# A Principled Approach to Ornamentation in ML

Thomas Williams, Didier Rémy

# A Principled approach to Ornamentation in ML

THOMAS WILLIAMS, Inria, France
DIDIER RÉMY, Inria, France

Ornaments are a way to describe changes in datatype definitions 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, including code refactoring, study the metatheoretical properties of ornaments, and describe their elaboration process. We formalize ornamentation via an a posteriori abstraction of the bare code, returning a generic term, which lives in a meta-language above ML. The lifted code is obtained by application of the generic term to well-chosen arguments, followed by staged reduction, and some remaining simplifications. We use logical relations to closely relate the lifted code to the bare code.

CCS Concepts: • **Software and its engineering → Functional languages**; **Polymorphism**; **Data types and structures**; **Semantics**; **Software maintenance tools**;

Additional Key Words and Phrases: Ornaments, ML, Refactoring, Depedent Types, Logical Relations.

## 1 INTRODUCTION

Inductive datatypes and parametric polymorphism are 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 and is a key to modular programming and reusability.

Datatype definitions are inductively defined as labeled sums and products over primitive types. However, the same data can often be represented with several isomorphic data-structures, using a different arrangement of sums and products. Two data-structures may also differ in minor ways, for instance sharing the same recursive structure, but one carrying an extra information at some specific nodes. Having established the structural ties between two datatypes, one soon realizes that both admit strikingly similar functions, operating similarly over their common 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, the type checker 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, which is boring, time consuming, and error prone. Worse, 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.

The recent theory of ornaments [Dagand and McBride 2013, 2014; McBride 2011] seems 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, an ornament is a relation between two datatypes, reorganizing, specializing, and adding data to a bare type to obtain an ornamented type. In previous work [Williams et al. 2014], we have already explored the interest of ornamentation in the context of ML where ornaments are added as a primitive notion rather than encoded, and sketched how functions operating on some datatype could be lifted to work on its ornamented version instead. We both generalize and formalize the previous approach, and propose new typical uses of ornaments.

Our contributions are the following: we extend the definition of ornaments to the higher-order setting; we give ornaments a semantics using logical relations and establish a close correspondence between the bare code and the lifted code (Theorem 9.9); we propose a new principled approach to the lifting process, through *a posteriori* abstraction of the bare code to a most general syntactic elaborated form, which is then instantiated into a concrete lifting, meta-reduced, and simplified back to ML code; this appears to be a general schema for refactoring tools that should also be useful for other transformations than ornamentation; we introduce an intermediate meta-language above ML with a restricted form of dependent types, which are used to keep track of selected branches during pattern matching, and could perhaps also be helpful 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 presented 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 §5.2 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 formally describe the lifting process that transforms a lifting declaration into actual ML code, and we justify its correctness. We discuss our implementation and possible extensions in §10 and related work in §11.

## 2 EXAMPLES OF ORNAMENTS

Let us discover ornaments by means of examples. All examples preceded by a blue vertical bar have been processed by a prototype implementation[1], which follows an OCaml-like[2] syntax. Output of the prototype appears with a wider green vertical bar. 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 of ornaments is the special case of code refactoring, which is an often annoying but necessary task when programming. We start with an example reorganizing a sum data structure into a sum of sums. Consider 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 the binary operators Add and Mul can be factorized, and thus prefer the following version expr' using an auxiliary type of binary operators (given below on the left-hand side). There is a relation between these two types, which we may describe as an *ornament* oexpr from the base type expr to the ornamented type expr' (right-hand side).

[1] The prototype, available at url http://gallium.inria.fr/~remy/ornaments/, contains a library of detailed examples (including those presented here). More examples can also be found in the extended version of this article [Williams and Rémy 2017].
[2] http://caml.inria.fr/

```
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)   when u v : oexpr
  | Mul (u, v)  ⇒ Binop' (Mul', u, v)   when u v : oexpr
```

This definition is to be understood as

```
type ornament oexpr : expr ⇒ expr' with
  | Const i_-          ⇒ Const' i_+   when i_- ⇒ i_+ in int
  | Add (u_-, v_-) ⇒ Binop' (Add', u_+, v_+)  when u_- ⇒ u_+ and v_- ⇒ v_+ in oexpr
  | Mul (u_-, v_-) ⇒ Binop' (Mul', u_+, v_+)  when u_- ⇒ u_+ and v_- ⇒ v_+ in oexpr
```

This recursively defines the oexpr relation. A clause "$x_- \Rightarrow x_+$ **in** orn" means that $x_-$ and $x_+$ should be related by the ornament relation orn. The first clause is the base case. By default, the absence of ornament specification for variable i (of type int) has been expanded to "**when** $i_- \Rightarrow i_+$ **in** int" and means that $i_-$ and $i_+$ should be related by the identity ornament at type int, which is also named int for convenience. The next clause is an inductive case: it means that Add($u_-$, $v_-$) and Binop' (Add'($u_+$, $v_+$)) are in the oexpr relation whenever $u_-$ and $u_+$ on the one hand and $v_-$ and $v_+$ on the other hand are already in the oexpr relation.

In this example, the relation happens to be an isomorphism and we say that the ornament is a *pure* refactoring. Hence, the compiler has enough information to automatically lift the old version of the code to the new version. We just request this lifting as follows:

```
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.) Here, the compiler will automatically elaborate 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 is this well-typed, but the semantics is also preserved—by construction. Notice that a pure 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'.

Pure refactorings such as oexpr are a particular, but quite interesting subcase of ornaments because the lifting process is fully automated. As a tool built upon ornamentation, we provide a shortcut for refactoring: one only has to write the definitions of expr' and oexpr, and lifting declarations are generated to transform a whole source file. Thus, pure refactoring is already a very useful applications of ornaments: these transformations become almost free, even on a large code base.

Besides, proper ornaments as described next that decorate an existing node with new pieces of information can often be decomposed into a possibly complex but pure refactoring and another proper, but hopefully simpler ornament. Notice that pure code refactoring need not even define a new type. One such example is to invert values of a boolean type:

```
type bool = True | False
type ornament not : bool ⇒ bool with True ⇒ False | False ⇒ True
```

Then, we may define bor as a lifting of band, and the compiler inverts the constructors:

```
let band u v = match u with True → v | False → False
let bor = lifting band : not → not → not
let bor u v = match u with
    | True → True
    | False → v
```

It may also do this selectively, only at some given occurrences of the bool type. For example, we may only invert the first argument:

```
let bnotand = lifting band : not → bool → bool
let bnotand u v = match u with
    | True → False
    | False → v
```

Still, the compiler will carefully reject inconsistencies, such as:

```
let bandnot = lifting band : bool → not → bool
```

Indeed, the result should be an ornament of bool.

## 2.2 Code Refinement

Code refinement is an example of a proper ornament where the intention is to *derive* new code from existing code, rather than *modify* existing code and forget the original version afterwards. To illustrate code refinement, observe that lists can be considered as an ornament of Peano numbers:

```
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)  when m : 'a natlist
```

The parametrized ornamentation relation 'a natlist is not an isomorphism: a natural number S $m_-$ will be in relation with all values of the form Cons (x, $m_+$) as long as $m_-$ is in relation with $m_+$, for any x. We use an underscore "_" instead of x on Cons (_, m) to emphasize that it does not appear on the left-hand side and thus freely ranges over values of its type. Hence, the mapping from nat to 'a list is incompletely determined: we need additional information to translate a successor node.(Here, 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 (below, on the left). However, this returns an incomplete lifting (on the right):

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

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

Indeed, this requires building a cons node from a successor node, which is underdetermined. This is reported to the user by leaving a labeled hole #2 in the generated code. The programmer may use this label to provide a *patch* that will fill this hole. The patch may use all bindings that were 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 matching on *m* again:

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

```
  with #2 ← match m with Cons(x, _) → x
```

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 also removes an incomplete pattern matching. 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 explicitly determined and we always leave holes in the skeleton when patches are missing.

This example is chosen here for pedagogical purposes, as it illustrates the key ideas of ornamentation. While it may seem anecdotal, there is a strong relation between recursive data structures and numerical representations, whose relation to ornamentation has been considered by Ko [2014].

### 2.3 Composing Transformations—a Practical Use Case

Ornamentation could be used in different scenarios: the intent of *refactoring* is to replace the base code with the generated code, even though the base code could also be kept for archival purposes; when *enriching* a data structure, both codes may coexist in the same program. To support both of these usages, we try to generate code that is close to manually written code. For other uses, the base code and the lifting instructions may be kept to regenerate the lifted code when the base code changes. This already works well in the absence of patches; otherwise, we would need a patch description language that is more robust to changes in the base code. We could also postprocess ornamentation with some simple form of code inference that would automatically try to fill the holes with "obvious" patches, as illustrated below. Our tool currently works in batch mode and is just providing the building blocks for ornamentation. The ability to output the result of a partially specified lifting makes it possible to build an interactive tool on top of our interface.

The following example shows how different use-cases of ornaments can be composed to reorganize, enrich, and cleanup an incorrect program, so that the final bug fix can be reduced to a manual but simple step. The underlying idea is to reduce manual transformations by using automatic program transformations whenever possible. Notice that since lifting preserves the behavior of the original program, fixing a bug cannot just be done by ornamentation.

Let us consider a small calculus with abstractions and applications and tuples (which we will take unary for conciseness) and projections. We assume given a type id representing variables.

```
type expr = Abs of id * expr | App of expr * expr | Var of id | Tup of expr | Proj of expr
```

We write an expression evaluator using environments. We assume given an assoc function of type 'a → ('a * 'b) list → 'b option that searches a binding in the environment:

```
let rec eval env e =
  match e with
  | Var x → assoc x env
  | Abs(x, f)  → Some (Abs(x, f))
  | App(e1,e2) →
    begin match eval env e1 with
     | Some (Abs(x,f)) →
        begin match eval env e2 with
        | Some v → eval (Cons((x,v), env)) f
        | None → None
        end
     | Some (Tup _) → None    (* Type error          *)
     | None → None            (*  error  propagation *)
     | Some _ → fail ()       (*  Not a value  ?!    *)
    end
  | Tup(e) →
     begin match eval env e with
     | Some v → Some (Tup v)
     | None → None
     end
  | Proj(e) →
     begin match eval env e with
     | Some (Tup v) → Some v
     | Some (Abs _) → None    (* Type error          *)
     | None → None            (*  error  propagation *)
     | Some _ → fail ()       (*  Not a value  ?!    *)
     end
(*  eval  :  (id  *  expr)  list  −> expr −> expr option *)
```

The evaluator distinguishes type (or scope) errors in the program, where it returns None, and internal errors when the expression returned by the evaluator is not a value. In this case, the evaluator raises an exception by calling fail ().

We soon realize that we mistakenly implemented dynamic scoping: the result of evaluating an abstraction should not be an abstraction but a closure that holds the lexical environment of the abstraction. One path to fixing this evaluator is to start by separating the subset of *values* returned by the evaluator from general expressions. We define a type of values as an ornament of expressions.

```
type value =
  | VAbs of id * expr
  | VTup of value
```

```
type ornament expr_value : expr ⇒ value with
  | Abs(x, e)  ⇒ VAbs(x, e) when e : expr
  | Tup(e)     ⇒ VTup(e) when e : expr_value
  | _ → ~
```

This ornament is intendedly *partial*: some cases are not lifted. Indeed, ornaments define a relation between the bare type and the ornamented type. They are defined syntactically, both sides being linear pattern expressions. Moreover, the pattern for the ornamented type should be total, i.e. match all expressions of the ornamented type. Conversely, the pattern of the bare type need not be total. When lifting a pattern matching with a partial ornament, the inaccessible cases will be dropped. On the other hand, when constructing a value that is impossible in the lifted type, the user will be asked to construct a patch of the empty type, which could be filled for example by an assertion failure. In some cases, the simplification will notice that all constructions are possible. The notation ~ corresponds to the empty pattern.

This ornament does not preserve the recursive structure of the original datatype: the recursive occurrences are transformed into values or expressions depending on their position. By contrast with prior works [Dagand and McBride 2013, 2014; Williams et al. 2014], we do not treat recursion specifically. Hence, mutual recursion is not a problem; for instance, we can ornament a mutually recursive definition of trees and forests or modify the recursive structure during ornamentation. There are still some limitations during lifting: since we preserve the structure of the code, we are not able to transform a single recursive function into a mutually recursive function. This limits possible lifting to those that do not require this unfolding, although unfolding could be done in a preprocessing pass if needed.

Using the ornament expr_value we transform the evaluator by making explicit the fact that it only returns values and that the environment only contains values (as long as this is initially true):

```
let eval' = lifting eval : (id * expr_value) list → expr → expr_value option
(* val eval' : (id * value) list -> expr -> value option *)
```

The lifting succeeds—and eliminates all occurrences of fail () in eval'.

```
let rec eval' env e = match e with
  | Abs(x, e) → Some (VAbs(x, e))
  | App(e1, e2) → bind' (eval' env e1) (function
     | VAbs(x, e) → bind' (eval' env e2) (fun v → eval' (Cons((x, v), env)) e)
     | VTup _ → None)
  | Var x → assoc x env | ...
```

We may now refine the code to add a field for storing the environment in closures:

```
type value' =
  | VClos' of id * (id * value') list * expr
  | VTup' of value'
```

```
type ornament value_value' : value ⇒ value' with
  | VAbs(x, e) ⇒ VClos'(x, _, e)
  | VTup(v)    ⇒ VTup'(v) when v : value_value'
```

Since this ornament is not one-to-one, the lifting of eval' is partial. The advanced user may realize that there should be a single hole in the lifted code that should be filled with the current environment env, and may directly write the clause "| * ← env":

```
let eval'' = lifting eval' with ornament * ← value_value', @id | * ← env
```

The annotation **ornament** * ← value_value', @id is another way to indicate which ornaments to use that is sometimes more convenient than giving a signature: for each type that needs to be ornamented, we first try value_value', and use the identity ornament if this fails (*e.g.* on types other than value). A more pedestrian path to writing the patch is to first look the output of the partial lifting:

```
let eval'' = lifting eval' with ornament * ← value_value', @id
```

```
let rec eval'' env e = match e with
  | Abs(x, e) → Some (VClos'(x, #32, e))
  | App (e1, e2) → ... | ...
```

The hole has been labeled #32 which can then be used to refer to this specific program point:

```
let eval'' = lifting eval' with ornament * ← value_value', @id | #32 ← env
```

An interactive tool could point the user to this hole in the partially lifted code shown above, so that she directly enters the code env, and the tool would automatically generate the lifting command just above. Notice that env is the most obvious way to fill the hole here, because it is the only variable of the expected type available in context. Hence, a very simple form of type-based code inference could pre-fill the hole with env and just ask the user to confirm.

When the programmer is quite confident, she could even ask for this to be done in batch mode:

```
let eval'' = lifting eval' with ornament * ← value_value', @id | * ← try by type
```

Example-based code inference would be another interesting extension of our prototye, which would increase the robustness of patches to program changes. Here, the user could instead write:

```
let eval'' = lifting eval' with ornament * ← value_value', @id
                                        | * ← try eval env (VAbs (_, _)) = Some (Closure (_, env, _))
```

providing a partial definition of eval that is sufficient to completely determine the patch.

For each of these possible specifications, the system will return the same answer:

```
let rec eval'' env e = match e with
  | Abs(x, e) → Some (VClos'(x, env, e))
  | App(e1, e2) → bind' ( eval'' env e1) (function
      | VClos'(x, _, e) → bind' ( eval'' env e2) (fun v → eval'' (Cons((x, v), env)) e)
      | VTup' _ → None)
  | Var x → assoc x env | ...
```

So far, we have not changed the behavior of the evaluator: the ornaments guarantee that the result of eval'' on some expression is essentially the same as the result of eval—up to the addition of an environment in closures. The final modification must be performed manually: when applying functions, we need to use the environment of the closure instead of the current environment.

```
let rec eval'' env e = match e with
  | Var x → assoc x env
  | Abs(x, e) → Some (VClos'(x, env, e))
  | App(e1, e2) →
    begin match eval'' env e1 with
      | (None | Some (VTup' _)) → None
      | Some (VClos'(x, closure_env, e)) →
        begin match eval'' env e2 with
          | None → None
          | Some v → eval'' (Cons((x, v), closure_env (* was env *) )) e
        end
    end
  | Tup e →
    begin match eval'' env e with
      | None → None
      | Some v → Some (VTup' v)
    end
  | Proj e →
    begin match eval'' env e with
      | (None | Some (VClos'(_, _, _))) → None
      | Some (VTup' v) → Some v
    end
```

## 2.4 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 $\tau_J$ is the sum $\Sigma^{j \in J} A_j$, say $\tau_J$ containing a few constant constructors and $\tau_K$ are the remaining cases. One may then chose to split cases into two sum types $\tau_J$ and $\tau_K$ and use the isomorphism $\tau_I$ set $\approx \tau_J$ set $\times\, \tau_K$ set to enable the optimization of $\tau_J$ set, for example by representing all cases as an integer—when $|J|$ is not too large.

### 2.5 Hiding Administrative Data

Sometimes data structures need to carry annotations, which are useful information for certain purposes but not at the core of the algorithms. A typical example is location information attached to abstract syntax trees for error 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 $\lambda$-expressions and its call-by-name evaluator:

```
type 'a option =
  | None
  | Some of 'a
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)
```

The datatype expr' that holds location information can be presented 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, _) when f : add_loc → add_loc
  | App (u, v) ⇒ (App' (u, v), _) when u v : add_loc
  | Const i ⇒ (Const' i, _)
```

The datatype for returning results 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 without further information,

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

the system will only be able to do a partial lifting, unsurprisingly:

```
let rec eval_incomplete e = match e with
    | (App'(u, v), x) →
      begin match (* _2 *) eval_incomplete u with
        | Ok (App'(u, v), x) → Error #4
        | Ok (Abs' f, x) → Ok (f v)
        | Ok (Const' i, x) → Error #4
        | Error x → Error #4
      end
    | (Abs' f, x) → Ok e
    | (Const' i, 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 #4. 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 places, but they all share the same label as they have the same origin. 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, using the auxiliary identifier _2 to refer to the inner match expression, as indicated in the inferred code.

