}$, then $(\mathtt{let}\ x = a_1\ \mathtt{in}\ b_1, \mathtt{let}\ x = a_2\ \mathtt{in}\ b_2) \in \mathcal{E}^2[\tau]_\gamma$. We use the induction hypothesis on the subterm that contains the hole, and the fundamental lemma for the other subterms. □

Lemma 6.17 (Contextual equivalence implies relation). *Consider a_1, a_2 such that $\Gamma \vdash a_i : \tau$ and $\Gamma \vdash a_1 \simeq a_2$. Moreover, suppose that they are contextually equivalent: if $\emptyset \vdash C[\Gamma \vdash a_1 : \tau] : \mathtt{unit}$, then $C[a_1] \precsim C[a_2]$ and $C[a_2] \precsim C[a_1]$. Then, for all $\gamma \in \mathcal{G}_k[\Gamma]$, $(\gamma(a_1), \gamma(a_2)) \in \mathcal{E}^2[\tau]_\gamma$.*

Proof. We have $(\gamma(a_1), \gamma(a_1)) \in \mathcal{E}_k[\tau]_\gamma$. Consider C_1, C_2 such that $C_1[\gamma(a_1)] \precsim C_2[\gamma(a_1)]$. Then, by contextual equivalence, we can substitute $\gamma(a_1)$ by $\gamma(a_2)$ in the right-hand side. □

Lemma 6.18 (Reduction). *Suppose $a_1 \longrightarrow a_2$, and $C[a_1], C[a_2]$ have the same type τ in the empty environment. Then, $(C[a_1], C[a_2]) \in \mathcal{E}^2[\tau]_\gamma$.*

Proof. Use Lemma 6.16, add the context C, use Lemma 6.17. □

These definitions give a restricted form of transitivity. Full transitivity may hold but is not easy to prove: essentially, it requires inventing out of thin air a context "between" two contexts, that is related to the first one in a specific environment and to the second one in another specific environment. If we restrict ourselves to the *sides* of an environment, then we can simply reuse the same context. a *side* of an environment is, in spirit, a relational version of the left and right environment γ_1 and γ_2.

Definition 6.19 (Sides of an environment). Consider an environment γ. Its left and right sides $\epsilon_1\gamma$ and $\epsilon_2\gamma$ are defined as follows:

- $\epsilon_1\gamma(x) = (\gamma_1(x), \gamma_1(x))$ and $\epsilon_2\gamma(x) = (\gamma_2(x), \gamma_2(x))$;
- $\epsilon_1\gamma(\alpha) = \{(v_1, v_2) \mid \forall w,\ (v_1, w) \in \gamma(\alpha) \Leftrightarrow (v_1, w) \in \gamma(\alpha)\}$ and $\epsilon_2\gamma(\alpha) = \{(w_1, w_2) \mid \forall v,\ (v, w_1) \in \gamma(\alpha) \Leftrightarrow (v, w_2) \in \gamma(\alpha)\}$ if α is interpreted by a relation;
- the sides of interpretations of types of higher-order kinds are interpreted pointwise on the base kinds.

Lemma 6.20 (Properties of sides).
- *If an environment γ verifies an equality $a_1 =_\tau a_2$, then its sides respect this equality*
- *If $(a_1, a_2) \in \mathcal{E}_k[\tau]_\gamma$, then $(a_i, a_i) \in \mathcal{E}_k[\tau]_{\epsilon_i\gamma}$.*
- *If $\gamma \in \mathcal{G}_k[\Gamma]$, then $\epsilon_1\gamma \in \mathcal{G}_k[\Gamma]$ and $\epsilon_2\gamma \in \mathcal{G}_k[\Gamma]$.*

Proof. By induction on the structure of the logical relation. □

Lemma 6.21 (Side-transitivity). *Consider an environment γ, and suppose:*

- $(a_0, a_1) \in \mathcal{E}^2[\tau]_{\epsilon_1\gamma}$ *and* $\emptyset \vdash a_0 \simeq a_1$*;*
- $(a_1, a_2) \in \mathcal{E}^2[\tau]_{\gamma}$*;*
- $(a_2, a_3) \in \mathcal{E}^2[\tau]_{\epsilon_2\gamma}$ *and* $\emptyset \vdash a_2 \simeq a_3$*.*

Then, $(a_0, a_3) \in \mathcal{E}^2[\tau]_{\gamma}$*.*

Proof. Consider $(C_0, C_3) \in C[\tau \mid a_0, a_3]_{\gamma}$. Then, we have $(C_0, C_0) \in C[\tau \mid a_0, a_1]_{\epsilon_1\gamma}$, and $(C_3, C_3) \in C[\tau \mid a_2, a_3]_{\epsilon_1\gamma}$, and $(C_0, C_3) \in C[\tau \mid a_1, a_3]_{\epsilon_1\gamma}$. Conclude by transitivity of $\precsim$. □

Lemma 6.22 (Equality implies relation). *Suppose* $\Gamma \vdash a_1 : \tau$ *and* $\Gamma \vdash a_1 \simeq a_2$*. Then, for all* $\gamma \in \mathcal{G}_k[\Gamma]$, $(\gamma(a_1), \gamma(a_2)) \in \mathcal{E}^2[\tau]_{\gamma}$*.*

Proof. By induction on an equality derivation. Since the relation is not symmetric, we a stronger result by induction on derivations: if $\Gamma \vdash a_1 \simeq a_2$, then for all $\gamma \in \mathcal{G}_k[\Gamma]$, $(\gamma(a_1), \gamma(a_2)) \in \mathcal{E}^2[\tau]_{\gamma}$ and $(\gamma(a_2), \gamma(a_1)) \in \mathcal{E}^2[\tau]_{\gamma}$. Then, each rule translates to one of the previous lemmas. □

7 Simplification from *e*ML to ML

Consider an ML environment Γ, an ML type τ, and an *e*ML term a, such that $\Gamma \vdash a : \tau$ hold in *e*ML. Our goal is to find a term a' such that $\Gamma \vdash a' : \tau$ holds in ML and that is equivalent to a: $\Gamma \vdash a \simeq a'$. This is a good enough definition, since all terms that are provably equal are related. Restricting the typing derivation of a' to ML introduces two constraints. First, no type-level pattern matching can appear in the types in the term. Then, we must ensure that the term admits a typing derivation that does not involve conversion and pattern matching.

The types appearing in the terms are only of kind Typ, thus they cannot contain a type-level pattern matching, and explicitly-typed bindings cannot introduce in the context a variable whose kind is not Typ. There remains the case of let bindings: they can introduce a variable of kind Sch. We can get more information on these types by the following lemma, that proves that "stuck" types such as match $f\ x$ with $(di(y_j)^j \to \tau_i)^i$ do not contain any term:

Lemma 7.1 (Match trees). *A type τ is said to be a* match tree *if the judgment* $\Gamma \vdash \tau$ tree *defined below holds:*

$$\textsc{Tree-Scheme}\quad \frac{\Gamma \vdash \tau : \mathsf{Sch}}{\Gamma \vdash \tau\ \mathsf{tree}}
\qquad
\textsc{Tree-Match}\quad \frac{\begin{array}{c}(d_i : \forall(\alpha_k)^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \\ \Gamma \vdash a : \zeta\ (\tau_k)^k \qquad \left(\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, \qquad a =_{\zeta\ (\tau_k)^k} d_i(\tau_{ik})^k (x_{ij})^j \vdash \tau'_i\ \mathsf{tree}\right)^i\end{array}}{\Gamma \vdash \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_{ik})^k(x_{ij})^j \to \tau'_i)^i\ \mathsf{tree}}$$

Suppose all types of variables in Γ *are match trees. If* $\Gamma \vdash a : \tau$*, there exists a type* τ' *such that* $\Gamma \vdash \tau \simeq \tau'$ *and* $\Gamma \vdash \tau'$ tree.

Proof. By induction on the typing derivation of a. Consider the different rules:

- By hypothesis on Γ, the output of Var is a match tree.
- The outputs of rules TAbs, TApp, Fix, App, and Con have kind Sch, thus are match trees.
- Consider a conversion Conv from τ to τ'. If τ is equal to a match tree, then τ' is equal to a match tree by transitivity.
- For rule Let-Poly:

$$\textsc{Let-Poly}\quad \frac{\begin{array}{cc} & \Gamma \vdash \tau : \mathsf{Sch} \\ \Gamma \vdash u : \tau & \Gamma, x : \tau, (x =_\tau u) \vdash b : \tau'\end{array}}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ b : \tau'}$$

By induction hypothesis, the type of u is equal in Γ to a match tree. Thus we can assume that τ is a match tree. Then, all types in $\Gamma, x : \tau, (x =_\tau u)$ are match trees. There exists τ'' such that $\Gamma, x : \tau, (x =_\tau u) \vdash \tau''$ tree and $\Gamma, x : \tau, (x =_\tau u) \vdash \tau' \simeq \tau''$. We can substitute using $x =_\tau u$ (by Lemma 5.15), and we obtain $\Gamma, (u_\tau u) \vdash \tau' \simeq \tau''[x \leftarrow u]$ and $\Gamma, (u =_\tau u) \vdash \tau''$ tree. All uses of the equality can be replace by C-Refl so we can eliminate it.

- For rule Let-Mono:

$$\text{Let-Mono}\quad \frac{\Gamma \vdash a : \tau \qquad \begin{array}{l}\Gamma \vdash \tau : \mathsf{Typ}\\ \Gamma, x : \tau, (x =_\tau a) \vdash b : \tau'\end{array}}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau'}$$

By induction hypothesis, the type of a is equal in Γ to a match tree. Thus we can assume that τ is a match tree. Then, all types in $\Gamma, x : \tau, (x =_\tau a)$ are match trees. There exists τ'' such that $\Gamma, x : \tau, (x =_\tau a) \vdash \tau''$ tree and $\Gamma, x : \tau, (x =_\tau a) \vdash \tau' \simeq \tau''$. We can assume a is expansive (otherwise we can use the same reasoning as in the last case). Then, the equality is useless by Lemma 5.20. Moreover, since τ has kind Typ, there is a value v in τ. Then, $\Gamma \vdash \tau''[x \leftarrow v]$ tree and $\Gamma \vdash \tau' \simeq \tau''[x \leftarrow v]$.