```
let eval_loc = lifting eval : add_loc → (add_loc, loc) optres with
  | #4 ← begin match _2 with Error err → err
              | Ok _ → (match e with (_, loc) → loc) end
```

We then obtain the expected complete code:

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

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

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 still needed to see how the ornament approach scales up here to more complex transformations. This might be a case where appropriate tactics for filling the holes could be helpful.

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

## 2.6 Higher-order and Recursive Types

Lifting also works with higher-order types and recursive datatype definitions with negative occurrences. For example, we could extend arithmetic expressions with nodes for abstraction and application, with functions represented by functions of the host language:

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

Then, the evaluation function is partial:

```
let rec eval e = match e with
| Const i → Some(Const i)
| Add ( u , v ) →
   begin match (eval u, eval v) with
   | (Some (Const i1), Some (Const i2)) → Some(Const (add i1 i2))
   | _ → None
   end
| Mul ( u , v ) →
   begin match (eval u, eval v) with
   | (Some (Const i1), Some (Const i2)) → Some(Const (mul i1 i2))
   | _ → None
   end
| Abs f → Some(Abs f)
| App(u, v) →
   begin match eval u with
   | Some(Abs f) → begin match eval v with None → None | Some x → f x end
   | _ → None
   end
 val eval : expr → expr option
```

We could still prefer the following representation factoring the arithmetic operations:

```
type binop' =
  | Add'
  | Mul'

type expr' =
  | Const' of int
  | Binop' of binop' * expr' * expr'
  | Abs' of (expr' → expr' option)
  | App' of expr' * expr'
```

Then, we can define an ornament between these types, despite the presence of functions and negative occurrences in the type:

```
type ornament oexpr : expr ⇒ expr' with
  | Const ( i ) ⇒ Const' ( i )
  | Add ( u , v ) ⇒ Binop' ( Add' , u , v ) when u v : oexpr
  | Mul ( u , v ) ⇒ Binop' ( Mul' , u , v ) when u v : oexpr
  | Abs f ⇒ Abs' f when f : oexpr → oexpr option
  | App ( u , v ) ⇒ App' ( u , v ) when u v : oexpr
```

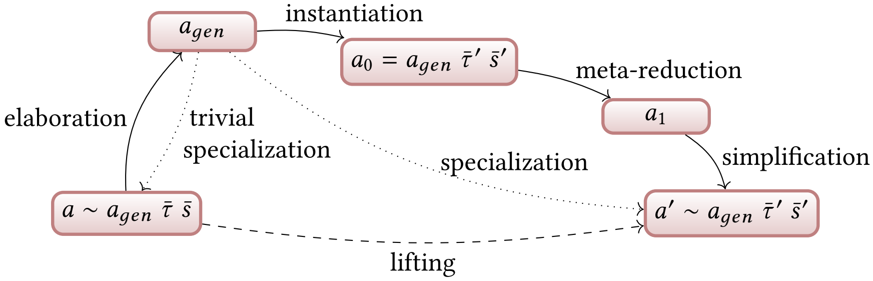
$a_{gen}$
instantiation
$a_0 = a_{gen}\ \bar{\tau}'\ \bar{s}'$
meta-reduction
$a_1$
elaboration
trivial specialization
specialization
simplification
$a \sim a_{gen}\ \bar{\tau}\ \bar{s}$
$a' \sim a_{gen}\ \bar{\tau}'\ \bar{s}'$
lifting

Fig. 1. Overview of the lifting process

In the clause of Abs, the lifting of the argument is specified by an higher-order ornament type oexpr → oexpr option that recursively uses oexpr as argument of another type, and on the left of a an arrow. We can then use this to lift the function eval:

```
let eval' = lifting eval : oexpr → oexpr option
  with ornament * ← @id
 val eval' : expr' → expr' option
```

The annotation **ornament** * ← @id indicates that, for all ornaments that are not otherwise constrained, the identity ornament should be used by default. This is necessary because we create and destruct a tuple in eval, but the type of the tuple does not appear in the signature, so we cannot specify the ornament that should be used through the signature. We give more details in §3.4.

## 3 OVERVIEW OF THE LIFTING PROCESS

Whether used for refactoring or refinement, ornaments are about code reuse. Code reuse is usually obtained by modularity, which itself relies on both type and value abstraction mechanisms. Typically, one writes a generic function *gen* that abstracts over the representation details, say described by some structures $\bar{s}$ of operations on types $\bar{\tau}$. Hence, a concrete implementation $a$ is schematically obtained by the application $gen\ \bar{\tau}\ \bar{s}$; changing the representation to small variation $\bar{s}'$ of types $\bar{\tau}'$ of the structures $\bar{s}$, we immediately obtain a new implementation $gen\ \bar{\tau}'\ \bar{s}'$, say $a'$.

Although the case of ornamentation seems quite different, as we start with a non-modular implementation $a$, we may still get inspiration from the previous schema: modularity through abstraction and polymorphism is the essence of good programming discipline. Instead of directly going from $a$ to $a'$ on some ad hoc track, we may first find a modular presentation of $a$ as an application $a_{gen}\ \bar{\tau}\ \bar{s}$ so that moving from $a$ to $a'$ is just finding the right parameters $\bar{\tau}'$ and $\bar{s}'$ to pass to $a_{gen}$.

This is depicted in Figure 1. In our case, the *elaboration* that finds the generic term $a_{gen}$ is syntactic and only depends on the source term $a$. Hence, the same generic term $a_{gen}$ may be used for different liftings of the same source code. The *specialization* process is actually performed in several steps, as we do not want $a'$ to be just the application $a_{gen}\ \bar{\tau}'\ \bar{s}'$, but be presented in a simplified form as close as possible to the term we started with and as similar as possible to the code the programmer would have manually written. Hence, after instantiation, we perform *meta-reduction*, which eliminates all the abstractions that have been introduced during the elaboration—but not others. This is followed by *simplifications* that will mainly eliminate intermediate pattern matchings.

Having recovered a modular schema, we may use parametricity results, based on logical relations. As long as the arguments $s$ and $s'$ passed to the polymorphic function $a_{gen}$ are related—and they are by the ornamentation relation!—the two applications $a_{gen}\ \bar{\tau}\ \bar{s}$ and $a_{gen}\ \bar{\tau}'\ \bar{s}'$ are also related. Since meta-reduction preserves the relation, it only remains to check that the simplification steps also preserve equivalence to establish a relationship between the bare term $a$ and the lifted term $a'$ (see §7).

The lifting process is formally described in section §9. In the rest of this section, we present it informally on the example of add and append.

### 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 structure as add, and operate on constructors Nil and Cons similarly to the way add proceeds with constructors S and Z.

To help with this transformation, 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, called the *skeleton*, using new constructors Z' and S' corresponding to Z and S but remaining parameterized over the type of the argument of the constructor S:

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

We define the head projection of a list into nat_skel[3] where the head is transformed and the tail stays a list:

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

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, we can pattern match on it in the same way we match on nat in the definition of add. Hence, the definition of append should look like:

```
let rec append_1 m n = match proj_nat_list # m with
  | Z' → n
  | S' m' → ... S' (append_1 m' n) ...
```

In the second branch, we must construct a list out of the hybrid list-nat skeleton S' ($\text{append}_1$ m' n). We use a helper function to inject an 'a list nat_skel into an 'a list :

```
  | S' m' → inj_nat_list_1 (S'(append m' n)) ...
```

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

```
let inj_nat_list : 'a list nat_skel → 'a → 'a list = fun n x #⇒ match n with
  | Z' → Nil
  | S' n' → Cons (x, n')
```

As explained above (§2.2), this supplementation is (**match** m **with** Cons (x, _) → x). and must be user provided as patch #2. Hence, the lifting of add into lists is:

```
let rec append_2 m n = match proj_nat_list # m with
  | Z' → n
  | S' m' → inj_nat_list # (S'(append_2 m' n)) # (match m with Cons (x, _) → x)
```

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

This version is 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 inj_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 inj_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 generate code with fewer indirections that is more similar to hand-written code.

We first perform meta-reduction of $\mathsf{append}_2$, 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 m n =  match (match m with Nil → Z' | Cons (x, m') → S' m') witÂŕh
   | Z' → n
   | S' m' → b              where b is    match S'(append₃ m' n) with
                                            | Z' −> Nil
                                            | S' r' → Cons ((match m with Cons(x, _) → x), r')
```

(The grayed out branch is inaccessible). Still, $\mathsf{append}_3$ computes two pattern matchings that do not appear in the manually written version append. Interestingly, both of them can be eliminated. Extruding the inner match on m in $\mathsf{append}_3$, we get:

```
let rec append₄ 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 know that m is equal to Cons(x,m') in the Cons branch, we simplify *b* to Cons(x, append m' n). After removing all remaining dead branches, we exactly obtain the manually written version append:

```
let rec append = fun m n →
    match m with
      | Nil → n
      | Cons (x, m') → Cons (x, 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 abstracted over all possible instantiations, and only then specialize it to some specific ornamentation by passing encoding and decoding functions as arguments, as well as a set of *patches* that generate the additional data.

Let us detail this process by building the generic lifting add_gen of add, which we remind below.

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

Because they will be passed together to the function, we group the injection and projection into a record:

```
type ('a,'b,'c) orn = { inj : 'a → 'b → 'c; proj : 'c → 'a }
let nat_list = { inj = inj_nat_list ; proj = proj_nat_list ; }
```

The code of append$_2$ could have been written as:

```
let rec append$_2$ m n = match nat_list.proj # m with
  | Z' → n
  | S' m' → nat_list . inj # (S'(add m' n)) # (match m with Cons (x, _) → x)
  in append$_2$
```

Instead of using the concrete ornament nat_list, the generic version abstracts over arbitrary ornaments of nats and over the patch:

```
let add_gen = fun m_orn n_orn p$_1$ #⇒
  let rec add_gen' m n = match m_orn.proj # m with
    | Z' → n
    | S' m' → n_orn.inj # S'(add_gen' m' n) # (p$_1$ # add_gen' # m # m' # n)
  in add_gen'
```

While append$_2$ uses the same ornament nat_list for ornamenting both arguments m and n, this need not be the case in general; hence add_gen has two different ornament arguments m_orn and n_orn. The patch p$_1$ is abstracted over all variables in scope, *i.e.* m, n and m'.

In general, we ask for a different ornament for each occurrence of a constructor or pattern matching on a datatype. We then apply ML inference on the generic term (ignoring the patches) allowing us to deduce that some ornaments encode to the same datatype. In order to preserve the relation between the bare and lifted terms (see §9), these ornaments are merged into a single ornament, with a single record. 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_orn n_orn p$_1$ #⇒
  let rec add_gen' m n = match m_orn.proj # m with
    | Z' → n
    | S' m' → n_orn.inj # S'(add_gen' m' n) # (p$_1$ # add_gen' # m # m' # n)
  in add_gen'
```

The patch p$_1$ describes how to obtain the missing information from the environment (namely add_gen, m, n, m') when building a value of the ornamented type. While the parameters m_orn and n_orn 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 with some specific ornaments. We may for instance 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

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

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

Fig. 2. Kinding rules for ML

trivial functions:

```
let proj_nat_nat = fun x ⇏                       let inj_nat_nat = fun x () ⇏
  match x with Z → Z' | S x → S' x                match x with Z' → Z | S' x → S x
let orn_nat_nat = { proj=proj_nat_nat; inj  = inj_nat_nat }
```

There is no information added, so we may use the following unit_patch for $p_1$:

```
let unit_patch = fun _ _ _ _ ⇏ ()
let add₁ = add_gen # orn_nat_nat # orn_nat_nat # unit_patch
```

As expected, meta-reducing $\mathsf{add}_1$ and simplifying the result returns the original program add.

We may also instantiate the generic lifting with the ornament from nat to lists and the following patch. Meta-reduction of $\mathsf{append}_5$ gives $\mathsf{append}_2$ which can then be simplified to append, as explained above.