- For rule Match:

$$\text{Match}\quad \frac{\Gamma \vdash \tau : \mathsf{Sch} \qquad (d_i : \forall (\alpha_k)^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k)^i \qquad \Gamma \vdash a : \zeta\ (\tau_k)^k \qquad \left(\begin{array}{l}\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j,\\ a =_{\zeta\ (\tau_k)^k} d_i(\tau_{ij})^k (x_{ij})^j \ \vdash b_i : \tau\end{array}\right)^i}{\Gamma \vdash \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_{ij})^k (x_{ij})^j \to b_i)^i : \tau}$$

Proceed as for Let in the case where we match on an expansive term a: we can use the default value for all the bound variables in one branch and get the equality we need. If $a = u$ is non-expansive, use the induction hypothesis on each branch: there exists $(\tau_i)^i$ such that $(\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (u =_{\zeta\ (\tau_k)^k} d_i(\tau_{ij})^k (x_{ij})^j) \vdash \tau \simeq \tau_i)^i$ and $(\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (u =_{\zeta\ (\tau_k)^k} d_i(\tau_{ij})^k (x_{ij})^j) \vdash \tau_i \text{ tree})^i$. Then, consider $\tau' = \mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_{ij})^k (x_{ij})^j \to \tau_i)^i$. We have $\Gamma \vdash \tau' \simeq \mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_{ij})^k (x_{ij})^j \to \tau)^i$ (by applying the previous equality in each branch), and $\Gamma \vdash \mathsf{match}\ u\ \mathsf{with}\ (d_i(\tau_{ij})^k (x_{ij})^j \to \tau)^i \simeq \tau$ by C-Split and C-Red-Iota for each case. Moreover, $\Gamma \vdash \tau'$ tree by Tree-Match.

□

From this lemma, we can deduce that there exists a typing derivation where the type of all variables bound in let is a match tree. The pattern matching in the types can then be eliminated by lifting the pattern-matching outside of the let, as in $\longmapsto_t$ on Figure 23. This transformation is well-typed, and the terms are equal (the equality can be proved by case-splitting on u). Moreover, it strictly decreases the number of match ... with ... in the types of let-bindings. Thus we can apply it until we obtain a term with a derivation where all bindings are of kind Typ.

In the new derivation, no variable in context has a match type. We can transform the derivation such that all conversions are between ML types.

Lemma 7.2. *Suppose Γ is a context where all variables have a ML type. Suppose $\Gamma \vdash a : \tau$, where τ is a ML type and no variables are introduced with a non-ML type in the main type derivation. Then, there exists a derivation of $\Gamma \vdash a : \tau$ where all conversions of the main type derivation are between ML types.*

Proof. We proceed by induction, pushing the conversions in the term until they meet a syntactic construction that is not a match or a let. If we encounter a conversion, we combine it by transitivity with the conversion we are currently pushing. □

$$
\begin{array}{rcl}
\mathsf{let}\ (x : \mathsf{match}\ u\ \mathsf{with}\ (d_i\,\overline{\tau}\,(x_{ij})^j \to \tau_i)^i) = a\ \mathsf{in}\ b & \longmapsto_t & \mathsf{match}\ u\ \mathsf{with}\ (d_i\,\overline{\tau}\,(x_{ij})^j \to \mathsf{let}\ (x : \tau_i) = a\ \mathsf{in}\ b)^i \\
\mathsf{match}\ d_j\,\overline{\tau_j}\,(v_i)^i\ \mathsf{with}\ (d_j\,\overline{\tau_j}\,(x_{ji})^i \to a_j)^j & \longmapsto & a_j[x_{ij} \leftarrow v_i]^i \\
\mathsf{let}\ x = u\ \mathsf{in}\ b & \longmapsto & b[x \leftarrow u] \\
\left(\begin{array}{r}\mathsf{match}\ (\mathsf{match}\ u\ \mathsf{with}\ (d_k\,\overline{\sigma}\,(x_{kj})^j \to a_k)^k) \\ \mathsf{with}\ (d_i\,\overline{\tau}\,(x_{ij})^j \to b_i)^i\end{array}\right) & \longmapsto & \left(\begin{array}{l}\mathsf{match}\ u\ \mathsf{with}\ (d_k\,\overline{\sigma}\,(x_{kj})^j \to \\ \qquad (\mathsf{match}\ a_k\ \mathsf{with}\ (d_i\,\overline{\tau}\,(x_{ij})^j \to b_i)^i)^k\end{array}\right)
\end{array}
$$

Fig. 23. Match and let lifting

We know (by soundness) that all equalities between ML types are either trivial (*i.e.* between two identical types) or are used in a branch of the program that will never be run (otherwise, it would provoke an error). In order to translate the program to ML, we must eliminate these branches from the program. There are two possible approaches: we could extend ML with an equivalent of `assert false` and insert it in the unreachable branches, but the fact that the program executes without error would not be guaranteed by the type system anymore, and could be broken by subsequent manual modification to the code. The other approach is to transform the program to eliminate the unreachable branches altogether. This sometimes requires introducing extra pattern matchings and duplicating code. The downside to this approach is that in some cases the term could grow exponentially. This blowup can be limited by only doing the expansions that are strictly necessary.

The transformations of Figure 23 all preserve types and equality. All let bindings and pattern matchings that can be reduced by $\longrightarrow_\iota$ are reduced. When a pattern matching matches on the result of another pattern matching, the inner pattern matching is lifted around the formerly outer pattern matching. These transformations preserve the types and equality. These transformations terminate. After applying them, all pattern matching is done either on a variable, or on an expansive term.

We first show that, in inhabited environments, all conversions between ML types are trivial.

Definition 7.3 (Inhabited typing environment). A typing environment Γ is *inhabited* if there exists an environment γ mapping type variables to ground types and term variables to terms such that $\vdash \gamma : \Gamma$, that is, $\vdash \gamma(x) : \gamma(\tau)$ for every binding $x : \tau$ in Γ.

Lemma 7.4 (ML conversions are trivial). *If $\Gamma \vdash \tau_1 \simeq \tau_2$ in eML where Γ is inhabited and τ_1 and τ_2 are ML types, then $\tau_1 = \tau_2$.*

Proof. We only need to prove that, for ML types, if $\mathbf{E}[\tau_1]_\gamma = \mathbf{E}[\tau_2]_\gamma$, then $\tau_1 = \tau_2$. This is easy by induction. For universal quantification, instantiate with a unique type. □

Lemma 7.5 (ML conversions in context are trivial). *Consider an inhabited environment Γ_0, and assume $\Gamma_0 \vdash C[\Gamma \vdash a : \tau] : \tau'$. Suppose that Γ does not contain any equality. Then, if $\Gamma \vdash \tau_1 \simeq \tau_2$ and τ_1, τ_2 are ML types, $\tau_1 = \tau_2$.*

Proof. We only have to show how to construct an appropriate environment γ. We proceed by induction on the context, examining the introductions in the environment.

- If we introduce a variable x of type τ, and τ has kind Typ, it is either a function type or a datatype, thus is inhabited: for datatypes, we use the hypothesis made at the start that all datatypes are inhabited, and $\mathsf{fix}\ (x : \tau_1 \to \tau_2)\ y.\ x\ y$ is a function of type $\tau_1 \to \tau_2$.

  Inhabitation of all datatypes is necessary here because we are doing a semantic proof. If we were doing a syntactic proof, we would have to change our hypothesis from inhabitability of Γ to something similar to $\Gamma \vdash \simeq$.

- If we introduce a variable x of type scheme σ of kind σ that is not of kind Typ, the introducing form is necessarily a polymorphic let. Let us call γ the model until now (of Γ). Then, we bind x to a non-expansive term u, and $\Gamma \vdash u : \gamma(\sigma)$. Thus, $\gamma[x \leftarrow u]$ is a model of Γ.
- If we introduce a type variable α, we can instantiate it with a fresh type ζ_α. Then, $\tau_1 = \tau_2$ if and only if $\tau_1[\alpha \leftarrow \zeta_\alpha] = \tau_2[\alpha \leftarrow \zeta_\alpha]$. □

Then, we transform an *e*ML term into an equivalent ML term (*i.e.* where all conversions have been removed), by removing all absurd branches, typing it without equalities, and removing the conversions since they must be trivial.

Lemma 7.6 (Conversion elimination). *Consider an inhabited environment Γ. Assume $\Gamma \vdash a : \tau$ in eML and every pattern matching in a is on a variable or an expansive term and all variables introduced in the environment during the main type derivation have a type in* Sch. *Then, there exists a' such that $\emptyset \vdash a \simeq a'$ and $\emptyset \vdash a' : \tau$ in ML.*

Proof. We generalize to an environment Γ: suppose $\Gamma \vdash a : \tau$ and suppose we have a substitution γ of term variables with non-expansive terms without match or let (*i.e.* values with variables). Moreover, suppose that all substitutions in γ are equalities provable in Γ, and $\gamma(\Gamma)$ only has trivial equalities (*i.e.* provable by reflexivity) or useless equalities (*i.e.* involving an expansive term). Removing these equalities, we obtain Γ'. Then, there exists a' such that $\Gamma' \vdash a' : \tau$ in ML (the type τ is in ML, and, in particular, does not reference term variables from Γ), and $\Gamma' \vdash \gamma(a) \simeq a'$. We will also need to suppose that there is a context such that $\emptyset \vdash C[\Gamma' \vdash a : \tau] : \tau_e$.

We prove this result by induction on the typing rules. We will examine the two interesting cases: the conversion rule and the pattern matching rule.