```
let orn_nat_list = { proj = proj_nat_list; inj = inj_nat_list }
let append_patch = fun _ m _ _ ⇏ match m with Cons(x, _) → x
let append₅ = add_gen # orn_nat_list # orn_nat_list # append_patch
```

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 #2 corresponds to the argument $p_1$.

### 3.4 Lifting and Ornament Specifications

A lifting definition comes with an optional *ornament signature* and ornamentation instructions which are propagated during instantiation to choose appropriate ornaments of the types appearing in the definition. This process will be described in §9.

During elaboration, liftings of auxiliary functions are chosen among the liftings already defined. Sometimes, a lifting may be required at some type while none or several[4] are available. In such situations, lifting information must also be provided as additional rules. Some patches are ignored: either they do not contain any information or are in a dead branch. In this case, the user need not provide them.

## 4 META ML

As explained above (§3), we elaborate programs into a larger meta-language *m*ML that extends ML with dependent types and separate meta-abstractions and meta-applications. We extend ML in

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

$$\begin{array}{rcl} \kappa & ::= & \mathsf{Typ} \mid \mathsf{Sch} \\ \tau, \sigma & ::= & \alpha \mid \tau \rightarrow \tau \mid \zeta\ \overline{\tau} \mid \forall(\alpha : \mathsf{Typ})\ \tau \\ & & \\ \Gamma & ::= & \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha : \mathsf{Typ} \\ \zeta & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{list} \mid \ldots \end{array}$$

$$\begin{array}{rcl} a, b & ::= & x \mid \mathsf{let}\ x = a\ \mathsf{in}\ a \mid \mathsf{fix}\,(x : \tau)\, x.\, a \mid a\ a \mid a\ \tau \\ & \mid & \Lambda(\alpha : \mathsf{Typ}).\, u \mid d\, \overline{\tau}\ \overline{a} \mid \mathsf{match}\ a\ \mathsf{with}\ \overline{P \rightarrow a} \\ P & ::= & d\, \overline{\tau}\ \overline{x} \\ v & ::= & d\, \overline{\tau}\ \overline{v} \mid \mathsf{fix}\,(x : \tau)\, x.\, a \\ u & ::= & x \mid d\, \overline{\tau}\ \overline{u} \mid \mathsf{fix}\,(x : \tau)\, x.\, a \mid u\ \tau \mid \Lambda(\alpha : \kappa).\, u \\ & \mid & \mathsf{let}\ x = u\ \mathsf{in}\ u \mid \mathsf{match}\ u\ \mathsf{with}\ \overline{P \rightarrow u} \end{array}$$

Fig. 3. Syntax of ML

Var

$$\frac{\vdash \Gamma \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

TAbs

$$\frac{\Gamma, \alpha : \mathsf{Typ} \vdash u : \sigma}{\Gamma \vdash \Lambda(\alpha : \mathsf{Typ}).\, u : \forall(\alpha : \mathsf{Typ})\ \sigma}$$

TApp

$$\frac{\Gamma \vdash \tau : \mathsf{Typ} \qquad \Gamma \vdash a : \forall(\alpha : \mathsf{Typ})\ \sigma}{\Gamma \vdash a\ \tau : \sigma[\alpha \leftarrow \tau]}$$

Fix

$$\frac{\Gamma, x : \tau_1 \rightarrow \tau_2, y : \tau_1 \vdash a : \tau_2}{\Gamma \vdash \mathsf{fix}\,(x : \tau_1 \rightarrow \tau_2)\, y.\, a : \tau_1 \rightarrow \tau_2}$$

App

$$\frac{\Gamma \vdash b : \tau_1 \qquad \Gamma \vdash a : \tau_1 \rightarrow \tau_2}{\Gamma \vdash a\ b : \tau_2}$$

Let-Mono

$$\frac{\Gamma \vdash \tau' : \mathsf{Typ} \qquad \Gamma \vdash a : \tau' \qquad \Gamma, x : \tau' \vdash b : \tau}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau}$$

Let-Poly

$$\frac{\Gamma \vdash \sigma : \mathsf{Sch} \qquad \Gamma \vdash u : \sigma \qquad \Gamma, x : \sigma \vdash b : \tau}{\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ b : \tau}$$

Con

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

Match

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

Fig. 4. Typing rules of ML

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 terms that have an *m*ML typing containing only *e*ML types can be meta-reduced to *e*ML (Theorem 5.38). Then, in an environment without equalities, they can be simplified into ML terms (§7). It is also an important 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.

## 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 $Q_i$ in $Q$ for all $i$ in $I$.

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

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

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

Fig. 5. Reduction rules of ML

### 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[5]. The programmer's language is core ML with recursion and datatypes. Its syntax is described in Figure 3. 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 $\zeta$. 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 3. Well formedness of types and type schemes are asserted by judgments $\Gamma \vdash \tau : \mathsf{Typ}$ and $\Gamma \vdash \tau : \mathsf{Sch}$, defined in Figure 7.

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 $\forall(\alpha_i : \mathsf{Typ})^i\,(\tau_j)^j \to \zeta\,(\alpha_i)^i$. We assume that all datatypes have at least one constructor. 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 types. 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 $\tau_2$ is the function return type.

The language is equipped with a weak (no reduction under binders), left-to-right, call-by-value small-step reduction semantics. The evaluation contexts $E$ and the reduction rules are given in Figure 5. This reduction is written $\longrightarrow_\beta$, and the corresponding head-reduction is written $\longrightarrow^h_\beta$. Reduction must proceed under type abstractions, so that we have a type-erasing semantics.

Typing environments $\Gamma$ contain term variables $x : \tau$ and type variables $\alpha : \mathsf{Typ}$. Well-formedness rules for types and environments are given in figures 2 and 4. 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 $\Gamma$, *i.e.* it is neither in the domain nor in the image of $\Gamma$. 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 $\Gamma \vdash \tau : \mathsf{Sch}$. Although we do not have references, we

[5] The issue of type inference is orthogonal, since the generic lifting is obtained from the typed term.

$$\begin{aligned}\text{let } x = u \text{ in } b &\longrightarrow_{\iota}^{h} b[x \leftarrow u] \\ (\Lambda(\alpha : \mathsf{Typ}).\, u)\ \tau &\longrightarrow_{\iota}^{h} u[\alpha \leftarrow \tau] \\ \text{match } d_j\, \overline{\tau_j}\, (u_i)^i \text{ with } (d_j\, \overline{\tau_j}\, (x_{ji})^i \to \tau_j)^{j \in J} &\longrightarrow_{\iota}^{h} \tau_j[x_{ji} \leftarrow u_i]^i \\ \text{match } d_j\, \overline{\tau_j}\, (u_i)^i \text{ with } (d_j\, \overline{\tau_j}\, (x_{ji})^i \to a_j)^{j \in J} &\longrightarrow_{\iota}^{h} a_j[x_{ji} \leftarrow u_i]^i\end{aligned}$$

$$\textsc{Context-Iota}\quad \frac{a \longrightarrow_{\iota}^{h} b}{C[a] \longrightarrow_{\iota} C[b]}$$

Fig. 6. New reduction rules of *e*ML

still have a form of value restriction: Rule Let-Poly restricts polymorphic binding to *non-expansive terms* $u$, defined in Figure 3, that extend values with type application, pattern matching, and let binding—and whose reduction always terminate. Binding of an expansive term is still allowed (and is heavily used in the elaboration), but its typing is monomorphic (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 in Figure 7. We classify type pattern matching in Sch to prevent it from appearing deep inside types. Typing contexts are extended with type equalities, which are accumulated along pattern matching branches:

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

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 in Figure 8.

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 in Figure 6, 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 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

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

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

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

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

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

Fig. 7. New typing rules for *e*ML

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

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

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

Fig. 8. Equality judgment for *e*ML

$X$ is $\Gamma' \vdash X : Y'$. The context $\Gamma'$ 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.

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.* as equalities. This help with the ornamentation: almost all pre- and post-processing on the terms 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 §5.2).

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 $\sigma_1$ and $\sigma_2$ as follows: consider the two types $\sigma'_i$ equal to $\mathsf{match}\ d_i\ \overline{\tau_i}\ \overline{a_i}\ \mathsf{with}\ d_1\ \overline{\tau_1}\ \overline{a_1}\ \to \sigma_1 \mid d_2\ \overline{\tau_2}\ \overline{a_2}\ \to \sigma_2$. By C-Context and C-Eq, they are equal. But one reduces to $\sigma_1$ and the other to $\sigma_2$. Thus, the code in provably unreachable branches need not be well typed. When writing *e*ML programs, such branches

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

Fig. 9. Syntax of *m*ML

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. We would not have preservation otherwise. Consider for example the following term:

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

It would correctly type if we allowed the use of equalities between expansive terms: 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 dead. But, after one reduction step, the first occurrence of $(\lambda(x : \mathsf{unit}).\mathsf{True})\ ()$ reduces to True, and to prove the incoherence we need to reduce the application in the second occurrence, which we do not want to do to preserve termination of $\longrightarrow_{\iota}$. Thus we need to forbid equalities between terms containing application in a position where it may be evaluated.

Instead of forbidding to introduce equalities between non-expansive terms, we check that equalities are between non-expansive terms when they are used. 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 because the values of some variables are yet unknown. 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, 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

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

Fig. 10. The $\longrightarrow_\sharp$ reduction for *m*ML

$$\overset{\textsc{S-Type}}{\Gamma\vdash \mathsf{Typ} : \mathsf{wf}} \qquad \overset{\textsc{S-Scheme}}{\Gamma\vdash \mathsf{Sch} : \mathsf{wf}} \qquad \overset{\textsc{S-Meta}}{\Gamma\vdash \mathsf{Met} : \mathsf{wf}} \qquad \overset{\textsc{S-VArr}}{\dfrac{\Gamma\vdash\tau:\mathsf{Met}\quad \Gamma\vdash\kappa:\mathsf{wf}}{\Gamma\vdash \tau\rightarrow^\ell \kappa : \mathsf{wf}}}$$

$$\overset{\textsc{S-TArr}}{\dfrac{\Gamma\vdash\kappa_1:\mathsf{wf}\quad \Gamma,\alpha:\kappa_1\vdash\kappa_2:\mathsf{wf}}{\Gamma\vdash \forall(\alpha:\kappa_1)\,\kappa_2 : \mathsf{wf}}}$$

Fig. 11. Well-formedness rules for *m*ML

$$\overset{\textsc{K-Conv}}{\dfrac{\Gamma\vdash\tau_1:\kappa\quad \Gamma\vdash\kappa\simeq\kappa'}{\Gamma\vdash\tau_1:\kappa'}} \qquad \overset{\textsc{K-SubEqu}}{\dfrac{\Gamma\vdash\tau:\mathsf{Sch}}{\Gamma\vdash\tau:\mathsf{Met}}} \qquad \overset{\textsc{K-Pi}}{\dfrac{\Gamma\vdash\tau_1:\mathsf{Met}\quad \Gamma,x^\ell:\tau_1\vdash\tau_2:\mathsf{Met}}{\Gamma\vdash\Pi(x^\ell:\tau_1).\,\tau_2:\mathsf{Met}}}$$

$$\overset{\textsc{K-Forall-Meta}}{\dfrac{\Gamma,\alpha:\kappa\vdash\tau:\mathsf{Met}\quad \Gamma\vdash\kappa:\mathsf{wf}}{\Gamma\vdash\forall^\sharp(\alpha:\kappa).\,\tau:\mathsf{Met}}} \qquad \overset{\textsc{K-Pi-Eq}}{\dfrac{\Gamma\vdash a:\tau'\quad \Gamma\vdash b:\tau'\quad \Gamma\vdash\tau':\mathsf{Sch}\quad \Gamma,(a=_{\tau'}b)\vdash\tau:\mathsf{Met}}{\Gamma\vdash\Pi(\diamond:a=_{\tau'}b).\,\tau:\mathsf{Met}}}$$

$$\overset{\textsc{K-TLam}}{\dfrac{\Gamma,\alpha:\kappa_1\vdash\tau:\kappa_2}{\Gamma\vdash\Lambda^\sharp(\alpha:\kappa_1).\,\tau:\forall(\alpha:\kappa_1)\,\kappa_2}} \qquad \overset{\textsc{K-TApp}}{\dfrac{\Gamma\vdash\tau_1:\forall(\alpha:\kappa_a)\,\kappa_b\quad \Gamma\vdash\tau_2:\kappa_a}{\Gamma\vdash\tau_1\,\sharp\,\tau_2:\kappa_b[\alpha\leftarrow\tau_2]}}$$

$$\overset{\textsc{K-VLam}}{\dfrac{\Gamma\vdash\tau_1:\mathsf{Met}\quad \Gamma,x:\tau_1\vdash\tau_2:\kappa_2}{\Gamma\vdash\lambda^\sharp(x:\tau_1).\,\tau_2:\tau_1\rightarrow\kappa_2}} \qquad \overset{\textsc{K-VApp}}{\dfrac{\Gamma\vdash\tau_1:\tau_2\rightarrow\kappa_2\quad \Gamma\vdash a:\tau_2}{\Gamma\vdash\tau_1\,\sharp\,a:\kappa_2}}$$

Fig. 12. Kinding rules for *m*ML

$$\overset{\textsc{TAbs-Meta}}{\dfrac{\Gamma,\alpha:\kappa\vdash a:\tau}{\Gamma\vdash\Lambda^\sharp(\alpha:\kappa).\,a:\forall^\sharp(\alpha:\kappa).\,\tau}} \qquad \overset{\textsc{TApp-Meta}}{\dfrac{\Gamma\vdash a:\forall^\sharp(\alpha:\kappa).\,\tau_1\quad \Gamma\vdash\tau_2:\kappa}{\Gamma\vdash a\,\sharp\,\tau_2:\tau_1[\alpha\leftarrow\tau_2]}} \qquad \overset{\textsc{Abs-Meta}}{\dfrac{\Gamma\vdash\tau_1:\mathsf{Met}\quad \Gamma,x:\tau_1\vdash a:\tau_2}{\Gamma\vdash\lambda^\sharp(x:\tau_1).\,a:\Pi(x:\tau_1).\,\tau_2}}$$

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

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

Fig. 13. Typing and equality rules for *m*ML

*m*ML reduction were interleaved, *i.e.* as if the encoding and decodings of ornaments were called at runtime, but they happen at ornamentation time.

The syntax of *m*ML is described in Figure 9. We only describe the differences with *e*ML. Terms are extended with meta-abstractions and the corresponding meta-applications on types, equalities, and non-expansive terms, while types are extended with meta-abstractions and meta-applications on types and non-expansive terms. Both meta-abstractions and meta-applications are marked with $\sharp$ to distinguish them from ML abstractions and applications. 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 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. It is important that a non-expansive term remains non-expansive after substitution. Therefore, we may only allow substitution by non-expansive terms. In particular, arguments of redexes in Figure 10 must be non-expansive. To ensure that meta-redexes can still always be reduced before other redexes, arguments of meta-applications are syntactically restricted to non-expansive terms. 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. The meta-reduction, written $\longrightarrow_\sharp$, is defined in Figure 10. It is a strong reduction, allowed under 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 in Figure 13. The new kinding rules for type-level abstraction and application are given in Figure 12. 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 $\Gamma \vdash \kappa : \mathrm{wf}$ and defined on Figure 11.

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 the encoding of ornaments.

We write $\longrightarrow$ for the union of $\longrightarrow_\beta$, $\longrightarrow_\iota$, and $\longrightarrow_\sharp$, and $\longrightarrow^*$ for 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 ornamentation time.

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

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

Fig. 14. Stricter rules for equality

## 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 14. 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.

## 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 (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 $C_{\mathrm{a}}$ 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 $C_{\mathrm{t}}$ the set of reducbility sets of types, and $C_{\mathrm{k}}$ 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 15

On Figure 16, we also define an interpretation $[\![\kappa]\!]_{\rho}$ of kinds as sets of types and an interpretation $[\![\tau]\!]_{\rho}$ of a type $\tau$ under an assignment $\rho$ of reducibility sets to type variables by mutual induction

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

Fig. 15. Interpretation of kinds as sets of interpretations

$$
\begin{array}{rcl}
\llbracket \mathsf{Typ} \rrbracket_{\rho} \;=\; \llbracket \mathsf{Sch} \rrbracket_{\rho} & = & \llbracket \mathsf{Met} \rrbracket_{\rho} \;=\; \mathcal{N}_{\mathrm{t}} \\
\llbracket \forall(\alpha : \kappa_1)\,\kappa_2 \rrbracket_{\rho} & = & \{\tau \in \mathcal{N}_{\mathrm{t}} \mid \forall \tau' \in \llbracket \kappa_1 \rrbracket_{\rho}, \tau \sharp \tau' \in \llbracket \kappa_2 \rrbracket_{\rho[\alpha \leftarrow \llbracket \tau' \rrbracket_{\rho}]}\} \\
\llbracket \tau \longrightarrow \kappa \rrbracket_{\rho} & = & \{\tau \in \mathcal{N}_{\mathrm{t}} \mid \forall u \in \llbracket \tau \rrbracket_{\rho}, \tau \sharp u \in \llbracket \kappa \rrbracket_{\rho}\} \\
\llbracket (a_1 =_{\tau} a_2) \longrightarrow \kappa \rrbracket_{\rho} & = & \{\tau \in \mathcal{N}_{\mathrm{t}} \mid \tau \sharp \diamond \in \llbracket \kappa \rrbracket_{\rho}\}
\end{array}
$$

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

Fig. 16. Interpretation of kinds and types as sets of types and terms

*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.

- 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 $\llbracket \kappa \rrbracket_{\rho}$ is well-defined, and $\llbracket \kappa \rrbracket_{\rho} \in \mathcal{C}_{\mathrm{t}}$.*
- *If $\Gamma \vdash \tau : \kappa$, then $\llbracket \tau \rrbracket_{\rho}$ is well-defined, and we have $\llbracket \tau \rrbracket_{\rho} \in \langle\!\langle \kappa \rangle\!\rangle$.*

Proof. By simultaneaous induction on the sorting and kinding derivations. The case of all syntax-directed rules whose output is interpreted as $\mathcal{N}_{\mathrm{a}}$ or $\mathcal{N}_{\mathrm{t}}$ is follows by Lemma 5.4. For the variable rule, use the definition of $\rho \vDash \Gamma$ applied to the variable. For abstraction, abstract, add the interpretation to the context and interpret. For application, use the type of $\llbracket \kappa \rrbracket_{\rho}$ 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 $S = \{a \in \mathcal{N}_\mathrm{a} \mid \forall u \in S_1, a \sharp u \in S_2\}$.

- C1 $S$ is a subset of $\mathcal{N}_\mathrm{a}$.
- 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 $\tau$, $\kappa$, $\tau'$, $\alpha$, and $\rho$, we have both $\llbracket \tau \rrbracket_{\rho[\alpha \leftarrow \llbracket \tau' \rrbracket_\rho]} = \llbracket \tau[\alpha \leftarrow \tau'] \rrbracket_\rho$ and $\llbracket \kappa \rrbracket_{\rho[\alpha \leftarrow \llbracket \tau' \rrbracket_\rho]} = \llbracket \kappa[\alpha \leftarrow \tau'] \rrbracket_\rho$.*

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 $\tau$, $\tau'$, $\kappa$, $\kappa'$ are well-kinded (or well-formed) in $\Gamma$. Then $\llbracket \tau \rrbracket_\rho = \llbracket \tau' \rrbracket_\rho$ whenever $\tau \longrightarrow \tau'$ and $\llbracket \kappa \rrbracket_\rho = \llbracket \kappa' \rrbracket_\rho$ whenever $\kappa \longrightarrow \kappa'$.*

Proof. By structural induction on the context in which head reduction occurs. 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 $(\lambda S_\alpha \in \langle\!\langle \kappa \rangle\!\rangle.\ \llbracket \tau \rrbracket_{\rho[\alpha \leftarrow S_\alpha]})\ \llbracket \tau' \rrbracket_\rho = \llbracket \tau \rrbracket_{\rho[\alpha \leftarrow \llbracket \tau' \rrbracket_\rho]}$ and the interpretation of the right-hand side is $\llbracket \tau[\alpha \leftarrow \tau'] \rrbracket_\rho = \llbracket \tau \rrbracket_{\rho[\alpha \leftarrow \llbracket \tau' \rrbracket_\rho]}$ 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 $\llbracket \tau_1 \rrbracket_\rho = \llbracket \tau_2 \rrbracket_\rho$. If $\Gamma \vdash \kappa_1 \simeq \kappa_2$, then $\llbracket \kappa_1 \rrbracket_\rho = \llbracket \kappa_2 \rrbracket_\rho$*

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 $\gamma(a) \in [\![\tau_1 \longrightarrow \tau_2]\!]_\rho$ 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 $\tau_2$ 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 $\tau_1'$ 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 $\Gamma$. We can take the identity substitution $\gamma(x) = x$ for all $x \in \Gamma$ and apply the fundamental lemma. All interpretations are subsets of $\mathcal{N}_\mathrm{a}$, thus $X \in \mathcal{N}_\mathrm{a}$. □

### 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.

$$\frac{}{\emptyset \rhd \emptyset}\;\textsc{WEnv-Empty} \qquad \frac{\Gamma_1 \rhd \Gamma_2}{\Gamma_1 \rhd \Gamma_2, x^{\ell} : \tau}\;\textsc{WEnv-Weaken-Var} \qquad \frac{\Gamma_1 \rhd \Gamma_2 \qquad \Gamma_2 \vdash \tau_1 \simeq \tau_2}{\Gamma_1, x^{\ell} : \tau_1 \rhd \Gamma_2, x^{\ell} : \tau_2}\;\textsc{WEnv-Conv-Var} \qquad \frac{\Gamma_1 \rhd \Gamma_2}{\Gamma_1 \rhd \Gamma_2, \alpha^{\ell} : \kappa}\;\textsc{WEnv-Weaken-TVar}$$

$$\frac{\Gamma_1 \rhd \Gamma_2 \qquad \Gamma_2 \vdash \kappa_1 \simeq \kappa_2}{\Gamma_1, \alpha : \kappa_1 \rhd \Gamma_2, \alpha : \kappa_2}\;\textsc{WEnv-Conv-TVar} \qquad \frac{\Gamma_1 \rhd \Gamma_2}{\Gamma_1 \rhd \Gamma_2, (a =_{\tau} b)}\;\textsc{WEnv-Weaken-Eq}$$

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

Lemma 5.13. *Weakening is reflexive and transitive: for all well-formed environments $\Gamma_1$, $\Gamma_2$ and $\Gamma_3$, 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 $\Gamma_1$, $\Gamma_2$, $\Gamma'_1$, $\Gamma'_2$ 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 $\tau_2$ 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 $\mathsf{let}\ x = a_2$ 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: let $x = a$ in $b$ and let $x = a'$ 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 compound types or kinds into a conversion between their subtypes or subkinds. For example, from $\Gamma \vdash \tau_1 \to \tau_2 \simeq \tau_1' \to \tau_2'$, we need to extract $\Gamma \vdash \tau_1 \simeq \tau_1'$ and $\Gamma \vdash \tau_2 \simeq \tau_2'$. The easiest way to prove this 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 $\Gamma$ 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, $\kappa'$ and $\kappa'$ 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 $\kappa$ 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 $\Gamma$.*

- *If $\Gamma \vdash \lambda^\sharp (x : \tau_1').\, \tau_2 : \tau_1 \to \kappa$, 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* $\Gamma \vdash \lambda^{\sharp}(\diamond : a_1' =_{\tau'} a_2').\, \tau'' : (\diamond : a_1 =_{\tau} a_2) \to \kappa$, *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 $\Gamma \vdash \tau_1 \to \kappa \simeq \tau_1'' \to \kappa'$. 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 $\tau = (\lambda^{\sharp}(x : \tau_1).\, \tau_2) \mathbin{\sharp} u$ to $\tau' = \tau_2[x \leftarrow u]$.

Since the last rule of $\Gamma \vdash (\lambda^{\sharp}(x : \tau_1).\, \tau_2) \mathbin{\sharp} u : \kappa$ is syntax-directed, we can invert it and obtain $\Gamma \vdash u : \tau_1$ and $\Gamma \vdash \lambda^{\sharp}(x : \tau_1).\, \tau_2 : \tau_1 \to \kappa$, 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:*

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

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

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

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

Proof. We prove a stronger result: suppose $\Gamma \vdash X_1 \simeq X_2$, and $X_1', X_2'$ are the $\longrightarrow_{\sharp}^{t}$ 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 $\longrightarrow_{\sharp}^{t}$ 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 $\alpha$): $C[x]$ 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.

$$\begin{array}{cc}
\forall^\sharp(\alpha : \kappa).\, \tau \blacktriangleright \forall^\sharp(\alpha : \_).\, \_ \circ (\kappa, \tau) & \Pi(x : \tau).\, \tau' \blacktriangleright \Pi(x : \_).\, \_ \circ (\tau, \tau') \\
\Pi(\diamond : b =_\tau b').\, a \blacktriangleright \Pi(\diamond : \_ =_\_ \_).\, \_ \circ (b, b', \tau, a) & \forall(\alpha : \mathsf{Typ})\, \tau \blacktriangleright \forall(\alpha : \mathsf{Typ})\, \_ \circ (\tau) \\
\tau_1 \longrightarrow \tau_2 \blacktriangleright \_ \rightarrow_{\mathsf{tt}} \_ \circ (\tau_1, \tau_2) & \zeta\, (\tau_i)^i \blacktriangleright \zeta\, \_ \circ (\tau_i)^i
\end{array}$$

The meta-heads are all heads that can not be generated by ML reduction in well-kinded types, *i.e.* all except $\forall(\alpha : \mathsf{Typ})\, \_$, $\_ \rightarrow_{\mathsf{tt}} \_$ 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 $\longrightarrow^t_\sharp$-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 $\longrightarrow^t_\sharp$-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 $\tau$ and $\tau'$. We have $\tau' = \mathsf{match}\ d_j(u_i)^i\ \mathsf{with}\ (d_j(x_{ij})^i \to \tau_j)^{j \in J}$ and $\tau = \tau_j[x_{ij} \leftarrow u_i]^i$. The term $\tau_j$ has the same head as $\tau$. Moreover, inverting the last syntactic rule of a kinding derivation for $\tau'$, 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* $\Gamma$.

- *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* $\Gamma$ *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:*

C-Refl

$$\frac{\Gamma \vdash X : Y}{\Gamma \vdash^{n} X \simeq X}$$

C-Sym

$$\frac{\Gamma \vdash^{n} X_1 \simeq X_2}{\Gamma \vdash^{n} X_2 \simeq X_1}$$

C-Trans

$$\frac{\Gamma \vdash^{n} X_1 \simeq X_2 \qquad \Gamma \vdash^{n} X_2 \simeq X_3}{\Gamma \vdash^{n} X_1 \simeq X_3}$$

C-Red-Iota'

$$\frac{X_1 \longrightarrow_{\iota} X_2 \qquad \Gamma \vdash X_1 : Y_1 \qquad \Gamma \vdash X_2 : Y_2}{\Gamma \vdash^{n} X_1 \simeq X_2}$$

C-Red-Meta'

$$\frac{X_1 \longrightarrow_{\sharp}^{a} X_2 \qquad \Gamma \vdash X_1 : Y_1 \qquad \Gamma \vdash X_2 : Y_2}{\Gamma \vdash^{n} X_1 \simeq X_2}$$

C-Context

$$\frac{\Gamma \vdash C[\Gamma' \vdash X_1 : Y'] : Y \qquad \Gamma \vdash^{n} X_1 \simeq X_2}{\Gamma \vdash^{n} C[X_1] \simeq C[X_2]}$$

C-Eq

$$\frac{a_1 \longrightarrow_{\sharp}^{*} u_1 \qquad a_2 \longrightarrow_{\sharp}^{*} u_2 \qquad (a_1 =_{\tau} a_2) \in \Gamma}{\Gamma \vdash^{n} u_1 \simeq u_2}$$

C-Split

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

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 $\tau$ and $\tau'$ 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 $\mathsf{Any}_{\mathsf{a}}$ to stand for any well-typed term, $\mathsf{Any}_{\mathsf{t}}$ for a well-kinded type, and $\mathsf{Any}_{\mathsf{k}}$ 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 $\tau$ 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 $\tau$).

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.
- 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 $\tau = \mathsf{match}\ d_j(u_i)^i\ \mathsf{with}\ (d_k(x_{ki})^i \to \tau_k)^k$ and $\tau' = \tau_j[x_{ji} \leftarrow u_i]^i$. Then, compute the tail: $\mathsf{tails}(h; \tau)_l = \mathsf{match}\ d_j(u_i)^i\ \mathsf{with}\ (d_k(x_{ki})^i \to X_{kl})^k$ where $X_{kl} = \mathsf{tails}(h; \tau_k)_l$ and

$$\begin{aligned}
D_{\mathsf{t}} &::= [\,] \sharp a \mid [\,] \sharp \tau \mid [\,] \sharp \diamond \\
D_{\mathsf{v}} &::= [\,] \sharp a \mid [\,] \sharp \tau \mid [\,] \sharp \diamond \mid [\,]\ a \mid [\,]\ \tau \mid \mathsf{match}\ [\,]\ \mathsf{with}\ \overline{P \to a} \\
c_{\mathsf{t}} &::= \lambda^{\sharp}(x : \tau).\, \tau \mid \Lambda^{\sharp}(\alpha : \kappa).\, \tau \mid \lambda^{\sharp}(\diamond : a =_{\tau} a).\, \tau \\
c_{\mathsf{v}} &::= \lambda^{\sharp}(x : \tau).\, a \mid \lambda^{\sharp}(x : \tau).\, a \mid \Lambda^{\sharp}(\alpha : \kappa).\, a \mid \lambda^{\sharp}(\diamond : a =_{\tau} a).\, a \\
&\quad \mid \lambda(x : \tau).\, a \mid \mathsf{fix}\,(x : \tau)\, x.\, a \mid \Lambda(\alpha : \mathsf{Typ}).\, a \mid d(\overline{a})
\end{aligned}$$

Fig. 17. Destructors and constructors

$\mathsf{tails}(h; \tau') = \mathsf{tails}(h; \tau_j)[x_{ji} \leftarrow u_i]^i$. 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 $C = \mathsf{match}\ C'\ \mathsf{with}\ (P_i \to \tau_i)^i$ 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 $C = \mathsf{match}\ a\ \mathsf{with}\ (P_i \to \tau_i)^i \mid P_j \to C'$, 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 $\Gamma$ 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 $\Gamma$. 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 17) constructors $c_{\mathsf{t}}$ and $c_{\mathsf{v}}$ at the level of types and terms, and destructor contexts, or simply destructors, $D_{\mathsf{t}}$ and $D_{\mathsf{v}}$ 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 $\Gamma$ be an environment. Then:*

- *If $\Gamma \vdash D_{\mathsf{t}}[c_{\mathsf{t}}] : Y$, then $D_{\mathsf{t}}[c_{\mathsf{t}}] \longrightarrow^{h}$.*
- *If $\Gamma \vdash D_{\mathsf{v}}[c_{\mathsf{v}}] : Y$, then $D_{\mathsf{v}}[c_{\mathsf{v}}] \longrightarrow^{h}$.*

Proof. By case analysis on the destructor. We consider the case $D_{\mathrm{t}} = [\,] \sharp \tau$. Consider the various cases for $c_{\mathrm{t}}$: by Lemma 5.28, the only possible case is $c_{\mathrm{t}} = \Lambda^{\sharp}(\alpha : \kappa).\ \tau'$. Then, $D_{\mathrm{t}}[c_{\mathrm{t}}]$ 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* $\Gamma$*; 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 $\tau$, $u[\alpha \leftarrow \tau]$ is good. Goodness is stable by reduction.

We will prove the following property by induction on a term, type or kind $X$: suppose $\gamma$ 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 $\gamma$, 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 $\longrightarrow_\iota$ in Figure 18. It includes an interpretation $\mathbf{V}[\tau]_\gamma$ of values of type $\tau$, 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 $\tau$, 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 $\gamma$ is a well-typed environment that models $\Gamma$, and $\vdash \gamma_1 \simeq \gamma_2$ if all components of $\gamma_1$ and $\gamma_2$ 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 $\gamma$ is valid if for all $\alpha$ such that $\gamma(\alpha) = (\tau, S, R)$,

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

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

$$
\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 & = & \left\{\left(\begin{array}{l} \Lambda(\alpha : \mathsf{Typ}).\, v_1, \\ \Lambda(\alpha : \mathsf{Typ}).\, v_2 \end{array}\right) \;\middle|\; \begin{array}{l} \forall\,(\emptyset \vdash \tau' : \mathsf{Typ}) \\ \qquad (v_1[\alpha \leftarrow \tau'], v_2[\alpha \leftarrow \tau']) \in \mathbf{V}[\tau]_{\gamma[\alpha \leftarrow \mathbf{EqE}[\tau']_\gamma]} \end{array}\right\} \\
\mathbf{EqV}[\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 \\ \qquad \wedge((v_j, w_j) \in \mathbf{EqV}[\tau_j[\alpha_i \leftarrow \tau_i]^i]_\gamma)^j \end{array}\right\}
\end{array}
$$

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

Fig. 18. Logical relation for $\longrightarrow_\iota$

- 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:*

- *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 $\gamma$. 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 $\tau$. □

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 $\gamma$, 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 $\tau$. The term variables in $\gamma$ 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 $\gamma$ and $\tau_1, S, R$ 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 $\tau$, $\mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\tau_1, S, R)]} = \mathbf{E}[\tau]_{\gamma[\alpha \leftarrow (\tau_2, S, R)]}$;*
- *for all types $\tau$, $\mathbf{EqE}[\tau]_{\gamma[\alpha \leftarrow (\tau_1, S, R)]} = \mathbf{EqE}[\tau]_{\gamma[\alpha \leftarrow (\tau_2, S, R)]}$.*

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

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 $\tau''$ 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 $\tau_j$ 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 $\gamma_1$ 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. □

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 $\tau_1$ and $\tau_2$ 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 $\emptyset \vdash D_{\mathrm{t}}[c_{\mathrm{t}}] : Y$, then $D_{\mathrm{t}}[c_{\mathrm{t}}] \longrightarrow^h$.*
- *If $\emptyset \vdash D_{\mathrm{v}}[c_{\mathrm{v}}] : Y$, then $D_{\mathrm{v}}[c_{\mathrm{v}}] \longrightarrow^h$.*

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 $\longmapsto$ of $\longrightarrow$ by restricting the reduction $\longrightarrow_\sharp$ to a call-by-name left-to-right strategy, defined in Figure 19. 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 $\longmapsto$ defined as the ML and meta head-reductions for terms, applied under the extended ML evaluation contexts $E$. We write $\longmapsto^h$ the associated head-reduction, *i.e.* the union of all head-reduction of terms.

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

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

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

Fig. 19. Deterministic reduction $\mapsto$ for *m*ML