- For Conv:

$$\textsc{Conv}\quad \frac{\Gamma \vdash \tau \simeq \tau' \qquad \Gamma \vdash a : \tau}{\Gamma \vdash a : \tau}$$

  By hypothesis, τ' is a ML type. Then, $\Gamma' \vdash \gamma(\tau) \simeq \gamma(\tau')$. Then, use Lemma 7.4 in the context C: $\tau = \tau'$ and the conversion can be eliminated. Then, apply the induction hypothesis in the same context C with the substitution γ.
- For Match:

$$\textsc{Match}\quad \frac{\begin{array}{c}\Gamma \vdash \tau : \mathsf{Sch} \qquad \left(d_i : \forall (\alpha_k)^k\ (\tau_{ij})^j \to \zeta\ (\alpha_k)^k\right)^i \\ \Gamma \vdash a : \zeta\ (\tau_k)^k \quad \left(\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, \qquad (a =_{\zeta\ (\tau_k)^k} d_i(\tau_{ij})^k(x_{ij})^j) \vdash b_i : \tau\right)^i\end{array}}{\Gamma \vdash \mathsf{match}\ a\ \mathsf{with}\ (d_i(\tau_{ij})^k(x_{ij})^j \to b_i)^i : \tau}$$

  If a is expansive, the equality introduced is useless, and we can continue typing in the new context. Consider the case where a is non-expansive. Then, a is necessarily a variable x. Consider $\gamma(x)$.
  - If $\gamma(x) = y$, apply the induction hypothesis in each branch, with $\gamma' = \gamma[y \leftarrow d_i(\tau_{ij})^k(x_{ij})^j]$, adding $\mathsf{match}\ y\ \mathsf{with}\ \ldots$ to the context.
  - If $\gamma(x) = d_i(\tau_{ik})^k(u_{ij})^j$, we can substitute and reduce the pattern matching. Then, apply the induction hypothesis on the branch b_i, with $\gamma[x_{ij} \leftarrow u_{ij}]^j$.
  - The other cases are excluded by typing in the (equality-free) environment Γ'.

We can conclude: in the empty environment, no equalities involving variables are provable, thus γ is the identity and $\Gamma' = \Gamma = \emptyset$. □

Thus we prove that elimination is possible. The transformations we apply preserve the equality judgment of *e*ML, thus the *e*ML term and the ML term obtained after the transformation are equivalent for the logical relation:

Theorem 7.7 (Match elimination). *If $\Gamma \vdash a : \tau$ in eML where Γ is inhabited and all program variables have an ML type and τ is an ML type, then there exists an ML term a' such that $\Gamma \vdash a \simeq a'$ and $\Gamma \vdash a' : \tau$. Moreover, we have $\Gamma' \vdash a' : \tau$ in ML when Γ' is obtain from Γ by removing all equality assumptions.*

The restriction to inhabited typing environment is not a significant problem in practice, as we expect the initial environment to be inhabited.

Proof. Apply the transformations $\longmapsto_t$ and $\longmapsto$ described in this section. Transform the derivation so that all conversions are between ML types with Lemma 7.2. Eliminate the conversions using Lemma 7.6. □

8 Encoding ornaments

We now consider how ornaments are described and represented inside the system. This section bridges the gap between *m*ML, a language for meta-programming that doesn't have any notion of ornament, and the interface presented to the user for ornamentation. We need both a definition of the base datatype ornaments, and the higher-order functional ornaments that can be built from them.

As a running example, we consider the ornament natlist α from natural numbers to lists defined as:

$$\text{type ornament natlist } \alpha : \text{nat} \to \text{list } \alpha \text{ with Z} \to \text{Nil} \mid \text{S } w \to \text{Cons } (\_, w)$$

The ornament natlist α defines, for all types α, a relation between values of its *base* type nat which we write $(\text{natlist } \alpha)^-$ and its *lifted* type list α which we write $(\text{natlist } \alpha)^+$: the first clause says that Z is related to Nil; the second clause says that if w_- related to w_+, then S w_- is related to Cons (v, w_+) for any v. As a notation shortcut, the variables w_- and w_+ are identified in the definition above.

A higher-order ornament natlist $\tau \to$ natlist τ, say ω, relates two functions f_- of type ω^- equal to nat $\to$ nat and f_+ of type ω^+ equal to list $\tau \to$ list τ when for related inputs v_- and v_+, the outputs $f_-\ v_-$ and $f_+\ v_+$ are related.

We formalize this idea by defining a family of *ornament types* corresponding to the ornamentation definitions given by the user and giving them an interpretation in the logical relation. Then, we say that one function is a lifting of another if they are related at the desired ornament type. This relation also gives us the reasoning tool to establish the ornamentation relation between the base type and the lifted type.

The syntax of ornament types, written ω, mirrors the syntax of types:

$$\chi \ ::= \ \text{natlist} \mid \ldots \qquad\qquad \omega \ ::= \ \varphi \mid \chi\,(\omega)^i \mid \zeta\,(\omega)^i \mid \omega \to \omega$$

An ornament type may be an ornament variable φ; a *base* ornament χ (we consider the problem of defining base ornaments below); a higher-order ornament $\omega_1 \to \omega_2$. or an *identity* ornament $\zeta\,(\omega)^i$, which is automatically defined for any datatype of the same name (ω_i indicates how the i-th type argument of the datatype is ornamented).

An ornament type ω is interpreted as a relation between terms of type ω^- and ω^+ (the projection is detailed on Figure 24). For example, the ornament list (natlist nat) describes the relation between lists whose elements have been ornamented using the ornament natlist nat.

We will define in the next section how to interpret the base ornaments χ, and focus here on the interpretation of higher-order ornaments $\omega_1 \to \omega_2$ and identity ornaments $\zeta\,(\omega_i)^i$.

The interpretation we want for higher-order ornaments is as functions sending arguments related by ornamentation to results related by ornamentation. But this is exactly what the interpretation of the arrow *type* $\tau_1 \to \tau_2$ gives us, if we replace the types τ_1 and τ_2 by ornament types $\omega_1 \to \omega_2$. Thus, we do not have to define a new interpretation for higher-order ornament, it is already included in the logical relation. For this reason, we will use interchangeably (when talking about the logical relation) the function arrow and the ornament arrow.

We have the same phenomenon for the identity ornament: constructors are related at the identity ornament if their arguments are related. Once more, we can simply take the interpretation of a datatype $\zeta\,(\tau_i)^i$ and, by replacing the type parameters $(\tau_i)^i$ by ornament parameters $(\omega_i)^i$, reinterpret it as an interpretation of the identity ornament. We will show that this choice is coherent by presenting a syntactic version of the identity ornament and showing it is well-behaved with respect to its interpretation.

Finally, ornament variables must be interpreted by getting the corresponding relation in the relational environment. This is exactly the interpretation of a *type* variable.

Thus, the common subset between types and ornament specifications can be identified, because the interpretations are the same. This property will play a key role in the instantiation: from a relation at a type, we will deduce, by proving the correct instantiation, a relation at an ornament.

8.1 Defining datatype ornaments

We assume that all datatypes defined in the language are regular and at most single recursive. The extension to families of mutually-recursive datatypes is straightforward, but makes the notations significantly heavier. We do not treat the problem of non-regular datatypes. Then, each regular datatype $\zeta\ (\alpha_i)^i$ is defined by a family of constructors:

$$(d_j : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_{kj})^j \to \zeta\ (\alpha_i)^i)^k$$

where ζ may occur recursively in τ_{kj} but only as $\zeta\ (\alpha_i)^i$. Let $\hat{\alpha}$ be a fresh variable and $\hat{\tau}_{kj}$ be τ_{kj} where all occurrences of $\zeta\ (\alpha_i)^i$ have been replaced by $\hat{\alpha}$. We define the skeleton of ζ as the new non-recursive datatype[7] $\hat{\zeta}(\alpha_i)^i\,\hat{\alpha}$ with the family of constructors:

$$(\hat{d}_k : \forall(\alpha_i : \mathsf{Typ})^i\ \forall(\hat{\alpha} : \mathsf{Typ})\ (\hat{\tau}_{kj})^j \to \hat{\zeta}\ (\alpha_i)^i\hat{\alpha})^k$$

By construction the types $\zeta\ \overline{\tau}$ and $\hat{\zeta}\ \overline{\tau}\ \ (\zeta\ \overline{\tau})$ are isomorphic.

Ornament definitions match a pattern in one datatype to a pattern in another datatype. We allow deep pattern matching: the patterns are not limited to matching on only one level of constructors, but can be nested. Additionally, we allow wildcard patterns _ that match anything, alternative patterns $P \mid Q$ that match terms that match either P or Q, and the null pattern $\emptyset$ that does not match anything. We write deep pattern matching the same as shallow pattern matching, with the understanding that it is implicitly desugared to shallow pattern matching.

In general, an ornament definition is of the form:

$$\mathsf{type\ ornament}\ \chi\ (\alpha_j)^j : \zeta\ (\tau_k)^k \to \tau_+\ \mathsf{with}\ (P_i \to Q_i)^i$$

with χ the name of the datatype ornament, $\zeta\ (\tau_k)^k$ the base type, say τ_- for short, and τ_+ the lifted type. The meaning and restrictions on ornament definitions are easier to define by referencing the skeleton of the base type. We define the *skeleton patterns* $(\hat{P}_i)^i$ obtained from $(P_i)^i$ by replacing the head constructor d (of ζ) by $\hat{d}$. If one of the pattern P_i does not have a head constructor, the ornament definition is invalid.

The patterns $\hat{P}_i$ (or equivalently P_i) must be expressions *i.e.* in the sublanguage consisting only of variables and data constructors. The family of skeleton patterns $(\hat{P}_i)^i$ must form an exhaustive partition of the type $\forall(\hat{\alpha} : \mathsf{Typ})\ \hat{\zeta}\ (\tau_k)^k\hat{\alpha}$. The universal quantification ensures that they only match the recursive occurrences of τ_- with variables. This is necessary to be able to ornament incrementally: during ornamentation, we will use these patterns to match terms where the recursive variable of the skeleton is instantiated with τ_+ instead of τ_-. The patterns $(Q_i)^i$ must form a partition of a subset of τ_+. The free variables in $\hat{P}_i$ and Q_i must be the same and their types must correspond: if x matches a value of type σ in $\hat{P}_i$, it must match a value of type $\sigma[\hat{\alpha} \leftarrow \tau_+]$ in Q_i (the recursive part $\hat{\alpha}$ is instantiated with τ_+, the type of the variable in P_i is $\sigma[\hat{\alpha} \leftarrow \tau_-]$).

We define the meaning of a user-provided ornament by adding its interpretation to the logical relation on *m*ML. The interpretation is the union of the relations defined by each clause of the ornament. For each clause, the

[7] For convenience we treat $\hat{\zeta}$ as curried: we write $\hat{\zeta}(\alpha_i)^i\,\hat{\alpha}$ rather than $\hat{\zeta}((\alpha_i)^i, \hat{\alpha})$.

values of the variables must be related at the appropriate type. Since the pattern on the left is also an expression, the value on the left is uniquely defined. The pattern on the right can still represent a set of different values (none, one, or many, depending on whether the empty pattern, an or-pattern or a wildcard was used). We define a function $\llparenthesis \_ \rrparenthesis$ associating to a pattern this set of values.

$$\llparenthesis (\_ : \sigma) \rrparenthesis = \mathsf{Term} \qquad \llparenthesis P \mid Q \rrparenthesis = \llparenthesis P \rrparenthesis \cup \llparenthesis Q \rrparenthesis \qquad \llparenthesis d(\tau_k)^k (P_i)^i \rrparenthesis = d(\tau_k)^k (\llparenthesis P_i \rrparenthesis)^i$$

Assuming that $(x_\ell)^\ell$ are the bound variables, at types $(\tau_\ell)^\ell$, in the skeleton $\hat{P}_i$ of the left-hand side of the clause i, the interpretation is:

$$\mathcal{V}_p[\chi\,(\omega_j)^j]_\gamma = \bigcup_i \left\{ (P_i[x_\ell \leftarrow v_{\ell-}]^\ell, v_+) \;\middle|\; v_+ \in \llparenthesis Q_i[x_\ell \leftarrow v_{\ell+}]^\ell \rrparenthesis \wedge \forall l, (v_{\ell-}, v_{\ell+}) \in \mathcal{V}_p[\tau_\ell[\hat{\alpha} \leftarrow \chi\,(\omega_j)^j]]_\gamma \right\}$$

The key point of the definition is that we instantiate the variable $\hat{\alpha}$ recursively with our ornament so the recursive parts are correctly ornamented.

For example, on natlist, we get the following definition (omitting the typing conditions):

$$\mathcal{V}_k[\mathsf{natlist}\ \tau]_\gamma = \{(\mathsf{Z}, \mathsf{Nil})\} \cup \left\{ (\mathsf{S}(v_-),\ \mathsf{Cons}(\_, v_+)) \;\mid\; (v_-, v_+) \in \mathcal{V}_k[\mathsf{natlist}\ \tau]_\gamma \right\}$$

8.2 Encoding ornaments in *m*ML

We now describe the encoding of datatype ornaments in *m*ML. We consider a specific instance $\chi\,(\omega_j)^j$ of the datatype ornament χ. The lifted type $\sigma_{\chi\,(\omega_j)^j}$, which we abbreviate as σ, is $\tau_+[\alpha_j \leftarrow \omega_j^+]^j$. We write[8] $\hat{\tau}_-$ for the type $\hat{\zeta}\ (\tau_k[\alpha_j \leftarrow \omega_j^+]^j)^k \sigma$ of the skeleton where the recursive parts and the type parameters have already been lifted. The ornament is encoded as a quadruple $(\sigma, \delta, \mathsf{proj}, \mathsf{cstr})$ where $\sigma : \mathsf{Typ}$ is the lifted type; δ is the *extension*, a type-level function describing the information that needs to be added; and proj and cstr are the projection and construction functions shown in §3. More precisely, the projection function proj from the lifted type to the skeleton has type $\Pi(x : \sigma).\ \hat{\tau}_-$ and the construction function cstr has type $\Pi(x : \hat{\tau}_-).\ \Pi(y : \delta \sharp x).\ \sigma$, where the argument y is the additional information necessary to build a value of the lifted type. The type of y is given by the *extension* δ of kind $\hat{\tau}_- \to \mathsf{Typ}$, which takes the skeleton and gives the type of the missing information. This dependence allows us to add different pierces of information for different shapes of the skeleton, *e.g.* in the case of natlist α, we need no additional information when the skeleton is $\hat{\mathsf{Z}}$, but a value of type α when the skeleton starts with $\hat{\mathsf{S}}$, as explained at the end of §3.1. The encoding works inductively, *i.e.* one layer at a time, thus all the functions take an argument of type $\hat{\tau}_-$.

The projection $\mathsf{proj}_{\chi\,(\omega_j)^j}$ from the lifted type to the skeleton is given by reading the clauses of the ornament definition from right to left:

$$\mathsf{proj}_{\chi\,(\omega_j)^j} \;:\; \sigma \to \hat{\tau}_- \;\triangleq\; \lambda^\sharp(x : \sigma_{\chi\,(\omega_j)^j}).\ \mathsf{match}\ x\ \mathsf{with}\ (Q_i \to \hat{P}_i)^i$$

The extension $\delta_{\chi\,(\omega_j)^j}$ is determined by computing, for each clause $P_i \to Q_i$, the type of the information missing to reconstitute a value. There are many possible representations of this information. The representation we use is given by the function $[\![Q_i]\!]$ mapping a pattern to a type and defined below[9]. There is no missing information in the case of variables, since they correspond to variables on the left-hand side. In the case of constructors, we need the missing information corresponding to each subpattern, given as a tuple. For wildcards, we need a value of the type matched by the wildcard. Finally, for an alternative pattern, we require to choose between the two

[8]The subscript – in $\hat{\tau}_-$ is to suggest that the non-recursive part has not yet been lifted, by contrast with τ_+.
[9]Formally, we translate pattern typing derivations instead of patterns

sides of the alternative and give the corresponding information, representing this as a sum type $\tau_1 + \tau_2$.

$$\begin{array}{rcl@{\qquad\qquad}rcl} [\![(\_ : \tau)]\!] &=& \tau & [\![P \mid Q]\!] &=& [\![P]\!] + [\![Q]\!] \\ [\![x]\!] &=& \mathsf{unit} & [\![d(P_1, .. P_n)]\!] &=& [\![P_1]\!] \times .. [\![P_n]\!] \end{array}$$

Then, the extension $\delta_{\chi\,(\omega_j)^j}$ matches on the $(P_i)^i$ to determine by which clause of the ornament definition the given skeleton is handled, and returns the corresponding extension type:

$$\delta_{\chi\,(\omega_j)^j} \;:\; \Pi(x:\sigma).\,\hat{\tau}_- \;\triangleq\; \lambda^\sharp(x:\hat{\tau}_-).\,\mathsf{match}\,x\,\mathsf{with}\,(\hat{P}_i \to [\![Q_i]\!])^i$$

The code reconstructing the ornamented value is given by the function $\mathsf{Lift}(Q_i, y)$ defined below, assuming that the variables of Q_i are bound and that y of type $[\![Q_i]\!]$ contains the missing information:

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{Lift}(\_, y) &=& y & \mathsf{Lift}(P \mid Q, y) &=& \mathsf{match}\ y\ \mathsf{with\ inl}\ y_1 \to \mathsf{Lift}(P, y_1) \mid \mathsf{inr}\ y_2 \to \mathsf{Lift}(Q, y_2) \\ \mathsf{Lift}(x, y) &=& x & \mathsf{Lift}(d(P_i)^i, y) &=& \mathsf{match}\ y\ \mathsf{with}\ (y_i)^i \to d(\mathsf{Lift}(P_i, y_i))^i \end{array}$$

The construction function $\mathsf{cstr}_{\chi\,(\omega_j)^j}$ then examines the skeleton to determine which clause of the ornament to apply, and calls the corresponding reconstruction code (writing just δ for $\delta_{\chi\,(\omega_j)^j}$:

$$\mathsf{cstr}_{\chi\,(\omega_j)^j} \;:\; \Pi(x:\hat{\tau}_-).\,\Pi(y:\delta\sharp x).\,\sigma \;\triangleq\; \lambda^\sharp(x:\hat{\tau}_-).\,\lambda^\sharp(y:\delta\sharp x).\,\mathsf{match}\ x\ \mathsf{with}\ (\hat{P}_i \to \mathsf{Lift}(Q_i, y))^i$$

In the case of natlist, we recover the definitions given in §3.3, with a slightly more complex (but isomorphic) encoding of the extra information:

$$\begin{array}{rcl} \sigma_{\mathsf{natlist}\,\tau} &=& \mathsf{list}\ \tau \\ \delta_{\mathsf{natlist}\,\tau} &=& \lambda^\sharp(x:\hat{\mathsf{nat}}(\mathsf{list}\ \tau)).\ \mathsf{match}\ x\ \mathsf{with}\ \hat{\mathsf{Z}} \to \mathsf{unit} \mid \hat{\mathsf{S}}\ x \to \tau \times \mathsf{unit} \\ \mathsf{proj}_{\mathsf{natlist}\,\tau} &=& \lambda^\sharp(x:\mathsf{list}\ \tau).\ \mathsf{match}\ x\ \mathsf{with\ Nil} \to \hat{\mathsf{Z}} \mid \mathsf{Cons}\ (y, \_) \to \hat{\mathsf{S}}\ y \\ \mathsf{cstr}_{\mathsf{natlist}\,\tau} &=& \lambda^\sharp(x:\hat{\mathsf{nat}}(\mathsf{list}\ \tau)).\ \lambda^\sharp(y:\delta_{\mathsf{natlist}\,\tau}\sharp x). \\ && \quad \mathsf{match}\ y\ \mathsf{with}\ \hat{\mathsf{Z}} \to (\mathsf{match}\ y\ \mathsf{with}\ () \to \mathsf{Nil}) \mid \hat{\mathsf{S}}\ x' \to (\mathsf{match}\ y\ \mathsf{with}\ (y', ()) \to \mathsf{Cons}\ (y', x')) \end{array}$$

The identity ornament corresponding to a datatype ζ defined as $(d_i : \forall(\alpha_j : \mathsf{Typ})^j\ (\tau_k)^k \to \zeta\ (\alpha_j)^j)^i$ is automatically generated and is described by the following code (since we do not add any information, the extension is isomorphic to unit):

$$\mathsf{type\ ornament}\ \zeta\ (\alpha_j)^j : \zeta\ (\alpha_j)^j \to \zeta\ (\alpha_j)^j\ \mathsf{with}\ (d_i(x_k)^k \to d_i(x_k)^k)^i$$

8.3 Correctness of the encoding

We must ensure that the terms defined in the previous section do correspond to the ornament as interpreted by the logical relation, as this is used to prove the correctness of the lifting. More precisely, we use the fact that the functions describing the ornamentation from the base type τ_- to the ornamented type $\sigma_{\chi\,(\omega_j)^j}$ are related to the functions defining the identity ornament of τ_-: the projection function maps related values to the same skeleton (with related recursive occurrences), and the construction function maps a skeleton with related recursive occurrences and *any* patch to related values.