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 $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$: 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 $\mathsf{let}\ x = E'[b]\ \mathsf{in}\ a_2$.
- $a$ 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, $\beta$-reductions and $\iota$-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 $i \leq n$, 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.

$$\begin{aligned}
(\mathsf{fix}\,(x:\tau)\,y.\,a)\;v &\longmapsto_1^h a[x \leftarrow \mathsf{fix}\,(x:\tau)\,y.\,a, y \leftarrow v] \\
(\Lambda(\alpha : \mathsf{Typ}).\,v)\;\tau &\longmapsto_0^h v[\alpha \leftarrow \tau] \\
\mathsf{let}\ x = v\ \mathsf{in}\ a &\longmapsto_0^h a[x \leftarrow v] \\
\begin{array}{r}\mathsf{match}\ d_j\,\overline{\tau_j}\,(v_i)^i\ \mathsf{with} \\ (d_j\,\overline{\tau_j}\,(x_{ji})^i \to a_j)^j\end{array} &\longmapsto_0^h a_j[x_{ij} \leftarrow v_i]^i \\
(\lambda^\sharp(x:\tau).\,a)\,\sharp\,u &\longmapsto_0^h a[x \leftarrow u] \\
(\Lambda^\sharp(\alpha:\kappa).\,a)\,\sharp\,\tau &\longmapsto_0^h a[\alpha \leftarrow \tau] \\
(\lambda^\sharp(\diamond : b_1 =_\tau b_2).\,a)\,\sharp\,\diamond &\longmapsto_0^h a
\end{aligned}$$

$$\textsc{Context}\ \frac{a \longmapsto_i^h b}{E[a] \longmapsto_i E[b]} \qquad \textsc{Identity}\ \frac{}{a \longmapsto_0 a}$$

$$\textsc{Composition}\ \frac{a_1 \longmapsto_i a_2 \qquad a_2 \longmapsto_j a_3}{a_1 \longmapsto_{i+j} a_3}$$

Fig. 20. The counting reduction $\longmapsto_i$

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 $a \longmapsto$. 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 $\tau$, $b$ 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 $\longmapsto$. Otherwise, suppose $a_1$ is not already a value. Then, by induction hypothesis it reduces by $\longmapsto$, and $a$ reduces. Otherwise, we have $a = (\lambda(x:\tau).\,b)\ v$, and $a$ is head-reducible.
- 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 $\longmapsto$. Thus, $a$ reduces by $\longmapsto$.
- 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 $b \longmapsto^* d(a_i')^i$. 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, $\longmapsto_i$ 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 in Figure 20. Since $\longmapsto_0$ 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.

$$\textsc{Env-TVar}\ \frac{\vdash \gamma : \Gamma \qquad \emptyset \vdash \tau : \gamma(\kappa)}{\vdash \gamma[\alpha \leftarrow \tau] : \Gamma, \alpha : \kappa} \qquad \textsc{Env-Var-NonExp}\ \frac{\vdash \gamma : \Gamma \qquad \emptyset \vdash u : \gamma(\tau)}{\vdash \gamma[x \leftarrow u] : \Gamma, x : \tau} \qquad \textsc{Env-Eq}\ \frac{\vdash \gamma : \Gamma \qquad \emptyset \vdash \gamma(a) \simeq \gamma(b)}{\vdash \gamma : \Gamma, (a =_\tau b)}$$

$$\textsc{Env-Empty}\ \vdash \emptyset : \emptyset$$

Fig. 21. The environment typing judgment $\vdash \gamma : \Gamma$

$$\begin{aligned}
\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}
\end{aligned}$$

Fig. 22. Interpretation of kinds

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 $\gamma$ 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 $\kappa$, $\gamma_1$, $\gamma_2$ such that there exists $\Gamma$ 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.*

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 $\longmapsto$. 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 $\gamma$. 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 $\Gamma$ are interpreted as a set of environments $\gamma$ 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 compacity, we will only specify the value of $R_i$ and leave implicit the values of $\tau_1$ and $\tau_2$ (that are simply obtained by applying $\gamma_1, \gamma_2$ to the type $\tau$).

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 $\Gamma$ 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 $\Gamma \vdash \tau :$ 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 $\gamma$, index $k$, we have:*

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

$$
\begin{array}{lcl}
\mathcal{G}_k[\emptyset] & = & \{\emptyset\} \\
\mathcal{G}_k[\Gamma, x : \tau] & = & \{\gamma[x \leftarrow (u_1, u_2)] \mid (u_1, u_2) \in \mathcal{E}_k[\tau]_\gamma \wedge \gamma \in \mathcal{G}_k[\Gamma]\} \\
\mathcal{G}_k[\Gamma, \alpha : \kappa] & = & \{\gamma[\alpha \leftarrow (\tau_1, (R_j)^{j \leq k}, \tau_2)] \mid (\tau_1, (R_j)^{j \leq k}, \tau_2) \in \mathcal{K}[\kappa]_{\gamma_1, \gamma_2} \wedge \gamma \in \mathcal{G}_k[\Gamma]\} \\
\mathcal{G}_k[\Gamma, (a_1 =_\tau a_2)] & = & \{\gamma \in \mathcal{G}_k[\Gamma] \mid (\vdash \gamma_1(a_1) \simeq \gamma_1(a_2)) \wedge (\vdash \gamma_2(a_1) \simeq \gamma_2(a_2))\} \\
\mathcal{E}_k[\tau]_\gamma & = & \{(a_1, a_2) \mid \forall i,\ \forall v_2,\ a_2 \mapsto_i v_2 \implies \exists v_1,\ a_1 \mapsto^* v_1 \wedge (v_1, v_2) \in \mathcal{V}_{k-i}[\tau]_\gamma\} \\
\mathcal{V}_k[\alpha]_\gamma & = & \gamma(\alpha) \\
\mathcal{V}_k[\tau_1 \sharp \tau_2]_\gamma & = & \mathcal{V}_k[\tau_1]_\gamma\ \mathcal{V}_k[\tau_2]_\gamma \\
\mathcal{V}_k[\tau \sharp u]_\gamma & = & \mathcal{V}_k[\tau]_\gamma\ (\gamma_1(u), \gamma_2(u)) \\
\mathcal{V}_k[\tau \sharp \diamond]_\gamma & = & \mathcal{V}_k[\tau]_\gamma\ \bullet \\
\mathcal{V}_k[\Lambda^\sharp(\alpha : \kappa).\ \tau]_\gamma & = & \lambda(\mathcal{R} \in \mathcal{K}[\kappa]_{\gamma_1, \gamma_2}).\ \mathcal{V}_k[\tau]_{\gamma[\alpha \leftarrow \mathcal{R}]} \\
\mathcal{V}_k[\lambda^\sharp(x : \kappa).\ \tau]_\gamma & = & \lambda((u_1, u_2) \in \mathsf{Term} \times \mathsf{Term}).\ \mathcal{V}_k[\tau]_{\gamma[x \leftarrow (u_1, u_2)]} \\
\mathcal{V}_k[\lambda^\sharp(\diamond : a_1 =_{\tau_2} a_2).\ \tau_1]_\gamma & = & \lambda(\bullet \in \mathbf{1}).\ \mathcal{V}_k[\tau_1]_\gamma
\end{array}
$$

$$
\mathcal{V}_k[\tau_1 \to \tau_2]_\gamma = \left\{ \begin{pmatrix} \mathsf{fix}\,(x : \tau_1' \to \tau_2')\, y.\ a_1, \\ \mathsf{fix}\,(x : \tau_1'' \to \tau_2'')\, y.\ a_2 \end{pmatrix} \,\middle|\, \begin{array}{l} \forall\,(j < k)\ (v_1, v_2) \in \mathcal{V}_j[\tau_1]_\gamma \implies \\ \begin{pmatrix} a_1[x \leftarrow (\mathsf{fix}\,(x : \tau_1' \to \tau_2')\, y.\ a_1), y \leftarrow v_1], \\ a_1[x \leftarrow (\mathsf{fix}\,(x : \tau_1'' \to \tau_2'')\, y.\ a_2), y \leftarrow v_2] \end{pmatrix} \in \mathcal{E}_j[\tau_2]_\gamma \end{array} \right\}
$$

$$
\begin{array}{lcl}
\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 \leq 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{array}{c} (\emptyset \vdash \gamma_1(a_1) \simeq \gamma_1(a_2) \wedge \emptyset \vdash \gamma_2(a_1) \simeq \gamma_2(a_2)) \\ \implies (a_1, a_2) \in \mathcal{E}_k[\tau']_\gamma \end{array} \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 \leq 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 \leq 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 \leq 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 \leq 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}{rl} & (d : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\,(\alpha_i)^i) \\ \wedge & \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. 23. Definition of the logical relation

Proof. By structural induction on $\tau$, $\kappa$. □

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:

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

  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:

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

  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:

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

  Consider $\gamma \in \mathcal{G}_k[\Gamma]$. We want to prove $(\mathsf{fix}\,(x : \gamma_1(\tau_1) \longrightarrow \gamma_1(\tau_2))\,y.\,\gamma_1(a), \mathsf{fix}\,(x : \gamma_2(\tau_1) \longrightarrow \gamma_2(\tau_2))\,y.\,\gamma_2(a)) \in \mathcal{V}_k[\tau_1 \longrightarrow \tau_2]_\gamma$. Consider $j < k$, and $(v_1, v_2) \in \mathcal{V}_j[\tau_1]_\gamma$. We need to show:
  $(\gamma_1(a)[x \leftarrow \mathsf{fix}\,(x : \gamma_1(\tau_1) \longrightarrow \gamma_1(\tau_2))\,y.\,\gamma_1(a), y \leftarrow v_1],$
  $\gamma_2(a)[x \leftarrow \mathsf{fix}\,(x : \gamma_2(\tau_1) \longrightarrow \gamma_2(\tau_2))\,y.\,\gamma_2(a), y \leftarrow v_2]) \in \mathcal{V}_j[\tau_2]_\gamma$ Note that by weakening, $\gamma \in \mathcal{G}_j[\Gamma]$. Moreover, by induction hypothesis at rank $j < k$, $(\mathsf{fix}\,(x : \gamma_1(\tau_1) \longrightarrow \gamma_1(\tau_2))y.\gamma_1(a), \mathsf{fix}\,(x : \gamma_2(\tau_1) \longrightarrow \gamma_2(\tau_2))\,y.\,\gamma_2(a)) \in \mathcal{V}_j[\tau_1 \longrightarrow \tau_2]_\gamma$. Consider
  $\gamma' = \gamma[x \leftarrow (\mathsf{fix}\,(x : \gamma_1(\tau_1) \longrightarrow \gamma_1(\tau_2))\,y.\,\gamma_1(a),$
  $\mathsf{fix}\,(x : \gamma_2(\tau_1) \longrightarrow \gamma_2(\tau_2))\,y.\,\gamma_2(a)), y \leftarrow (v_1, v_2)]$. Then, $\gamma' \in \mathcal{G}_j[\Gamma, x : \tau_1 \longrightarrow \tau_2, y : \tau_1]$. Thus, by induction hypothesis at rank $j < k$, $(\gamma'_1(a), \gamma'_2(a)) =$
  $(\gamma_1(a)[x \leftarrow \mathsf{fix}\,(x : \gamma_1(\tau_1) \longrightarrow \gamma_1(\tau_2))\,y.\,\gamma_1(a), y \leftarrow v_1]$
  $, \gamma_2(a)[x \leftarrow \mathsf{fix}\,(x : \gamma_2(\tau_1) \longrightarrow \gamma_2(\tau_2))\,y.\,\gamma_2(a), y \leftarrow v_2]) \in \mathcal{V}_j[\tau_2]_{\gamma'} = \mathcal{V}_j[\tau_2]_\gamma$.

- Consider the App rule:

$$\textsc{App}\quad \frac{\Gamma \vdash b : \tau_1 \qquad \Gamma \vdash a : \tau_1 \longrightarrow \tau_2}{\Gamma \vdash a\ b : \tau_2}$$

  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 $(w_1, w_2) \in \mathcal{V}_{k-i_1}[\tau_1 \longrightarrow \tau_2]_\gamma$ 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 $w_1 = \mathsf{fix}\,(x : \tau'_1 \longrightarrow \tau'_2)\,y.\,a'_1$ and $w_2 = \mathsf{fix}\,(x : \tau''_1 \longrightarrow \tau''_2)\,y.\,a'_2$, 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 $(C_1, C_2) \in \mathcal{C}[\tau \mid a_1, a_2]_\gamma$ 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 $(C_1, C_2) \in \mathcal{C}[\tau \mid a_1, a_2]_\gamma$, $C_1[a_1] \precsim C_2[a_2]$.

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 $\emptyset \vdash C[\Gamma \vdash a_1 : \tau] : \mathsf{unit}$ and $\emptyset \vdash C[\Gamma \vdash a_2 : \tau] : \mathsf{unit}$, 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 $\emptyset \vdash C[\Gamma \vdash a_1 : \tau] : \mathsf{unit}$, 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 $\tau$ 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 $\gamma_1$ and $\gamma_2$.

*Definition 6.19 (Sides of an environment).* Consider an environment $\gamma$. 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 $\alpha$ 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 $\gamma$ 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 $\gamma$, 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 $(C_0, C_3) \in \mathcal{C}[\tau \mid a_0, a_3]_\gamma$. Then, we have $(C_0, C_0) \in \mathcal{C}[\tau \mid a_0, a_1]_{\epsilon_1\gamma}$, and $(C_3, C_3) \in \mathcal{C}[\tau \mid a_2, a_3]_{\epsilon_1\gamma}$, and $(C_0, C_3) \in \mathcal{C}[\tau \mid a_1, a_3]_{\epsilon_1\gamma}$. 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 $\Gamma$, an ML type $\tau$, 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 $\tau$ is said to be a* match tree *if the judgment* $\Gamma \vdash \tau$ tree *defined below holds:*

$$\frac{\Gamma \vdash \tau : \mathsf{Sch}}{\Gamma \vdash \tau \ \mathsf{tree}} \ \textit{Tree-Scheme}$$

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

*Suppose all types of variables in $\Gamma$ are match trees. If $\Gamma \vdash a : \tau$, there exists a type $\tau'$ 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 $\Gamma$, 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 $\tau$ to $\tau'$. If $\tau$ is equal to a match tree, then $\tau'$ is equal to a match tree by transitivity.
- For rule Let-Poly:

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

  By induction hypothesis, the type of $u$ is equal in $\Gamma$ to a match tree. Thus we can assume that $\tau$ is a match tree. Then, all types in $\Gamma, x : \tau, (x =_\tau u)$ are match trees. There exists $\tau''$ 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:

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

  By induction hypothesis, the type of $a$ is equal in $\Gamma$ to a match tree. Thus we can assume that $\tau$ is a match tree. Then, all types in $\Gamma, x : \tau, (x =_\tau a)$ are match trees. There exists $\tau''$ 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 $\tau$ has kind Typ, there is a value $v$ in $\tau$. Then, $\Gamma \vdash \tau''[x \leftarrow v]$ tree and $\Gamma \vdash \tau' \simeq \tau''[x \leftarrow v]$.
- For rule Match:

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

  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 $(\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, (u =_{\zeta\,(\tau_k)^k} d_i(\tau_{ij})^k(x_{ij})^j) \vdash \tau_i$ tree$)^i$. Then, consider $\tau' =$ match $u$ with $(d_i(\tau_{ij})^k(x_{ij})^j \to \tau_i)^i$. We have $\Gamma \vdash \tau' \simeq$ match $u$ with $(d_i(\tau_{ij})^k(x_{ij})^j \to \tau)^i$ (by applying the previous equality in each branch), and $\Gamma \vdash$ match $u$ 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 $\longmapsto_{\mathrm{t}}$ in Figure 24. 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 $\Gamma$ is a context where all variables have a ML type. Suppose $\Gamma \vdash a : \tau$, where $\tau$ 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. □

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 24 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.

Lemma 7.3 (ML conversions are trivial). *If $\Gamma \vdash \tau_1 \simeq \tau_2$ in eML where $\Gamma$ is equality-free and $\tau_1$ and $\tau_2$ are ML types, then $\tau_1 = \tau_2$.*

Proof. Apply an equivalent of $\longmapsto_{\mathrm{t}}$ and $\longmapsto$ to the complete derivation. Then, no useful equalities are introduced, and the derivation can be normalized by $\longrightarrow_\beta$. □

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathsf{let}\ (x : \mathsf{match}\ u\ \mathsf{with}\ (d_i\,\overline{\tau}\,(x_{ij})^j \longrightarrow \tau_i)^i) = a\ \mathsf{in}\ b \quad \longmapsto_{\mathrm{t}}} \\
\multicolumn{3}{r}{\mathsf{match}\ u\ \mathsf{with}\ (d_i\,\overline{\tau}\,(x_{ij})^j \longrightarrow \mathsf{let}\ (x : \tau_i) = a\ \mathsf{in}\ b)^i} \\
\mathsf{match}\ d_j\,\overline{\tau_j}\,(v_i)^i\ \mathsf{with}\ (d_j\,\overline{\tau_j}\,(x_{ji})^i \longrightarrow a_j)^j & \longmapsto & a_j[x_{ij} \leftarrow v_i]^i \\
\mathsf{let}\ x = u\ \mathsf{in}\ b & \longmapsto & b[x \leftarrow u] \\
\left(\begin{array}{r} \mathsf{match}\ (\mathsf{match}\ u\ \mathsf{with}\ (d_k\,\overline{\sigma}\,(x_{kj})^j \longrightarrow a_k)^k) \\ \mathsf{with}\ (d_i\,\overline{\tau}\,(x_{ij})^j \longrightarrow b_i)^i \end{array}\right) & \longmapsto & \\
\multicolumn{3}{r}{\left(\begin{array}{l} \mathsf{match}\ u\ \mathsf{with}\ (d_k\,\overline{\sigma}\,(x_{kj})^j \longrightarrow \\ \qquad\qquad (\mathsf{match}\ a_k\ \mathsf{with}\ (d_i\,\overline{\tau}\,(x_{ij})^j \longrightarrow b_i)^i)^k \end{array}\right)}
\end{array}
$$

Fig. 24. Match and let lifting

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 7.4 (Conversion elimination). *Consider an equality-free environment $\Gamma$. Assume $\Gamma \vdash a : \tau$ in eML and every pattern matching in $a$ is on a variable or an expansive term/ Then, there exists $a'$ such that $\emptyset \vdash a \simeq a'$ and $\emptyset \vdash a' : \tau$ in ML.*

Proof. The term is obtained by applying $\longmapsto_{\mathrm{t}}$ and $\longmapsto$. Then, the equalities introduced by pattern matchings are not used, and all equalities used in conversions can be proved in an equality-free environment (and then weakened). Conversions are not necessarily between ML types, but in that case, it is a conversion of the result of a match or let, and the conversion can be pushed inside. Then, all conversions are between ML types and can be eliminated. □

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 7.5 (Match elimination). *If $\Gamma \vdash a : \tau$ in eML where $\Gamma$ is an ML environment and $\tau$ is an ML type, then there exists an ML term $a'$ such that $\Gamma \vdash a \simeq a'$ and $\Gamma \vdash a' : \tau$.*

Proof. Apply the transformations $\longmapsto_{\mathrm{t}}$ 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 7.4. □