Theorem 8.1 (Ornaments are related to the identity ornament). *Consider an ornament $\varphi = \chi\,(\omega_j)^j$ with base type $\tau_- = \zeta\ (\tau_k[\alpha_j \leftarrow \omega_j^-]^j)^k$ and ornamented type σ. Then:*

- *$\mathcal{E}[\chi\,(\omega_j)^j]_\emptyset$ is a relation, say R, between τ_- and σ;*
- *$(\mathsf{proj}_{\tau_-}, \mathsf{proj}_{\chi\,(\omega_j)^j}) \in \mathcal{V}[\Pi(x:\varphi).\ \hat{\zeta}\ (\tau_k[\alpha_j \leftarrow \omega_j]^j)^k\varphi]_{[\varphi \leftarrow R]}$;*
- *$(\mathsf{cstr}_{\tau_-}, \mathsf{cstr}_{\chi\,(\omega_j)^j}) \in \mathcal{V}[\Pi(x:\hat{\zeta}\ (\tau_k[\alpha_j \leftarrow \omega_j]^j)^k\varphi).\ \Pi(y:\delta\sharp x).\ \varphi]_\gamma$ where γ is $[\varphi \leftarrow R, \delta_\varphi \leftarrow (\lambda\_.\ \mathsf{Top})]$ and Top is the relation relating any two values (of the appropriate types).*

Proof. The relation is well-defined by induction on the index and the structure of the left-hand side term. For the second and third point, case-split on the structure of the arguments until the terms reduce to values, and compare them using the relation. □

Together, these properties allow us to take a term that uses the encoding of a yet-unspecified ornament φ and relate the terms obtained by instantiating with the identity and with another ornament, using the ornament's relation. We use this technique to prove the correctness of the elaboration.

We also prove that the logical interpretation of the identity ornament is the interpretation of the base type. Let us note (temporarily) id_ζ the identity ornament defined from the datatype ζ.

Lemma 8.2 (Identity ornament). *For all γ, $\mathcal{E}_k[\mathrm{id}_\zeta\ (\omega_i)^i]_\gamma = \mathcal{E}_k[\zeta\ (\omega_i)^i]_\gamma$.*

Proof. The definitions are the same. □

9 Elaboration

The goal of the elaboration is to transform an ML program into a generalized *m*ML meta-program such that ornamentation can take place simply by passing the appropriate meta-arguments to the *m*ML program. We need a way to describe how a generic program can be instantiated, what the result is, and how it is related to the original program.

Instead of abstracting over the meta-arguments, we will pass them through the environment. Thus, a generic program comes with a description of an ornamentation environment that will be later instanciated. For example, the function add defined in §3.3 has a generic lifting add_gen that takes the terms describing two ornaments φ_m and φ_n of nat, as well as a patch. We then get a lifting of ornament type $\varphi_m \to \varphi_n \to \varphi_n$.

We describe the lifting of an ML term a_- to an *m*ML term a_+ through an elaboration judgment $\Gamma \vdash a_- \rightsquigarrow a_+ : \omega$. The elaboration to a generic lifting depends only on the term, and not on the lifting specified by the user, which will only be used during the instantiation phase. Hence, the ornamentation environment Γ and the resulting ornamentation type ω may be read as an outputs of the judgment: Γ contains a list of ornaments and patches needed to type the generic lifting, and will have to be instantiated to produce a concrete lifting; ω gives the ornament type linking the base term to the lifted type given by the instantiation.

The ornamentation environment Γ is not simply an *m*ML environment: it describes both the role of each argument that must be given to the generic lifting and the link between the base term and its lifting. Namely, each an abstract ornament is represented as a tuple $\varphi \mapsto (\sigma, \delta, \mathrm{proj}, \mathrm{cstr}) \triangleleft \tau$, which binds an ornament variable φ that can be used as an ornament of the type τ, and a quadruple of variables $(\sigma, \delta, \mathrm{proj}, \mathrm{cstr})$ corresponding to the encoding of the ornament φ in *m*ML. These are free variables in the generic lifting that will later be instantiated with the components of a concrete ornament. The environment Γ also contains patch variables $x :^\sharp \sigma$ that are used to pass patches to the generic lifting. To cope with ML parametric polymorphism, the environment also contains type variables $(\alpha_-, \alpha, \alpha_+)$ where α_- and α_+ are the base type variable and lifted type variable associated to the ornament type variable α. The base type variable is bound in the base term, while the lifted type variable can be used in the lifted term.

During the ornamentation process, the ornamentation environment Γ also accumulates bindings $x : \omega$ and $A =_\sigma A$ corresponding to the variables and equalities in scope. The ornamentation environment can be projected to a base environment Γ^- and a lifted environment Γ^+, such that if $\Gamma \vdash a_- \rightsquigarrow a_+ : \omega$, we have both $\Gamma^- \vdash a_- : \omega^-$ and $\Gamma^+ \vdash a_+ : \omega^+$. The projections of an environment Γ^ϵ and of an ornament ω^ϵ_Γ are defined on Figure 24 (we write ϵ for either + or − to avoid duplication).

$$\Gamma \ ::= \ \emptyset \mid \Gamma, (\alpha, \alpha, \alpha) \mid \Gamma, x : \omega \mid \Gamma, A =_\sigma A \mid \Gamma, \varphi \mapsto (\sigma, \delta, \mathsf{cstr}, \mathsf{proj}) \lhd \tau \mid \Gamma, x :^\sharp \sigma$$

$$\begin{aligned}
(\Gamma, x : \omega)^\epsilon &= \Gamma^\epsilon, x : \omega^\epsilon_\Gamma \\
(\Gamma, (\alpha_-, \alpha, \alpha_+))^\epsilon &= \Gamma^\epsilon, \alpha_s : \mathsf{Typ} \\
(\Gamma, x :^\sharp \sigma)^- &= (\Gamma, A =_\sigma B)^- \ = \ (\Gamma, \varphi \mapsto (\ldots) \lhd \tau)^- \ = \ \Gamma^- \\
(\Gamma, x :^\sharp \sigma)^+ &= \Gamma^+, x : \sigma \\
(\Gamma, A =_\sigma B)^+ &= \Gamma^+, A =_\sigma B \\
(\Gamma, \varphi \mapsto (\sigma, \delta, \mathsf{proj}, \mathsf{cstr}) \lhd \tau)^+ &= \Gamma^+, \sigma : \mathsf{Typ}, \ \delta : \hat{\tau}\sigma \to \mathsf{Typ}, \ \mathsf{proj} : \Pi(x : \sigma).\ \hat{\tau}\sigma, \\
&\qquad \mathsf{cstr} : \Pi(x : \hat{\tau}\sigma).\ \Pi(y : \delta \sharp x).\ \sigma
\end{aligned}$$

$$\frac{(\alpha_-, \alpha, \alpha_+) \in \Gamma}{\alpha^\epsilon_\Gamma \ = \ \alpha_s} \qquad \frac{\varphi \mapsto (\sigma, \ldots) \lhd \tau \in \Gamma}{\varphi^-_\Gamma = \tau^-_\Gamma \qquad \varphi^+_\Gamma \ = \ \sigma} \qquad (\omega_1 \to \omega_2)^\epsilon_\Gamma \ = \ (\omega_1)^\epsilon_\Gamma \to (\omega_2)^\epsilon_\Gamma \qquad (\zeta\ (\omega_i)^i)^\epsilon_\Gamma \ = \ \zeta\ ((\omega_i)^\epsilon_\Gamma)^i$$

Fig. 24. Projection of ornament environments and types

As an example, the generic lifting of the code a_- of the function add of §3.3 is the code a_+ of the function add_gen[10] which verifies $\Gamma \vdash a_- \rightsquigarrow a_+ : \varphi_n \to \varphi_m \to \varphi_m$ where Γ is $\varphi_n \mapsto (\sigma_n, \delta_n, n_{\mathsf{proj}}, n_{\mathsf{cstr}}) \lhd \mathsf{nat}, \ \ \varphi_m \mapsto (\sigma_m, \delta_m, m_{\mathsf{proj}}, m_{\mathsf{cstr}}) \lhd \mathsf{nat}, \ \ p_1 :^\sharp \Pi(\rho).\ \delta_m \sharp \mathsf{S}\ (\mathsf{add}_+\ m'\ n)$ and ρ binds[11] $\mathsf{add}_+ : \sigma_m \to \sigma_n \to \sigma_n, \ m : \sigma_m, \ m' : \sigma_m, \ n : \sigma_n, \ \diamond : m_{\mathsf{proj}} \sharp m = \hat{\mathsf{S}}(m')$. The genetic lifting add_gen can then be instantiated by substituting the terms and types describing an ornament for the variables $\sigma_n, \delta_n, n_{\mathsf{proj}}, n_{\mathsf{cstr}}$ and similarly for m, and by providing an appropriate patch.

9.1 Elaboration rules

To simplify the meta-theory, we restrict lifting to ML terms that do not contain polymorphism. This does not restrict the class of ML programs that can be lifted, as we can obtained such a program by expanding all polymorphic bindings. We will discuss in §9.3 how to handle term variables defined in other phrases that have a polymorphic type.

For a simpler presentation, we assume that constructors and pattern matching are always applied to variables. This condition can be met by lifting terms appearing in constructors and as arguments of pattern matching into (monomorphic) let bindings.

The elaboration rules, given on Figure 25, closely follow the syntax of the original terms: variables, abstractions, applications, and let bindings are ornamented to themselves. In Rule E-Con, the constructor is elaborated to a call to an ornament construction function. The additional information is generated by a patch, which must come from the environment Γ, and which receives as argument all the variables available in the lifted projection of the context Γ^+. In Rule E-Match, a call to the projection function is inserted before the pattern matching and the branches are elaborated separately.

At each construction and pattern matching, we need to choose a datatype ornament. The datatype ornaments are described through an auxiliary judgment $\Gamma \vdash \omega \mapsto \sigma, \delta, \mathsf{proj}, \mathsf{cstr} \lhd \tau$ that returns the tuple containing values representing the ornament ω of the type τ. We only allow using the identity ornament (Rule Orn-Id) or an abstract ornament from the environment (Rule Orn-Var): the variables can later be instantiated with concrete ornaments, but keeping them available as variables for the moment is necessary to prove the correctness of the ornamentation using a parametricity argument.

[10]The function add_gen given in the overview abstracts over the components of the ornamentation environment, which this version of does not.
[11]The equality $\diamond : m_{\mathsf{proj}} \sharp m = \hat{\mathsf{S}}(m')$ was left implicit in the overview.

We have the typing property we announced previously:

Lemma 9.1 (Well-typed elaboration). *The generated mML terms are well-typed: if $\Gamma \vdash a_- \rightsquigarrow a_+ : \omega$, then $\Gamma^- \vdash a_- : \omega^-$ and $\Gamma^+ \vdash a_+ : \omega^+$.*

Proof. By induction on an elaboration, we can reconstruct a typing derivation for the two sides. □

Moreover, all well-typed terms have an elaboration:

Lemma 9.2 (An elaboration exists). *Let a be a well-typed ML term without polymorphism and where construction and pattern matching are only done on variables. Suppose $\emptyset \vdash a : \tau$. Then, τ is also an ornament type, and there exists a_+ such that $\Gamma \vdash a \rightsquigarrow a_+ : \tau$, with Γ an environment only containing patches.*

Proof. By induction on the typing derivation: we can imitate the typing derivation. We only choose identity ornaments (Orn-Id), and add the required patches to the environment. □

9.2 Instantiation and correction

The elaboration of a base term a_- into a generic lifting a_+ uses only the definition of A_-, thus a_+ has not yet been specialized to a specific lifting. This returns an environment Γ and an ornament ω such that $\Gamma \vdash a_- \rightsquigarrow a_+ : \omega$ holds. The environment Γ describes the list of datatype ornaments and patches that are free in a_+. To obtain a concrete lifting, we have to build an instantiation of Γ with specific ornaments and patches. We will discuss later how to choose this instantiation from the information given by the user. The instantiation is represented by an environment γ_+ of signature Γ^+ (the projection of Γ).

We give a more precise definition of the instantiation:

Definition 9.3 (Instantiation). An instantiation γ_+ of an ornamentation context Γ is given by:

- For an ornament binding $\varphi \mapsto (\sigma, \delta, \mathrm{proj}, \mathrm{cstr}) \lhd \tau$, choose a concrete datatype ornament $\varphi = \chi\,(\omega_i)^i$, and substitute $\sigma, \delta, \mathrm{proj}, \mathrm{cstr}$ by the corresponding definitions given in §8.
- For a parametric binding $(\alpha_-, \alpha, \alpha_+)$, set an ornament ω and $\gamma(\alpha_+) = \omega^+$.
- For a patch, a term of the correct type.

We have: $\Gamma^+ \vdash \gamma_+$.

Since we restricted the elaboration to only using the identity ornament and ornament variables, there always exists an *identity instantiation* γ_+^{id} that instantiates every ornament with the identity ornament. We can show that the term $\gamma_+^{\mathrm{id}}(a_+)$ is equal to a_- (for the *m*ML equality judgment).

Lemma 9.4 (Identity instantiation). *Suppose $\Gamma \vdash a_- \rightsquigarrow a_+ : \omega$. There exists an identity instantiation γ_+^{id} of Γ^+ such that $\Gamma^- \vdash a_- \simeq \gamma_+^{\mathrm{id}}(a_+)$.*

Proof. By induction on the derivation. The identity instantiation is constructed as follows:

- Use the identity ornament for the ornament variable.
- All patches will return in the extension of an identity ornament. The extensions of identity ornament are isomorphic to unit, thus we can construct a suitable (terminating) patch.

The equality is proved syntactically by induction, using C-Split in the case of pattern matching. □

Correction of the lifting: We can then build a relation environment γ_+^{rel} that contains the identity ornaments and patches on the left-hand side, and our instantiation on the right-hand side. We have to give an interpretation of the ornament variables that appear in the result type: they are instantiated with the relation corresponding to the datatype ornament. Then, γ_+^{rel} is in $\mathcal{G}_k[\Gamma^+]$. We conclude that $(\gamma_+^{\mathrm{id}}(a_+), \gamma_+(a_+))$ is in $\mathcal{E}_k[\omega]_{\gamma_+^{\mathrm{rel}}}$, *i.e.* $\mathcal{E}_k[\gamma_+^{\mathrm{rel}}(\omega)]_\emptyset$.

$$\textsc{E-Var}\ \frac{x : \omega \in \Gamma}{\Gamma \vdash x \leadsto x : \omega} \qquad \textsc{E-Let}\ \frac{\Gamma \vdash a \leadsto A : \omega_0 \quad \Gamma, x : \omega_0 \vdash b \leadsto B : \omega}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b \leadsto \mathsf{let}\ x = A\ \mathsf{in}\ B : \omega} \qquad \textsc{E-App}\ \frac{\Gamma \vdash a \leadsto A : \omega_1 \to \omega_2 \quad \Gamma \vdash b \leadsto B : \omega_1}{\Gamma \vdash a\ b \leadsto A\ B : \omega_2}$$

$$\textsc{E-Fix}\ \frac{\Gamma, x : \omega_1 \to \omega_2, y : \omega_1 \vdash a \leadsto A : \omega_2 \quad \tau_1 = (\omega_1)^-_\Gamma \quad \tau_2 = (\omega_2)^-_\Gamma \quad \sigma_1 = (\omega_1)^+_\Gamma \quad \sigma_2 = (\omega_2)^+_\Gamma}{\Gamma \vdash \mathsf{fix}\ (x : \tau_1 \to \tau_2)\ y.\ a \leadsto \mathsf{fix}\ (x : \sigma_1 \to \sigma_2)\ y.\ A : \omega_1 \to \omega_2}$$

$$\textsc{E-Con}\ \frac{\begin{array}{c}\Gamma \vdash \omega \mapsto \sigma, \delta, \mathsf{cstr}, \mathsf{proj} \lhd \zeta\ (\omega_i)^i \\ \vdash \hat{d} : \forall (\alpha_i)^i \hat{\alpha}\ (\tau_j)^j \to \zeta\ (\alpha_i)^i \quad (x_j : \tau_j[(\alpha_i \leftarrow \omega_i)^i, \hat{\alpha} \leftarrow \omega])^j \in \Gamma \quad (p :^\sharp \Gamma^+ \to \delta \sharp \hat{d}(x_j)^j) \in \Gamma\end{array}}{\Gamma \vdash d(x_j)^j \leadsto \mathsf{let}\ y = p \sharp \Gamma^+\ \mathsf{in}\ \mathsf{cstr} \sharp \hat{d}(x_j)^j \sharp y : \omega}$$

$$\textsc{E-Match}\ \frac{\begin{array}{c}x : \omega_0 \in \Gamma \quad \Gamma \vdash \omega_0 \mapsto \sigma, \delta, \mathsf{cstr}, \mathsf{proj} \lhd \zeta\ (\omega_i)^i \quad (\vdash \hat{d}_k : \forall (\alpha_i)^i \hat{\alpha}\ (\tau_{kj})^j \to \zeta\ (\alpha_i)^i)^k \\ (\Gamma, (y_{kj} : \tau_{kj}[(\alpha_i \leftarrow \omega_i)^i, \hat{\alpha} \leftarrow \omega_0])^j,\ \mathsf{proj} \sharp x =_{\hat{\zeta}\ ((\omega_i)^+_\Gamma)^i \sigma} d_k(y_{kj})^j \vdash a_k \leadsto A_k : \omega)^k\end{array}}{\Gamma \vdash \mathsf{match}\ x\ \mathsf{with}\ (d_k(y_{kj})^j \to a_k)^k \leadsto \mathsf{match}\ \mathsf{proj} \sharp x\ \mathsf{with}\ (\hat{d}_k(y_{kj})^j \to A_k)^k : \omega}$$

$$\textsc{Orn-Var}\ \frac{\varphi \mapsto (\sigma, \delta, \mathsf{cstr}, \mathsf{proj}) \lhd \tau \in \Gamma}{\Gamma \vdash \varphi \mapsto \sigma, \delta, \mathsf{cstr}, \mathsf{proj} \lhd \tau} \qquad \textsc{Orn-Id}\ \frac{\zeta : (\mathsf{Typ})^i \to \mathsf{Typ}}{\Gamma \vdash \zeta\ (\omega_i)^i \mapsto \zeta\ ((\omega_i)^+_\Gamma)^i, \delta_{\zeta\ (\omega_i)^i}, \mathsf{cstr}_{\zeta\ (\omega_i)^i}, \mathsf{proj}_{\zeta\ (\omega_i)^i} \lhd \zeta\ (\omega_i)^i}$$

Fig. 25. Elaboration to a generalized term

Finally, because equal terms are in the same relations, $(a_-, \gamma_+(a_+)) \in \mathcal{E}_k[\gamma_+^{\mathrm{rel}}(\omega)]_\emptyset$: the base term is related to the lifted term at the ornament type $\gamma_+^{\mathrm{rel}}(\omega)$.