## 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 does not have any notion of ornament, and the interface presented to the user for ornamentation. We define both the datatype ornaments and the higher-order functional ornaments that can be built from them.

As a running example, we reuse the ornament natlist $\alpha$ from natural numbers to lists:

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

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

$$\begin{array}{rcl} \chi & ::= & \mathsf{natlist} \mid \ldots \\ \omega & ::= & \varphi \mid \chi\,(\omega)^i \mid \zeta\,(\omega)^i \mid \omega \to \omega \end{array} \qquad \begin{array}{rcl} \alpha^\epsilon & = & \alpha \\ (\omega_1 \to \omega_2)^\epsilon & = & \omega_1^\epsilon \to \omega_2^\epsilon \\ (\zeta\,(\omega_i)^i)^\epsilon & = & \zeta\,(\omega_i^\epsilon)^i \end{array} \qquad \frac{(\chi\,(\alpha_i)^i : \tau \Rightarrow \sigma)}{\begin{array}{rcl} (\chi\,(\omega_i)^i)^- & = & \tau[\alpha_i \leftarrow \omega_i^-]^i \\ (\chi\,(\omega_i)^i)^+ & = & \sigma[\alpha_i \leftarrow \omega_i^+]^i \end{array}}$$

$$(\alpha_i)^i \vdash \alpha_i \qquad \frac{(\alpha_i)^i \vdash \omega_1 \quad (\alpha_i)^i \vdash \omega_2}{(\alpha_i)^i \vdash \omega_1 \to \omega_2} \qquad \frac{\zeta : (\mathsf{Typ})^j \to \mathsf{Typ} \quad ((\alpha_i)^i \vdash \omega_j)^j}{(\alpha_i)^i \vdash \zeta\,(\omega_j)^j} \qquad \frac{\begin{array}{c} \chi\,(\alpha_j)^j : \ldots \Rightarrow \ldots \\ ((\alpha_i)^i \vdash \omega_j)^j \end{array}}{(\alpha_i)^i \vdash \chi\,(\omega_j)^j}$$

Fig. 25. Ornament types

A higher-order ornament natlist $\alpha \to$ natlist $\alpha$ relates two functions $f_-$ of type nat $\to$ nat and $f_+$ of type list $\tau \to$ list $\tau$ 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 term is a lifting of another if they are related at the desired ornament type.

The syntax of ornament types, given on Figure 25, mirrors the syntax of types. An ornament type, written $\omega$, may be an ornament variable $\varphi$, a datatype ornament $\chi\ \overline{\omega}$, 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 ($\omega_i$ indicates how the $i$-th type argument of the datatype is ornamented). An ornament type $\omega$ is interpreted as a relation between terms of type $\omega^-$ and $\omega^+$. The projection operation, defined on Figure 25, depends on the projections of the datatype ornaments: they are given by the global judgment $\chi\ \overline{\alpha} : \tau \Rightarrow \tau$.We also define a well-formedness judgment $(\alpha_i)^i \vdash \omega$ for ornaments given an environment of type variables. For example, the ornament list (natlist nat) describes the relation between lists whose elements have been ornamented using the ornament natlist nat. Thus, its projections are (list (natlist nat))$^-$ equal to list nat and (list (natlist nat))$^+$ equal to list (list nat).

The projection is defined and well-kinded for any well-formed ornament type:

Lemma 8.1 (Projection is a type). *If $\overline{\alpha} \vdash \omega$ holds, then we have $\overline{\alpha} \vdash (\omega)^\epsilon$ :* Typ.

Proof. By induction on the derivation of $\overline{\alpha} \vdash \omega$. □

We define in the next section how to interpret the base ornaments $\chi$, 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 $\tau_1$ and $\tau_2$ 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 use the function arrow and the ornament arrow interchangeably (when talking about the logical relation).

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 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 plays a key role in the instantiation: from a relation at a type, we deduce, by proving the correct instantiation, a relation at an ornament.

### 8.1 Defining Datatype Ornaments

A datatype $\zeta\ (\alpha_i)^i$ is defined by a family of constructors $(d_k)^k$ taking arguments of types $(\tau_{kj})^j$:

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

where the type $\zeta$ may occur recursively (possibly with some other types). We define the skeleton by *abstracting out* the concrete types from the constructors and replacing them by type parameters: the skeleton of $\zeta$, written $\hat{\zeta}$, is parametrized by types $(\alpha_{kj})^{kj}$ and has constructors:

$$(\hat{d}_\ell : \forall(\alpha_{kj} : \mathsf{Typ})^{kj}\ (\alpha_{\ell j})^j \to \hat{\zeta}\ (\alpha_{kj})^{kj})^\ell$$

Let us write $\mathcal{A}_\zeta(\tau_i)^i$ for $(\tau_{kj}[\alpha_i \leftarrow \tau_i]^i)^{kj}$, *i.e.* the function that expands arguments of the datatype into arguments of its skeleton. The types $\zeta\ (\tau_i)^i$ and $\hat{\zeta}\ (\mathcal{A}_\zeta(\tau_i)^i)$ are isomorphic by construction. Similarly to nat_skel in the overview, the skeleton allows us to incrementally ornament any subpart of a datatype before ornamenting the whole datatype (or, in the case of natlist, the recursive part).

Ornament definitions associate 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 either $P$ or $Q$, and the null pattern $\emptyset$ that matches nothing. 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 a mutually recursive group of definitions, each of the form:

$$\mathsf{type\ ornament}\ \chi\ (\alpha_j)^j : \zeta\ (\tau_k)^k \Rightarrow \sigma\ \mathsf{with}\ (P_i \Rightarrow Q_i\ \mathsf{when}\ (x_{i\ell} : \omega_{i\ell})^\ell)^i$$

with $\chi$ the name of the datatype ornament, $\zeta\ (\tau_k)^k$ the base type, and $\sigma$ the lifted type. The base and ornamented types must be such that $\zeta$ is a type constructor of arity $k$, $((\alpha_j : \mathsf{Typ})^j \vdash \tau_k : \mathsf{Typ})^k$ and $(\alpha_j : \mathsf{Typ})^j \vdash \sigma : \mathsf{Typ}$. Then, we can add $\chi\ (\alpha_j)^j : \zeta\ (\tau_k)^k \Rightarrow \sigma$ to the set of available ornaments. The ornaments of a recursive definition can be used in the body of this definition.

For each clause $i$ of the ornament, the patterns $P_i$ and $Q_i$ must each bind the same variables $(x_{i\ell})^\ell$, and the $(\omega_{i\ell})^\ell$ must be well-formed ornament types. In the user-facing syntax, we do not require an ornament signature for every variable: an identity ornament is inferred for the missing signatures. The patterns $(P_i)^i$ must be well-typed and form a partition of $\zeta\ (\tau_k)^k$, assuming $(x_{ij} : \omega^-_{ij})^j$. Moreover, they must consist only of variables and data constructors (thus they are also expressions – in particular, they do not contain constructs that erase information, *i.e.* alternative patterns and wildcards). The patterns $(Q_i)^i$ must form a well-typed partition of $\sigma$ assuming $(x_{ij} : \omega^+_{ij})^j$.

To be able to convert the ornament definitions to encoding and decoding functions, we introduce the *skeleton patterns* $(\hat{P}_i)^i$ obtained from $(P_i)^i$ by replacing the head constructor $d$ (of $\zeta$) by $\hat{d}$. If a pattern $P_i$ does not have a head constructor, the ornament definition is invalid. Assuming the pattern variables have types $(x_{i\ell} : \beta_{i\ell})^\ell$, the family of patterns $(\hat{P}_i)^i$ must form an exhaustive partition of some instance $\hat{\zeta}\ (\hat{\tau}_m)^m$ of the skeleton.

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 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 is used). We define a function $\llparenthesis \_ \rrparenthesis$ associating to a pattern this set of values.

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

Then, the interpretation is:

$$\mathcal{V}_p[\chi\,(\omega_j)^j]_\gamma = \bigcup_i \left\{ (P_i[x_{i\ell} \leftarrow v_{\ell-}]^\ell, v_+) \;\middle|\; \begin{array}{l} v_+ \in \llparenthesis Q_i[x_{i\ell} \leftarrow v_{\ell+}]^\ell \rrparenthesis \\ \forall \ell, (v_{\ell-}, v_{\ell+}) \in \mathcal{V}_p[\omega_{i\ell}[\alpha_j \leftarrow \omega_j]^j]_\gamma \end{array} \right\}$$

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 leave the type variables $(\alpha_j)^j$ free, so that they can be later instantiated. We write $\hat{\tau}_+$ for the type $\hat{\zeta}\,(\hat{\tau}_m[\beta_{i\ell} \leftarrow (\omega_{i\ell})^+]^{i\ell})^m$ of the skeleton where the recursive parts and the type parameters have already been lifted. The ornament is encoded as a quadruple $(\sigma, \delta, \mathsf{proj}, \mathsf{inj})$ where $\sigma : \mathsf{Typ}$ is the lifted type; $\delta$ is the *extension*, a type-level function describing the information that needs to be added; and proj and inj are the projection and injection functions introduced in §3. More precisely, the projection function proj from the lifted type to the skeleton has type $\Pi(x : \sigma).\ \hat{\tau}_+$ and, conversely, the injection inj 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* type function $\delta$ 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 pieces of information for different shapes of the skeleton, *e.g.* in the case of $\mathsf{natlist}\ \alpha$, we need no additional information when the skeleton is $\hat{\mathsf{Z}}$, but a value of type $\alpha$ when the skeleton starts with $\hat{\mathsf{S}}$, as explained at the end of §3.1. The encoding works incrementally: all functions manipulate the type $\hat{\tau}_+$, with all subterms already ornamented.

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 reconstruct 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, defined below[6]. 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 expect the missing information corresponding to each subpattern, given as a tuple. For wildcards, we expect a value of the type matched by the wildcard. Finally, for an alternative pattern, we require to choose between the two 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 $(\hat{P}_i)^i$ to determine which clause of the ornament definition can handle the given skeleton, and returns the corresponding extension type:

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

[6] Formally, we translate pattern typing derivations instead of patterns

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:

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

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

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

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:

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

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

$$\mathsf{type\ ornament}\ \zeta\,(\alpha_j)^j : \zeta\,(\alpha_j)^j \rightarrow \zeta\,(\alpha_j)^j\ \mathsf{with}\ (d_i(x_k)^k \rightarrow d_i(x_k)^k\ \mathsf{when}\ (x_k : \tau_{ik})^k)^i$$

## 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 rely on the fact that the functions describing the ornamentation from the base type $\tau_-$ to the ornamented type $\sigma_{\chi\,(\omega_j)^j}$ are related to the functions defining the identity ornament of $\tau_-$. Let us consider the relation on skeletons described by the ornament type $\hat{\omega} = \hat{\zeta}\,(\hat{\omega}_m)^m = \hat{\zeta}\,(\tau_m[\beta_{i\ell} \leftarrow \omega_{i\ell}]^{i\ell})^m$. This is the relation between a skeleton of the base type and a skeleton where the necessary subparts have been ornamented. Then, the projection function maps values related by the ornament to skeletons related by $\hat{\omega}$, and the injection maps related skeletons and any pair of patches to related values.

The relation on the skeleton is also important for lifting: it describes how we must lift the fields of a constructor of the base type. In the case of natlist $\alpha$, $\hat{\omega}$ is equal to $\widehat{\mathsf{nat}}$ (natlist $\alpha$): the field in $\hat{\mathsf{S}}$ must have already been ornamented with natlist $\alpha$ before we apply the injection.

The processed definitions are considered global and written $\chi\,(\alpha_j)^j \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\,(\hat{\omega}_m)^m : \zeta\,(\tau_i)^i \Rightarrow \sigma$. We require that all processed definitions are *valid*:

*Definition 8.2 (Valid ornament definition).* We say that $\chi\,(\alpha_j)^j \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\,(\hat{\omega}_m)^m : \zeta\,(\tau_i)^i \Rightarrow \sigma$ is a valid ornament definition for $\chi$ if:

- $(\alpha_j)^j \vdash \zeta\,(\tau_i)^i : \mathsf{Typ}$ and $(\alpha_j)^j \vdash \sigma : \mathsf{Typ}$;
- $\hat{\zeta}$ has arity $m$;
- $(\hat{\omega}_m^-)^m = \mathcal{A}_\zeta(\tau_i)^i$, which implies that the left projection of the skeleton is isomorphic to the base type;
- the types are correct:
  - $(\alpha_j)^j \vdash \sigma : \mathsf{Typ}$;

  - $(\alpha_j)^j \vdash \delta : \hat{\zeta}\,((\hat{\omega}_m)^+)^m \longrightarrow \mathsf{Typ}$;
  - $(\alpha_j)^j \vdash \mathsf{proj} : \Pi(x : \sigma).\ \hat{\zeta}\,((\hat{\omega}_m)^+)^m$;
  - $(\alpha_j)^j \vdash \mathsf{inj} : \Pi(x : \hat{\zeta}\,((\hat{\omega}_m)^+)^m).\ \Pi(y : \delta \sharp x).\ \sigma$
- for all $\gamma \in \mathcal{G}_k[(\alpha_j : \mathsf{Typ})^j]$, $\mathcal{V}[\chi\,(\alpha_j)^j]_\gamma$ is a relation between $\zeta\,(\gamma_1(\tau_i))^i$ and $\gamma_2(\sigma)$ and:
  - $\Big(\gamma_1(\mathsf{proj}_{\zeta\,(\tau_i)^i}), \gamma_2(\mathsf{proj})\Big) \in \mathcal{V}[\Pi(x : \chi\,(\alpha_j)^j).\ \hat{\zeta}\,(\omega_m)^m]_\gamma$;
  - $\Big(\gamma_1(\mathsf{inj}_{\zeta\,(\tau_i)^i}), \gamma_2(\mathsf{inj})\Big) \in \mathcal{V}[\Pi(x : \hat{\zeta}\,(\omega_m)^m)).\ \Pi(y : \delta \sharp x).\ \chi\,(\alpha_j)^j]_{\gamma[\delta \leftarrow \lambda\_.\ \mathsf{Top}]}$

Theorem 8.3. *The ornaments defined using the procedure described in this section are valid.*

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 points, 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 $\varphi$ and relate the terms obtained by instantiating it with the identity on the one hand and with another ornament on the other hand, 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) $\mathrm{id}_\zeta$ the identity ornament defined from the datatype $\zeta$.

Lemma 8.4 (Identity ornament). *For all $\gamma$, $\mathcal{V}_k[\mathrm{id}_\zeta\,(\omega_i)^i]_\gamma = \mathcal{V}_k[\zeta\,(\omega_i)^i]_\gamma$.*

Proof. Suppose the constructors of $\zeta$ are:

$$(d_k : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_{kj})^j \longrightarrow \zeta\,(\alpha_i)^i)^k$$

Expanding the logical relation given by the definition of the identity ornament on $\zeta$, we get:

$$\mathcal{V}_p[\chi\,(\omega_i)^i]_\gamma = \bigcup_k \left\{\ (d_k(v_{kj\_})^j, d_k(v_{kj_+})^j)\ \middle|\ \forall j, (v_{kj\_}, v_{kj_+}) \in \mathcal{V}_p[\tau_{kj}[\alpha_i \leftarrow \omega_i]^i]_\gamma \right\}$$

which is exactly the definition of $\mathcal{V}_p[\zeta\,(\omega_i)^i]_\gamma$.

□

## 9 ORNAMENTING TERMS

We now consider the problem of ornamenting terms. The ornamentation is done in two main steps: first the base term is elaborated to a generic term, which is then specialized using specific ornaments to generate ML code. The lifted code cannot be polymorphic in ornaments. To avoid the problem of considering parametric ornaments (ornaments depending on a type, but not in a computation-relevant way), we restrict ourselves to an input language with only top-level polymorphism. We also require that pattern matching be shallow, and the arguments of constructors be variables. The latter restriction can be met by compiling down deep pattern matching and explicitly binding the arguments of constructors to variables before passing these variables to constructors.

For the restriction to toplevel polymorphism, we need to make a distinction between (generalizable) toplevel bindings and monomorphic local bindings. The environment $\Gamma$ can then be split into $G, (\alpha_i : \mathsf{Typ})^i, \Delta$ where $G$ is an environment of polymorphic variable bindings, $(\alpha_i : \mathsf{Typ})^i$ the list of type variables parametrizing the current binding, and $\Delta$ a local environment binding only monomorphic variables. Additionally, we require that polymorphic variables are immediately instantiated when used in a term. This does not restrict the expressivity of the language: polymorphic local bindings can be duplicated (see §10.2 for a discussion of this point).

$$
\begin{array}{rcl}
\Gamma & ::= & G,\, \overline{\alpha}\,, S, R, \Delta \\
G & ::= & \emptyset \;\mid\; G, x\langle \overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A \\
\Delta & ::= & \emptyset \;\mid\; \Delta, x : \omega \;\mid\; \Delta, (a =_{\tau} a)^{\sharp}
\end{array}
\qquad
\begin{array}{rcl}
s & ::= & \emptyset \;\mid\; \varphi \mapsto \varphi \\