Definition 9.5 (Relational environment). The relational environment γ_+^{rel} is defined as follows:

- If we used the ornament ω to instantiate φ, $\sigma \leftarrow \mathcal{E}_k[\omega]_\emptyset$, $\delta \leftarrow \lambda(x).\ \mathrm{Top}$, $\mathsf{cstr} \leftarrow (\mathsf{cstr}_{\mathrm{id}}, \mathsf{cstr}_\omega)$, $\mathsf{proj} \leftarrow (\mathsf{proj}_{\mathrm{id}}, \mathsf{proj}_\omega)$.
- If we used a patch p to instantiate x, $x \leftarrow (\gamma_+^{\mathrm{id}}(x), p)$.

Then, ${\gamma_+^{\mathrm{rel}}}_1 = \gamma_+^{\mathrm{id}}$ and ${\gamma_+^{\mathrm{rel}}}_2 = \gamma_+$. We will also, as a notational convenience, say that $({}_+\varphi) = \gamma_+^{\mathrm{rel}}(\varphi) = \omega$, *i.e.* that ornament variables are mapped to the corresponding ornament.

Lemma 9.6 (The relational environment interprets). *We have:* $\gamma_+^{\mathrm{rel}} \in \mathcal{G}_k[\Gamma^+]$.

Proof. For patches this is immediate (Top contains everything if the left-hand side terminates). For ornaments, use the correction theorem (Theorem 8.1). □

Theorem 9.7 (Correction of the lifting). *Suppose* $\Gamma \vdash a_- \leadsto a_+ : \omega$*, and let* γ_+ *be an instantiation of* Γ*. Then,* $(a_-, \gamma_+(a_+)) \in \mathcal{E}_k[\gamma_+(\omega)]$.

Proof. We have $\gamma_+^{\mathrm{rel}} \in \mathcal{G}_k[\Gamma^+]$. Thus, $(\gamma_+^{\mathrm{id}}(a_+), \gamma_+(a_+)) \in \mathcal{E}_k[\omega^+]_{\gamma_+^{\mathrm{rel}}}$. We can substitute the left-hand side by a_- by stability of the relation by equality. We have a substitution result: $\mathcal{E}_k[\omega^+]_{\gamma_+^{\mathrm{rel}}} = \mathcal{E}_k[\gamma_+^{\mathrm{rel}}(\omega^+)] = \mathcal{E}_k[\gamma_+(\omega)]$ (syntactically for the last part: the types are simply equal). □

For example, in the case of add and append, we obtain that (add, append) is in $\mathcal{E}_k[\mathsf{natlist}\ \tau \to \mathsf{natlist}\ \tau \to \mathsf{natlist}\ \tau]_\emptyset$ for any type τ. We can subsequently generalize on τ to obtain a polymorphic append function.

This result is also used in reverse by the instantiation process to partially determine γ_+: from the ornament signature given by the user, we can infer some of the ornaments that should be used for instantiation. For example, if we ask for a lifting of add at type natlist $\tau \rightarrow$ natlist $\tau \rightarrow$ natlist τ, we can deduce that φ_m and φ_n should be instantiated with natlist τ. The ornament variables that do not appear in the signature ω are determined using the strategies given by the user (*e.g.* always using a given ornament if possible), or have to be instantiated explicitly.

9.3 Open ornamentation and polymorphism

These results can be extended to the case of open ornamentation, where a number of definitions (are assumed to) have already been ornamented but their code is not available. Ornamentation is modular: it is possible to lift a function that uses another function using only the signature of a lifting of this function.

For example, when generalizing add, we store the type of the generalized lifting add_gen: it can have ornament type $\varphi_m \rightarrow \varphi_n \rightarrow \varphi_n$ for any two ornaments φ_m, φ_n of nat. Now consider the instantiation that gives append: for any type α, we instantiate φ_m and φ_n with natlist α. As usual in ML, we generalize append on α when adding it to the context, and we store the information that, for all α, append α is an ornament of add with $\varphi_m = \varphi_n =$ natlist α. Then, when encountering an use of add in a function, we can replace it with some instance of append in the instantiation environment (and with add again in the identity environment). The two environments are related since add and append are related, thus the ornamentation is correct.

9.4 Termination via the inverse relation

In order to prove that, when the patches terminate, the lifted term does not terminate less that the base term, we need to use the relation the other way, with the base term on the right and the lifted type on the left.

The relation is defined similarly. The only difference is that the Top relation, in the first case, relates any term on the base side (*i.e.* the left) to a non-terminating term on the lifted side (*i.e.* the right), while the reversed Top relates any *terminating* term on the lifted side (*i.e.* this time, the left) to any term on the base side (*i.e.* the right). Thus, the difference occurs at instantiation: we need to prove that the patches terminate to inject them in the relation.

The definition is simply reversed and the properties are similar:

10 Discussion

10.1 Implementation and design issues

Our prototype tool for refactoring ML programs using ornaments closely follows the structure outlined in this paper: programs are first elaborated into a generic term and then instantiated and reduced in a separate phase. From our experience, this approach is more modular and less error prone than an attempt to go directly from base terms to ornamented terms. At this stage, our prototype is just of proof of concept and more features of different nature and different level of difficulties guided by more experimentation will certainly be needed to obtain a mature tool.

As presented, ornamentation abstracts over all possible ornamentation points, which requires to specify many identity ornaments and write corresponding trivial patches, while many datatypes will probably never be ornamented. Instead of specifying these ornaments manually, using rules that may need to be repeated in many liftings, we could also allow toplevel rules. For example, refactoring a library may need a specific ornament for one type and the identity ornament for all others. For efficiency reasons, we could also avoid generating ornamentation points in the generic lifting when we know in advance that they will use the identity ornament.

The ornamentation strategies we provide seem to work well for small examples, but it remains to see if they also scale to larger examples with numerous patches.

More exploration is certainly needed on user interface issues. Ornaments can be used in different scenarios. For refactoring, ornaments are just used to modify the base code into the lifted code, after which the base code may be ignored—or just kept for archival purposes. In other scenarios, both the base code and the lifted code may coexist in the same program. Then, the base code may be updated and changes should ideally be propagated to the lifted code. For such cases, we may need other constructs for describing patches that will be more robust to base code changes. Currently, our only user interface is to describe lifting and patches in batch mode, but it is easy to imagine building some interactive tools on top of this interface, that will display the skeleton of incomplete liftings and point the user interactively to all occurrences that need some user input.

10.2 Deep pattern matching

Most ML programs use deep pattern matching, while the language formalized, as well as the core of our prototype, only treats shallow pattern matching. When compiling deep pattern matching to shallow pattern matching, we annotate the generated matches with tags that are maintained during the elaboration and we try to merge back pattern matchings with identical tags after elaboration, so as to preserve the structure of the input program whenever possible. This seems to work well, and a primitive treatment of deep pattern matching does not seem necessary and would be more involved, so we currently do not feel the need for such an extension.

Pattern matched clauses with wildcards may be also exploded in different clauses with different head constructors. For the moment we factor them back only in obvious cases, but we plan to also use tags to try to refactor clauses in the lifted code that originate from the same clause in the base code.

10.3 Let bindings

The ornamentation of let-bound functions can be specialized differently at each use point, which may duplicate code during meta-reduction which happens at *ornamentation time*.

Currently, the elaboration expands all local polymorphic let-bindings in the base code. The expressions resulting from the expansion could be tagged so as to share them back after ornamentation if their instantiations are identical. This approach also requires the user to instantiate what appears to be the same code at different program points. Another approach would be to allow the user to provide several ornamentations at the definition point, and then choose one ornamentation at each usage point. From a theoretical point of view, this is equivalent to ornamenting each usage point separately and then folding back the common instantiations at the original definition point.

Another solution could be to restrict ornamentation polymorphism, and therefore force some instantiations to be shared. We could even treat (at least by default) all local let-bindings monomorphically, *e.g.* as in Haskell [6].

We also generate extra monomorphic let-bindings to ensure that constructor applications and pattern matchings are only done on variable. Besides, generating these extra bindings allow us to preserve the evaluation order of the original program. While this does not influence the result in our presentation as our language is assume to pure, this is necessary for applying ornaments in actual ML programs. These let-bindings are then expanded in the lifted program when they bind values or are linear and their expansion preserves the evaluation order. In our experience, these transformations produce easily readable output programs.

10.4 Naming conventions

Several phase introduce auxiliary variables: the lifting itself that may expend wildcards, introduce shallow pattern matching or auxiliary let-bindings. Some of these bindings will eventually be expanded, but some will remain in the lifted program. Since we wish the lifted code to be readable, it is important in practice to select meaningful names. While this is an orthogonal issue and we have not done anything yet, simple strategies such as choosing prefixes based on user-provided names appearing in similar patterns could be easy to implement and increase readability significantly.

10.5 Lifting

When the lifting process is partial, it returns code with holes that have to be filled manually. Our view is that filling the holes is an orthogonal issue that can be left as a post-processing pass, with several options that can be studied independently but also combined. One possibility is to use code inference techniques such as implicit parameters [2, 13, 15], which could return three kinds of answers: a unique solution, a default solution, *i.e.* letting the user know that the solution is perhaps not unique, or failure. In fact, it seems that a very simple form of code inference might be pertinent in many cases, similar to Agda's instance arguments [5], which only inserts variable present in the context. An alternative solution to code inference is an interactive tool to help the user build patches.

In our presented, we implicitly assume that all the code is available to the ornamentation tool. Ornamentation schemes can be derived for the whole program, ornamented at once. In many realistic scenarios, programs are written in a modular way. A module ornamentation could describe the relation between a base module and an ornamented module. As in the case of open ornamentation, generalization would have to consider functions from other modules as abstract, and such functions would have to be instantiated by an equivalent function. Modular ornamentation could be applied to libraries: when a new interface-incompatible version of a library appears, a maintainer could distribute an ornamentation specification allowing clients of the library to automatically migrate their code, leaving holes only at the crucial points requiring user input.

10.6 Semantic issues

Our approach to ornamentation is not *semantically* complete: we are only able to generate liftings that follow the syntactic structure of the original program, instead of merely following its observable behavior. Most reasonable ornamentations seem to follow this pattern. Syntactic lifting seems to be rather predictable and intuitive and lead to quite natural results. Syntactic lifting also helps with automation by reducing the search space. Still, it would be interesting to find a less syntactic description of which functions can be reached by this method.

Our presentation of ornamentation ignores effects, as well as the runtime complexity of the resulting program. A desirable result would be that an ornamented program produces the same effects as the original program, save from the effects done in patches. A problem is that effects done in patches could influence the automatically generated code (for example, modification of a reference). Similarly, the complexity of the ornamented program should be proportional to the complexity of the original one, as long as the patches run in constant time (or excluding the computation done in those patches).

We have described ornaments as an extension of ML, equipped a call-by-value semantics, but only to have a fixed setting: our proposal should apply seamlessly to *core* Haskell.

10.7 Beyond ML

Extending our results to the case of generalized abstract datatypes (GADT) is certainly useful but still challenging future work. Programming with GADTs requires writing multiple definitions of the same type, but holding different invariants. GADT definitions that only add *constraints* could be considered ornaments of regulars types, which was actually one of the main motivations for introducing ornaments in the first place [4]. It would then be useful to automatically derive, whenever possible, copies of the functions on the original type that preserve the GADT's invariants. A possible approach with our current implementation is to generate the function, ignoring the constraints, and hoping it typechecks, but a more effective strategy will probably be necessary. Besides the problem of type inference, GADTs will also make this analysis of dead branches more difficult. GADTs also raise the question of non-regular data-structures, which we haven't explored yet.

Although dependently typed, the meta-language *m*ML is in fact quite restrictive. It must fulfill two conflicting goals: be sufficiently expressive to make the generic lifting well-typed, but also restrictive enough so that elaborated programs can be reduced and simplified back to ML. Hence, many extensions of ML will require changing the language *e*ML as well, and it is not certain that the balance will be preserved, *i.e.* that lifted program will remain typable in the source language.

The languages *e*ML and *m*ML have only been used as intermediate languages and are not exposed to the programmer. We wonder whether they would have other useful applications either for other program transformations or providing the user with some meta-programming capabilities. For example, *e*ML is equipped to keep track of term equalities during pattern matching and could perhaps have applications in other settings. Similarly, *m*ML provides a form of meta-programming with good meta-theoretical properties and one might consider exposing it to the user, for example to let her write generic patches that could be instantiated as needed at many program points.

The design of *e*ML balances two contrasting goals: we need a language powerful enough to be able to type the ornament encoding, but we also want to be able to eliminate the non-ML features from a term. We could encode specific ornaments by reflecting the structure of the constructors into GADTs, but this would only work for one given structure. The extension function δ allows us to look arbitrarily deep into the terms, depending on what ornament we want to construct.

11 Related works

Ornamentation is a concept recently introduced by [3, 4] in the context of dependently typed languages, where it is not a primitive concept and can be encoded. The only other work to consider applying ornaments to an ML-like language we are aware of is [16].

Type-Theory in Color [1] is another way to understand the link between a base type and a richer one. Some parts of a datatype can be tainted with a color modality: this allows tracing which parts of the result depend on the tainted values and which are independent. Terms operating on a colored type can then be erased to terms operating on the uncolored version. This is internalized in the type theory: in particular, equalities involving the erasure hold automatically. This is the inverse direction from ornaments: once the operations on the ornamented datatype are defined, the base functions are automatically derived, as well as a coherence property between the two implementations. Moreover, the range of transformations supported by type theory in color is more limited: it only allows field erasure, but not, for example, to rearrange a products of sums as a sum of products; conversely, it also allows erasing function arguments, which we do not support yet—but could easily add.

Programming with GADTs may require defining one base structure and several structures with some additional invariants, along with new functions for each invariant. Ghostbuster [9] proposes a *gradual* approach, by allowing as a temporary measure to write a function against the base structure and dynamically check that it respects the invariant of the richer structure, until the appropriate function is written. While the theory of ornaments supports adding invariants, we do not yet support GADTs. Moreover, we propose ornaments as a way to generate new code to be integrated into the program, rather than to quickly prototype on a new datatype.

Ornaments are building on datatype definitions, which are a central feature of ML. Polytypic programming is a successful concept also centered on the idea of datatypes, but orthogonal to ornaments. Instead of lifting operations from one datatype to another with a similar structure, it tries to have a universal definition for an operation that applies to all datatypes at once, the behavior being solely determined by logical (sum or product) structure of the datatype.

In [10], the authors also observe that one often need many variants of a given data structures (typically an abstract syntax tree), and corresponding functions for each variant. They propose a programming idiom to solve this problem: they create an extensible version of the type, and use type families to determine from an extension name what information must be added to each constructor. In this approach, the type of the additional information only depends on the constructor, while our type-level pattern matching allows depending on the information stored in the already-present fields. This approach uses only existing features of GHC, avoiding a separate pre-processing step and allowing one to write generic functions that operate on all decorations of a tree. On the other hand, the programmer must pay the runtime cost of the encoding even when using only the undecorated tree. The encoding of extensible trees scales naturally to GADTs. Interestingly, this idiom and ornaments are largely orthogonal features with some common use case (factoring operations working on several variants of the same datatype) and might hopefully benefit from one another.

Ornamentation is a form of code refactoring on which there is a lot of literature, but based on quite different techniques and rarely supported by a formal treatment. It has however not been much explored in the context of ML-like languages.

Views, first proposed by Wadler [14] and later reformulated by Okasaki [11] have some resemblance with isomorphic ornaments. They allow several interchangeable representations for the same data, using isomorphism to switch between views *at runtime* whenever convenient. The example of location ornaments, which allows to program on the bare view while the data leaves in the ornamented view, may seem related to views, but this is a misleading intuition. In our case, the switch between views is at *editing time* and nothing happens at runtime where only the ornamented core with location is executed. In fact, this runtime change has a runtime cost, which is probably one of the reasons why the appealing concept of views never really took off. Lenses [7] also focus on switching representations at runtime.

The ability to switch between views may also be thought of as the existence of inverse coercions between views. Coercions may be considered as the degenerate of views in the non-isomorphic case. But coercions are not more related to ornaments than views—for similar reasons.

Conclusion

We have designed and formalized an extension of ML with ornaments. We have used logical relations as a central tool to give a meaning to ornaments, to closely relate the ornamented and original programs, and to guide the lifting process. We believe that this constitutes a solid, but necessary basis for using ornaments in programming. This is also a new use of logical relations applied to type-based program refactoring.

Ornaments seems to have several interesting applications in an ML setting. Still, we have so far only explored them on small examples and more experiment is needed to understand how they behave on large scale programs. We hope that our proof-of-concept prototype could be turned into a useful, robust tool for refactoring ML programs. Many design issues are still open to move from a core language to a full-fledged programming language. More investigation is also needed to extend our approach to work with GADTs.

A question that remains unclear is what should be the status of ornaments: should they become a first-class construct of programming languages, remain a meta-language feature used to preprocess programs into the core language, or a mere part of an integrated development environment?

References

[1] J.-P. Bernardy and M. Guilhem. Type-theory in color. In *International Conference on Functional Programming*, pages 61–72, 2013. doi: 10.1145/2500365.2500577.

[2] P. Chambard and G. Henry. Experiments in generic programming: runtime type representation and implicit values. Presentation at the OCaml Users and Developers meeting, Copenhagen, Denmark, sep 2012. URL http://oud.ocaml.org/2012/slides/oud2012-paper4-slides.pdf.

[3] P. Dagand and C. McBride. A categorical treatment of ornaments. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 530–539. IEEE Computer Society, 2013. ISBN 978-1-4799-0413-6. doi: 10.1109/LICS.2013.60. URL http://dx.doi.org/10.1109/LICS.2013.60.

[4] P. Dagand and C. McBride. Transporting functions across ornaments. *J. Funct. Program.*, 24(2-3):316–383, 2014. doi: 10.1017/S0956796814000069. URL http://dx.doi.org/10.1017/S0956796814000069.

[5] D. Devriese and F. Piessens. On the bright side of type classes: Instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 143–155, 2011. doi: 10.1145/2034773.2034796.

[6] T. S. Dimitrios Vytiniotis, Simon Peyton Jones. Let should not be generalised. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. Association for Computing Machinery, Inc., January 2010. URL https://www.microsoft.com/en-us/research/publication/let-should-not-be-generalised/.

[7] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007. doi: http://portal.acm.org/citation.cfm?doid=1232420.1232424.

[8] R. Hinze. Numerical representations as Higher-Order nested datatypes. Technical report, 1998.

[9] T. L. McDonell, T. A. K. Zakian, M. Cimini, and R. R. Newton. Ghostbuster: A tool for simplifying and converting gadts. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 338–350, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951914. URL http://doi.acm.org/10.1145/2951913.2951914.

[10] S. Najd and S. Peyton-Jones. Trees that grow. *JUCS*, 2016. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/trees-that-grow-2.pdf.

[11] C. Okasaki. Views for standard ml. In *In SIGPLAN Workshop on ML*, pages 14–23, 1998.

[12] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1998. ISBN 978-0521663502.

[13] Scala. Implicit parameters. Scala documentation. URL http://docs.scala-lang.org/tutorials/tour/implicit-parameters.

[14] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction, 1986.

[15] L. White, F. Bour, and J. Yallop. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2. URL http://dx.doi.org/10.4204/EPTCS.198.2.

[16] T. Williams, P. Dagand, and D. Rémy. Ornaments in practice. In J. P. Magalhães and T. Rompf, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 15–24. ACM, 2014. ISBN 978-1-4503-3042-8. doi: 10.1145/2633628.2633631. URL http://doi.acm.org/10.1145/2633628.2633631.