S & ::= & \emptyset \;\mid\; S, \varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\, \overline{\omega} : \zeta\, \overline{\tau} \Rightarrow \alpha \\
R & ::= & \emptyset \;\mid\; R, y :^{\sharp} \Gamma \to \delta_{\varphi}\ A \;\mid\; R, (x[\overline{\omega}, s] \rightsquigarrow y : \omega)
\end{array}
$$

Fig. 26. Environments

$$
\begin{array}{rcl}
\alpha^{\epsilon}_{S} & = & \alpha \\
(\omega_1 \to \omega_2)^{\epsilon}_{S} & = & (\omega_1)^{\epsilon}_{S} \to (\omega_2)^{\epsilon}_{S}
\end{array}
\qquad
\frac{(\varphi \mapsto \_ \lhd \_ : \tau \Rightarrow \sigma) \in S}{\varphi^{-}_{S} = \tau \qquad \varphi^{+}_{S} \;=\; \sigma}
\qquad
\begin{array}{rcl}
(R, x :^{\sharp} \sigma)^{+}_{S} & = & R^{+}_{S}, x : \sigma \\
(R, x[\_, \_] \rightsquigarrow y : \omega)^{+}_{S} & = & R^{+}_{S}, y : \omega^{+}_{S}
\end{array}
$$

$$
\begin{array}{rcl}
(\Delta, x : \omega)^{\epsilon}_{S} & = & \Delta^{\epsilon}_{S}, x : \omega^{\epsilon}_{S} \\
(\Delta, (a =_{\tau} b)^{\sharp})^{-}_{S} & = & \Delta^{-}_{S} \\
(\Delta, (a =_{\tau} b)^{\sharp})^{+}_{S} & = & \Delta^{+}_{S}, (a =_{\tau} b)
\end{array}
\qquad
\begin{array}{rcl}
(G, \overline{\alpha}, S, R, \Delta)^{-} & = & G^{-}, \overline{\alpha}, \Delta^{-}_{S} \\
(G, \overline{\alpha}, S, R, \Delta)^{+} & = & \overline{\alpha}, S^{+}, R^{+}_{S}, \Delta^{+}_{S} \\
(G, x\langle\, \overline{\alpha}\,, S, \_\rangle : \omega = \_ \rightsquigarrow \_)^{-} & = & G^{-}, x : \forall\, \overline{\alpha}\ \omega^{-}_{S}
\end{array}
$$

Fig. 27. Environment projections

To save notation, we just write $\alpha$ instead of $\alpha$ : Typ in typing contexts or polymorphic types, assuming that type variables have the Typ kind by default.

We now explain the ornamentation of a whole program, which is a sequence of toplevel definitions. For simplicity, we assume that type definitions and ornament definitions come first and are used to build the global environment of ornament definitions hereafter treated as a constant, followed by expression definitions, and last, by lifting definitions. Therefore, we may perform all elaborations first, followed by all specializations as requested by lifting definitions.

### 9.1 Elaborating to a Generic Program

Each toplevel definition "let $x = \Lambda\overline{\alpha}.\ a$" is elaborated in order of appearance, using the main elaboration judgment of the form $\Gamma \vdash a \rightsquigarrow A : \omega$ (described in Figure 29). The elaboration environment $\Gamma$ is actually of the form $G, \overline{\alpha}, S, R, \Delta$, as described in Figure 26. The local environment $\Delta$ is initially empty, as shown in Rule Elab-Decl (Figure 30) and used to bind variables appearing in $a$ to *ornament types*, as well as equalities $(a =_{\tau} a)^{\sharp}$ that may be needed to type the ornamented side. We use capital letter $A$ for elaborated terms to help distinguish them from base terms; $\omega$ is the ornament relating $a$ and $A$. $S$ and $R$ are explained below. The result of the elaboration of the definition is then folded into the *global* environment $G$ as a sequence of declarations of the form $x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A$ (rule Elab-Decl in Figure 30). The contexts $S$ and $R$ are new and used to describe abstract ornaments and patches, respectively.

The generic term $A$ is usually more polymorphic than $a$, since we abstract over ornaments where we originally had a fixed type. It is thus parametrized by a number of ornaments, described by the *ornament specification* environment $S$ which is a set of mutually recursive bindings, each of the form $\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\, (\omega_k)^{k} : \zeta\, (\tau_i)^{i} \Rightarrow \beta$. This binds an ornament variable $\varphi$ that can only be instantiated by a valid ornament (see Definition 8.2) of base type $\zeta\, (\tau_i)^{i}$ with skeleton $\hat{\zeta}\, (\omega_k)^{k}$; it also binds the target type $\beta$ and the ornament type extension, projection, and injection functions to the variables $\delta$, proj, and inj, respecting the types of valid ornaments.

An ornament type $\omega$ is well-formed in an environment $S$ with free type variables $\overline{\alpha}$, written $G, \overline{\alpha}, S \vdash \omega$ orn, if it contains only function arrows, type variables from $\overline{\alpha}$, and ornament variables bound in $S$ (see rules Wf-Orn-Arrow, Wf-Orn-TVar, and Wf-Orn-Var, in Figure 26).

A generic term also abstracts over patches and the liftings used to lift references to previously elaborated bindings. Since these bindings do not influence the final ornament type and are not

$$\textsc{G-Empty}\quad \vdash \emptyset$$

$$\textsc{G-Def}\quad \frac{\vdash G \qquad G, \overline{\alpha} \vdash S \qquad G, \overline{\alpha}, S \vdash \omega \text{ orn} \qquad (G, \overline{\alpha})^{-} \vdash a : \omega_S^{-} \qquad G, \overline{\alpha}, S \vdash R \qquad (G, \overline{\alpha}, S, R)^{+} \vdash A : \omega_S^{+}}{\vdash G, x\langle \overline{\alpha}, S, R \rangle : \omega = a \rightsquigarrow A}$$

$$\textsc{Wf-S}\quad \frac{\forall (\varphi \mapsto \_ \lhd \hat{\zeta}\, (\omega_k)^k : \zeta\, (\tau_j)^j \Rightarrow \_) \in S, (\overline{\alpha} \vdash \tau_j : \mathsf{Typ})^j \wedge (G, \overline{\alpha}, S \vdash \omega_k \text{ orn})^k \wedge ((\omega_k)_S^{-})^k = \mathcal{A}_{\zeta}(\tau_j)^j}{G, \overline{\alpha} \vdash S}$$

$$\textsc{Wf-R-Empty}\quad G, \overline{\alpha}, S \vdash \emptyset$$

$$\textsc{Wf-R-Patch}\quad \frac{G, \overline{\alpha}, S \vdash R \qquad (\varphi \mapsto (\delta, \ldots) \lhd \hat{\omega} : \_ \Rightarrow \_) \in S \qquad (G, \overline{\alpha}, S, R)^{+}, \Delta \vdash A : \hat{\omega}_S^{+}}{G, \overline{\alpha}, S \vdash R, y :^{\sharp} \Delta \longrightarrow \delta \sharp A}$$

$$\textsc{Wf-R-Inst}\quad \frac{G, \overline{\alpha}, S \vdash R \qquad (x\langle (\beta_j)^j, S', R \rangle : \omega) \in G \qquad (G, \overline{\alpha}, S \vdash \omega_j \text{ orn})^j \qquad G, \overline{\alpha}, S \vdash s : S'[\beta_j \leftarrow \omega_j]^j}{G, \overline{\alpha}, S \vdash R, (x[(\omega_j)^j, s] \rightsquigarrow y : \omega[(\beta_j \leftarrow \omega_j)^j, s])}$$

$$\textsc{Wf-inst}\quad \frac{\forall (\varphi \mapsto \_ \lhd \hat{\omega} : \tau \Rightarrow \_) \in S', (s(\varphi) \mapsto \_ \lhd s(\hat{\omega}) : \tau \Rightarrow \_) \in S}{G, \overline{\alpha}, S \vdash s : S'}$$

$$\textsc{Wf-Orn-Arrow}\quad \frac{G, \overline{\alpha}, S \vdash \omega_1 \text{ orn} \qquad G, \overline{\alpha}, S \vdash \omega_2 \text{ orn}}{G, \overline{\alpha}, S \vdash \omega_1 \longrightarrow \omega_2 \text{ orn}} \qquad \textsc{Wf-Orn-TVar}\quad \frac{\alpha \in \overline{\alpha}}{G, \overline{\alpha}, S \vdash \alpha \text{ orn}} \qquad \textsc{Wf-Orn-Var}\quad \frac{\varphi \in S}{G, \overline{\alpha}, S \vdash \varphi \text{ orn}}$$

Fig. 28. Well-formedness for elaboration

mutually recursive, they are stored in a separate *patch* environment $R$. Together, $S$ and $R$ specify all the parts that have to be user-provided at specialization time (see §9.2).

When encountering a variable $x$ corresponding to a global definition (Rule E-VarGlobal in Figure 29), we look up the signature of the elaboration of this definition $(x\langle \overline{\alpha}, S', R \rangle : \omega) \in G$. We choose an instantiation $\overline{\omega}$ of the type parameters $\overline{\alpha}$ by ornament types, an instantiation $s'$ of the ornament variables in $S'$ with ornament variables of $S$ (checked by the judgment $\Gamma \vdash s : S'[\overline{\alpha} \leftarrow \overline{\omega}]$, defined in the long version), and request a value $y$ corresponding to an instantiation of the function with the chosen type and ornament parameters (we do not instantiate the values in the $R'$, as they do not contribute to the lifting *specification*). We record this instantiation in the environment $R$ in the form $(x[\overline{\omega}, s] \rightsquigarrow y : \omega[\overline{\alpha} \leftarrow \overline{\omega}][s]) \in \Gamma$.

The environment $R$ also contains patches, *i.e.* *m*ML terms of the appropriate type, written in $R$ as $y :^{\sharp} \sigma$. Well-formedness rules require that the type $\sigma$ corresponds to meta-functions of multiple arguments returning a value of type $\delta \sharp A$ where $\delta$ is the extension function of some ornament in $S$.

The elaboration judgment $\Gamma \vdash a \rightsquigarrow A : \omega$ also contains the superposition of two typing judgments for the base term $a$ and lifted term $A$, as stated in Lemma 9.1. We use helper left and right projections to extract environments and types related to the base and lifted terms, respectively. These are defined in Figure 27. Most rules are unsurprising, once noticed that the projections of an ornament $\omega$ require an ornament specification $S$ in order to project ornament variables. For convenience, we may also use the superset $\Gamma$ instead of $S$ in the projection. The projection of a local environment $\Delta$ is the projection of its ornament types. The equalities are kept on the right-hand side and dropped

$$\frac{x : \omega \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \omega}\;\textsc{E-VarLocal} \qquad \frac{(x\langle \overline{\alpha}, S', R\rangle : \omega) \in \Gamma \quad \Gamma \vdash s : S'[\overline{\alpha} \leftarrow \overline{\omega}] \quad (\overline{\omega})^{-}_{S} = \overline{\tau} \quad (x[\overline{\omega}, s] \rightsquigarrow y : \omega[\overline{\alpha} \leftarrow \overline{\omega}][s]) \in \Gamma}{\Gamma \vdash x\ \overline{\tau} \rightsquigarrow y : \omega[\overline{\alpha} \leftarrow \overline{\omega}][s]}\;\textsc{E-VarGlobal}$$

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

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

$$\frac{\begin{array}{c}(\varphi \mapsto (\delta, \mathsf{inj}, \mathsf{proj}) \lhd \hat{\zeta}\,(\omega_i)^i : \zeta\,(\tau_\ell)^\ell \Rightarrow \_) \in \Gamma \quad \hat{d} : \forall(\alpha_i)^i\,(\omega_j)^j \to \hat{\zeta}\,(\alpha_i)^i \\ ((x_j : \omega_j[\alpha_i \leftarrow \omega_i]^i) \in \Gamma)^j \quad \Gamma = \_, \_, \_, \_, \Delta \quad (p :^{\sharp} \Delta^{+}_{S} \!\to \delta \sharp \hat{d}((\omega_i)^{+}_{S})^i(x_j)^j) \in \Gamma\end{array}}{\Gamma \vdash d(\tau_\ell)^\ell(x_j)^j \rightsquigarrow \mathsf{let}\ y = p \sharp \Delta^{+}_{S}\ \mathsf{in}\ \mathsf{inj} \sharp \hat{d}((\omega_i)^{+}_{S})^i(x_j)^j \sharp y : \varphi}\;\textsc{E-Con}$$

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

Fig. 29. Elaboration to a generalized term

$$\frac{G, \overline{\alpha}, S, R \vdash a \rightsquigarrow A : \omega}{G \vdash \mathsf{let}\ x = \Lambda\overline{\alpha}.\ a \Rightarrow G, (x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A)}\;\textsc{Elab-Decl}$$

Fig. 30. Elaborating a declaration

on the left-hand side. The right-hand side projection of ornament specifications is the *ordered* concatenation of two environments:

$$\begin{aligned} S^{+} \;=\; & \left\{\beta \mid (\varphi \mapsto \_ \lhd \_ : \_ \Rightarrow \beta) \in S\right\}, \\ & \left\{\delta : \hat{\omega}^{+}_{S} \to \mathsf{Typ},\ \mathsf{proj} : \Pi(x : \beta).\ \hat{\omega}^{+}_{S},\ \mathsf{inj} : \Pi(x : \hat{\omega}^{+}_{S}).\ \Pi(y : \delta \sharp x).\ \beta \right. \\ & \qquad\qquad\qquad\qquad \left. \mid (\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\omega} : \_ \Rightarrow \beta) \in S\right\} \end{aligned}$$

We use set notation for each environment as the internal order does not matter, but the respective order of the two sets does: the former binds the target types $\beta$ of ornaments, while the latter binds functions defining these ornaments. The order matters because, while $S$ is recursively defined, the environment $S^{+}$ is not. In the second part, we flatten the bindings $\delta, \mathsf{proj}, \mathsf{inj}$ whose types depend on the sequence of variables introduced first and make the typing constraints carried by $S$ explicit.

Given an ornament environment $S$, we can get the right projection $R^{+}_{S}$ of a patch environment $R$: it binds the patches and liftings requires by the elaborated term. Finally, the global environment of elaborated definitions $G$ projects on the left to a polymorphic environment $G^{+}$ and a substitution from global definitions to their non-elaborated values. We can chain these projections together to obtain a projection for $\Gamma$. On the left-hand side, we ignore $S$ and $R$ because they are not needed in the base term, while on the right-hand side we ignore $G$ because references to global definitions have been replaced with variables corresponding to liftings in $R$.

The well-formedness rules for all these constructions are given on Figure 28.

The main elaboration judgment $\Gamma \vdash a \rightsquigarrow A : \omega$, described in Figure 29, follows the structure of the original term. When used for type inference, we need to expand $\Gamma$ as $G, \overline{\alpha}, S, R, \Delta$ to see the flow of information: $G$, $\overline{\alpha}$, and $\Delta$ are inputs, while $S$ and $R$ are outputs, and added to on demand. The term $a$ is an input while $A$ and $\omega$ are outputs. Applications, abstractions, let-bindings, and local variables are translated to themselves. We have already explained the elaboration of global variables.

Pattern matching and construction of datatypes are the key rules. Reading Rule E-Match intuitively, $x$ is typed first, which determines the datatype ornament $\varphi$ and the type of the projection $\mathsf{proj}$ by looking up $\varphi$ in $S$. The type $\omega$ is given by elaborating the branches. In Rule E-Con, the type of $\hat{\zeta}\ (\omega_i)^i$ is first determined by the types of $\alpha_j$ and the type of the skeleton $\hat{d}$; then, an abstract ornament $\varphi$ is introduced in the $S$ subset of $\Gamma$ and a patch variable is introduced in the $R$ subset of $\Gamma$. The well-typedness comes again from the type constraints in $S$. Some ornament bindings in $S$ may in fact be forced to be equal.

As announced earlier, the elaboration judgments ensure well-typedness of the projections and the definition elaboration judgment preserves the well-typedness of $G$:

Lemma 9.1. *If $\Gamma \vdash a \rightsquigarrow A : \omega$ holds then both $\Gamma^- \vdash a : \omega^-_\Gamma$ and $\Gamma^+ \vdash A : \omega^+_\Gamma$ hold.*

Proof. By induction on the derivation. □

Lemma 9.2. *If $\vdash G$ and $G \vdash t \Rightarrow G'$ hold, then $\vdash G'$ holds.*

Proof. Expand the definitions, apply G-Def and use Lemma 9.1. □

In practice, the elaboration is obtained by inference. We first construct an elaborated term where all ornamentation records are different, and type it using the normal ML inference (this always succeeds because the term can be instantiated with records defining identity ornaments). Then, according to the constraints on elaboration environments, ornaments with the same lifted type must be the same. This is in fact sufficient: we only have to merge the ornaments whose lifted types are unified by ML inference. We thus obtain the most general generic program.

## 9.2 Specialization of the Generic Program

Specialization comes last, using the result $G$ of the elaboration and processing the sequence of user-given lifting declarations in order of appearance. We essentially describe the instantiation, since meta-reduction and simplification steps, described in section **??** and 7, can be done afterwards.

A lifting declaration of the form "$\mathsf{let}\ y\ \overline{\beta} = \mathsf{lifting}\ x\ (\omega_j)^j\ \mathsf{with}\ s, r$" defines $y$, polymorphic in the type variables $\overline{\beta}$, as a lifting of the base term $a$ bound by $x$ whose type parameters are instantiated by ornament types $(\omega_j)^j$. The user gives two substitutions $s$ and $r$: $s$ maps ornament variables (of some ornament specification $S$) to ornaments; $r$ maps term variables (of some patch specification $r$) to terms.

We use a judgment $\overline{\beta} \vdash s : S$ to state that the substitution $s$ conforms to an ornament specification $S$. This means that for every binding $(\varphi \mapsto \_ \lhd \hat{\omega}' : \_ \Rightarrow \_)$ in $S$, $s$ maps $\varphi$ to some ornament type $\chi\ (\omega_i)^i$ where $\chi$ is a concrete ornament such that $(\chi\ (\beta_i)^i \mapsto \_ \lhd \hat{\omega}'' : \_ \Rightarrow \_)$, the $(\omega_i)^i$ are well-formed ($(\overline{\alpha} \vdash \omega_i)^i$ holds), and $s(\hat{\omega}')$ is $\hat{\omega}''[\beta_i \leftarrow \omega_i]^i$. When $\overline{\beta} \vdash s : S$ holds, we may take the right-projection $s^+_S$ of $s$ that gives the code of the ornamentation functions, such that $\overline{\beta} \vdash s^+ : S^+$. That is for any $\varphi \leftarrow \chi\ (\omega_i)^i$ in $s$ corresponding to some $(\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \_ : \_ \Rightarrow \beta)$ in $S$, we put the bindings $\beta \leftarrow \sigma_{\chi\ (\omega_i)^i}, \delta \leftarrow \delta_{\chi\ (\omega_i)^i}, \mathsf{proj} \leftarrow \mathsf{proj}_{\chi\ (\omega_i)^i}, \mathsf{inj} \leftarrow \mathsf{inj}_{\chi\ (\omega_i)^i}$ in $s^+_S$.

$$I \ ::= \ \emptyset \mid I, \forall \overline{\alpha}\, (x[\overline{\omega}, s] \rightsquigarrow y\ \overline{\alpha} : \omega = A)$$

$$\emptyset^{+} = \emptyset \qquad\qquad (I, \forall \overline{\alpha}\, (x[(\omega_i)^i, s] \rightsquigarrow y\ \overline{\alpha} : \omega = A))^{+} = I^{+}, y : \forall \overline{\alpha}\, \omega^{+} = A$$

LiftEnv-Empty

$$G \vdash \emptyset$$

LiftEnv-Cons

$$\frac{(x\langle(\beta_j)^j, S, \_\rangle : \omega') \in G \qquad (\overline{\alpha} \vdash \omega_j)^j \qquad \begin{array}{c} G \vdash I \\ \overline{\alpha} \vdash s : S \end{array} \qquad I^{+}, \overline{\alpha} \vdash A : \omega^{+} \qquad \omega = s(\omega'[(\beta_j \leftarrow \omega_j)^j])}{G \vdash I, \forall \overline{\alpha}\, (x[(\omega_j)^j, s] \rightsquigarrow y\ \overline{\alpha} : \omega = A)}$$

Fig. 31. Lifting environment

Assume we have a *lifting environment* $I$ composed of bindings of the form $\forall \overline{\alpha}\, (x[\overline{\omega}, s] \rightsquigarrow y\ \overline{\alpha} : \omega = A)$ obtained from the previous liftings. The right projection of a binding $y : \forall \overline{\alpha}\, \omega^{+} = A$ gives the definition of the lifted term. We write $I^{+}$ for the projection of $I$(Figure **??**) which describes the definitions in scope in the lifted term.

We also use a judgment $I; \overline{\beta}; s \vdash r : R$ to check that the substitution $r$ is appropriate: this requires that the terms in $r$ are typed according to the specification $R$, namely $I^{+}, \overline{\beta} \vdash r : s(R_S^{+})$, using the projection $R_S^{+}$ defined in Figure 27. For any lifting $(x[\overline{\omega}, s] \rightsquigarrow y : \omega)$ in $R$, we require that $r(y) = z\ \overline{\tau}$ for some $z$, $\overline{\tau}$ and check that the lifting requirement matches the lifting signature of $z$ present in $I$.

To proceed with the instantiation, we first find the binding $(x\langle\, \overline{\alpha}\, , S, R\rangle : \omega = a \rightsquigarrow A)$ in $G$ of the variable $x$. We then construct a substitution $(\alpha_j \leftarrow \omega_j)^j$, say $\theta$, and check that $s$ and $r$ have types as prescribed by $S\theta$ and $R\theta$, that is, $\overline{\beta} \vdash s : S\theta$ and $I; \overline{\beta}; s \vdash r : R\theta$. Then, the instantiated term is $A[s_S^{+}, r, \theta^{+}]$, which we can meta-reduce and simplify into an ML term, say $B$. Finally, we build the lifting specification $\forall \overline{\beta}\, (x[(\omega_j)^j, s] \rightsquigarrow y\ \overline{\beta} : \omega[s_S^{+}, \theta] = B)$ which is added to the environment $I$ for subsequent liftings.

Lemma 9.3. *Suppose $\overline{\beta} \vdash s : S$. Then, $\overline{\beta} \vdash s_S^{+} : S^{+}$. Moreover, suppose $I; \overline{\beta}; s \vdash r : R$. Then, $(I, \overline{\beta})^{+} \vdash r : s(R_S^{+})$.*

Proof. For the first result, by unfolding the definitions. For the second result, by induction on the derivation. □

Lemma 9.4. *Suppose $G \vdash I$ and $G; I \vdash t \Rightarrow I'$. Then, $G \vdash I'$.*

Proof. Unfold the rule Lifting, then substitute (well-typed) instantiation in the typing judgment of generic term (well-typed by well-formedness of $G$). Apply Lift-Cons. □

These judgments check that the lifting is valid. In our prototype, they are also used to infer the ornaments and liftings that have not been specified by the user.

While ornaments are known to be well-typed, because they have been generated internally to the prototype, patches are given by the user and may contain type errors. We constrain the patches to be composed of a series of *m*ML abstractions and an *e*ML term. We can check the type of the *m*ML part, assuming the *e*ML part is well-typed. Then, *m*ML reduction does not diverge, and *e*ML simplification terminates independent of well-typedness (although it can signal type errors). We can then type the lifted term using ML inference: if it does not type, one of the patches was ill-typed.

$$\emptyset^+_S = \emptyset \qquad \frac{(\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \_ : \_ \Rightarrow \beta) \in S}{(s, \varphi \leftarrow \chi\ (\omega_i)^i)^+_S = s^+_S[\beta \leftarrow \sigma_{\chi\ (\omega_i)^i}, \delta \leftarrow \delta_{\chi\ (\omega_i)^i}, \mathsf{proj} \leftarrow \mathsf{proj}_{\chi\ (\omega_i)^i}, \mathsf{inj} \leftarrow \mathsf{inj}_{\chi\ (\omega_i)^i}]}$$

Wf-s

$$\frac{\forall(\varphi \mapsto \_ \lhd \hat{\omega} : \_ \Rightarrow \_) \in S,\ \exists(\chi\ (\beta_i)^i \mapsto \_ \lhd \hat{\omega}' : \_ \Rightarrow \_), (\overline{\alpha} \vdash \omega_i)^i,\ s(\varphi) = \chi\ (\omega_i)^i \wedge s(\hat{\omega}) = \hat{\omega}'[\beta_i \leftarrow \omega_i]^i}{\overline{\alpha} \vdash s : S}$$

Wf-r-Empty

$$I; \overline{\alpha}; s \vdash \emptyset : \emptyset$$

Wf-r-Patch

$$\frac{I; \overline{\alpha}; s \vdash r : R \qquad I^+, \overline{\alpha} \vdash A : \sigma[(s)^+, (r)^+]}{I; \overline{\alpha}; s \vdash r, y \leftarrow A : R, y :^\sharp \sigma}$$

Wf-r-Lifting

$$\frac{\begin{array}{ccc} I; \overline{\alpha}; s \vdash r : R & \forall(\beta_i)^i\ (x[(\omega'_j)^j, s'] \rightsquigarrow z\ (\beta_i)^i : \omega') \in I & ((\omega_i)^+ = \sigma_i)^i \\ (s(\omega''_j) = \omega'_j[\beta_i \leftarrow \omega_i]^i)^j & (s(s'') = s'[\beta_i \leftarrow \omega_i]^i)^j & s(\omega'') = \omega'[\beta_i \leftarrow \omega_i]^i \end{array}}{I; \overline{\alpha}; s \vdash r, y \leftarrow z\ (\sigma_i)^i : R, (x[(\omega''_j)^j, s''] \rightsquigarrow y : \omega'')}$$

Lifting

$$\frac{(x\langle(\alpha_j)^j, S, R\rangle : \omega = a \rightsquigarrow A) \in G \qquad \theta = (\alpha_j \leftarrow \omega_j)^j \qquad \overline{\beta} \vdash s : S\theta \qquad I; \overline{\beta}; s \vdash r : R\theta}{G; I \vdash \mathsf{let}\ y\ \overline{\beta} = \mathsf{lifting}\ x\ (\omega_j)^j\ \mathsf{with}\ s, r \Rightarrow I, \forall\overline{\beta}\ (x[(\omega_j)^j, s] \rightsquigarrow y\ \overline{\beta} : \omega\theta \mathsf{simplify}(A[s^+_S, r, \theta^+]))}$$

Fig. 32. Projection and checkingof ornament environment, lifting

## 9.3 Correctness of the Lifting

We use the logical relation from §5.2 to prove that the lifted term is related to the base term by ornamentation. We first focus on the elaboration: we introduce an *identity instantiation* and prove that, for all elaborated terms, the identity instantiation gives back the original term.

*Definition 9.5.* Given environments $S$ and $R$, the *identity instantiation* $\mathsf{id}_{S,R}$ is defined as the composition of $s^+$ and $r$ where:

- $s$ are identity ornaments: for all $(\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \hat{\zeta}\ (\omega_i)^i : \zeta\ (\tau_\ell)^\ell \Rightarrow \beta)$ in $S$, the substitution $s^+$ maps $\beta$, $\delta$, $\mathsf{proj}$, and $\mathsf{inj}$ to $\zeta\ (\tau_\ell)^\ell$, $\delta_{\zeta\ (\tau_\ell)^\ell}$, $\mathsf{proj}_{\zeta\ (\tau_\ell)^\ell}$, $\mathsf{inj}_{\zeta\ (\tau_\ell)^\ell}$, respectively.
- patches are trivial, *i.e.* for all $y :^\sharp \Delta \to \delta \sharp A$ in $R$, the substitution $r$ maps $y$ to $\lambda^\sharp\Delta.\ ()$.
- for all $(x[\overline{\omega}, s] \rightsquigarrow y : \omega')$ in $R$, the substitution $r$ maps $y$ to $x\ (\overline{\omega})^-_S$.

Lemma 9.6. *If* $\vdash G, \overline{\alpha}, S, R$, *then* $\mathsf{id}_{S,R}$ *exists and* $(G, \overline{\alpha})^- \vdash \mathsf{id}_{S,R} : (S, R)^+$.

Proof. By induction on the well-formedness judgment of $S$ and $R$. We also prove that, for any ornament in scope, the extension $\delta$ is $\lambda\_.\ \mathsf{unit}$. Thus, the patches are well-typed. For the liftings, notice that each ornament is the identity ornament: thus, we ask for a function of the same type as the original function. □

We say that an elaboration $(x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A)$ in $G$ is *appropriate* if the identity instantiation of the generic term gives back the original term: $(G, \overline{\alpha})^- \vdash a \simeq \mathsf{id}_{S,R}(A)$. An environment $G$ is appropriate if it contains only appropriate definitions.

Theorem 9.7. *Suppose* $G, \overline{\alpha}, S, R \vdash a \rightsquigarrow A : \omega$. *Then* $(x\langle\overline{\alpha}, S, R\rangle : \omega = a \rightsquigarrow A)$ *is appropriate. As a consequence, elaborating a declaration preserves the property that the environment is appropriate.*

Proof. By induction on the derivation, substituting and meta-reducing the definitions of the identity ornament. It is then necessary to split on the values to simplify the construction and immediate destruction of the skeleton in the match case. □

We now prove that the liftings we generate are indeed related to the base term by the ornamentation relation: we say that a lifting $\forall\overline{\beta}\ (x[(\omega_j)^j, s] \rightsquigarrow y\ \overline{\beta} : \omega = A)$ in $G$ is *appropriate* if the base term $a$ (typed in $G^-$) and the lifted term $A$ (typed in $I^+$) are related at $\omega$ for all choices of the type variables. Formally, we use the logical relation (§5.2): for all $\gamma \in \mathcal{G}[\overline{\beta}]$, we want

$$\big((G^- \circ \gamma_1)(a), (I^+ \circ \gamma_2)(A)\big) \in \mathcal{V}[\omega]_\gamma$$

A lifting environment is appropriate if it contains only appropriate liftings. The property we need to prove is that, in an appropriate $G$, the lifting environment $I$ stays appropriate when processing a new lifting. It suffices to show that the generated lifting is appropriate.

Consider $\gamma \in \mathcal{G}[\overline{\beta}]$. We prove the correctness of the ornamentation by constructing a relational instantiation $\gamma' \in \mathcal{G}[(\overline{\beta}, S, R)^+]$ as follows. For $\beta \in \overline{\beta}$, take $\gamma'(\beta) = \gamma(\beta)$. For ornaments $(\varphi \mapsto (\delta, \mathsf{proj}, \mathsf{inj}) \lhd \_ : \tau \Rightarrow \alpha) \in S$, take $\gamma'(\alpha) = \mathcal{V}[s(\varphi)]_\gamma$, $\gamma'(\delta) = \lambda\_.\ \mathrm{Top}$, $\gamma'(\mathsf{proj}) = (\mathsf{proj}_\tau, \mathsf{proj}_{s(\varphi)})$ and $\gamma'(\mathsf{inj}) = (\mathsf{inj}_\tau, \mathsf{inj}_{s(\varphi)})$ as in Definition 8.2. For patches $y :^\sharp \Delta \longrightarrow \delta \sharp A$, take $\gamma'(y) = (\lambda\Delta.(), r(y))$. For liftings $y$ of $x\ \overline{\tau}$, take $\gamma'(y) = ((G^- \circ \gamma_1)(x\ \overline{\tau}), (I^+ \circ \gamma_2)(r(y)))$.

Lemma 9.8. *Consider $\gamma \in \mathcal{G}[\overline{\alpha}]$, and suppose $G, S, R, I$ are well-formed. Then, $\gamma' \in \mathcal{G}[(\overline{\alpha}, S, R)^+]$, $\gamma'_1 = G^- \circ \gamma_1 \circ \mathsf{id}_{S,R}$, and $\gamma'_2 = I^+ \circ \gamma_2 \circ s^+_S \circ r$.*

Proof. For ornaments, use the validity of ornaments in the environment. For liftings, use well-formedness of $I$. For patches, use the fact that the relation on $\delta$ is constant equal to Top, and the unit patch terminates. □

Then, we can instantiate the generic term with $\gamma'$. On the left-hand side we obtain a term equivalent to the base term, and on the right-hand side a term equivalent to the lifted term (because simplification preserves equivalence). Both terms are related by the relation $\mathcal{V}[s(\omega)]_\gamma$. Thus we deduce correctness of the lifting process:

Theorem 9.9 (Correctness of lifting). *Suppose $I$ is appropriate, and consider a lifting request $s$. If $G; I \vdash s \Rightarrow I'$, then $I'$ is appropriate.*

Proof. Let us consider the rule Lifting. First, $(s^+_S \circ r)(A)$ is well-typed in an equality-free environment, and thus simplifies to an ML term, preserving equality (and thus the logical relation). Consider $\gamma \in \mathcal{G}_k[\overline{\beta}]$. We have to prove:

$$((G^- \circ \gamma_1)(a), (I^+ \circ \gamma_2 \circ s^+_S \circ r)(A)) \in \mathcal{V}[s(\omega)]_\gamma$$

Consider $\gamma'$ defined as above. We can rewrite $a$ to $\mathsf{id}_{S,R}(A)$ by the identity instantiation property of $G$. Thus, $(G^- \circ \gamma_1)(a) = \gamma'_1(a)$. By definition, $(I^+ \circ \gamma_2 \circ s^+_S \circ r)(A) = \gamma'_2(a)$. Suppose $S$ gives the type variables $\alpha_i$ as lifted types of the ornament variables $\varphi_i$. Then,

$$\begin{array}{rcl}
\mathcal{V}[s(\omega)]_\gamma & = & \mathcal{V}[\omega[\varphi_i \leftarrow s(\varphi_i)]^i]_\gamma \\
 & = & \mathcal{V}[\omega]_{\gamma[\varphi_i \leftarrow \mathcal{V}[s(\varphi_i)]_\gamma]^i} \\
 & = & \mathcal{V}[\omega^+_S]_{\gamma[\alpha_i \leftarrow \mathcal{V}[s(\varphi_i)]_\gamma]^i} \\
 & = & \mathcal{V}[\omega^+_S]_{\gamma'}
\end{array}$$

Thus, we have to prove $(\gamma'_1(A), \gamma'_2(A)) \in \mathcal{V}[\omega^+_S]_\rho$. This is true by the fundamental lemma, because $\gamma' \in \mathcal{G}_k[(\overline{\beta}, S, R)^+]$ and $(\overline{\beta}, S, R)^+ \vdash A : \omega^+_S$. □

In the case of strictly positive datatypes and first-order functions, this result can be translated to the *coherence* property of ornaments [Dagand and McBride 2014]. We can define a *projection function* that projects the whole datatype at once (rather than incrementally as with the proj function), *e.g.* the length function for natlist $\alpha$. Then, the relation expressed by natlist $\alpha$ between $a_1$ and $a_2$ is simply $a_1 = \mathsf{length}\ a_2$ (up to termination). The statement that append is a lifting of add at natlist $\alpha \to$ natlist $\alpha \to$ natlist $\alpha$ can then be translated (again, up to termination) to the fact that, for all $a_1, a_2$, length (append $a_1$ $a_2$) = add (length $a_1$) (length $a_2$).

### 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.

## 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, stored in an elaboration environment and then instantiated and simplified in a separate phase. From our experience, this principled approach is more modular and robust than an attempt to go directly from base terms to ornamented terms. Of course, our prototype also performs inference during elaboration, while we have only presented elaboration as a checking relation. Inference is a rather orthogonal issue and does not raise any difficulty.

The prototype is a proof of concept that only accepts programs in a toy language. Porting the implementation to a real language, such as OCaml, would allow to demonstrate the benefits of ornamentation on real, large cases. We believe that instances of pure refactoring would already be very useful to the programmer, even though it is just a small subset of the possibilities.

As presented, elaboration abstracts over all possible ornamentation points, which requires to specify many identity ornaments and corresponding trivial patches, while many datatypes may never be ornamented. For example, refactoring a library may need a specific ornament for one type and the identity ornament for all others. We already allow wildcard on occurrences to apply the identity ornaments by default. We could also use global rules, to avoid repeating local rules. We could also avoid generating the ornamentation points in the generic lifting that are known in advance to be always instantiated to the identity ornament. This information could be user-specified, or be inferred by scanning all ornament definitions prior to elaboration.

The lifting process, as described, only operates on ML terms restricted to shallow pattern matching and where constructors are only applied to variables. To meet these restrictions we preprocess terms, turning deep pattern matching into shallow pattern matching, and lifting the arguments of constructors into separate let bindings. This does not preserve the structure of the original program and creates unnatural looking terms as output. To recover a term closer to the original one, we mark the bindings we introduced and substitute them back afterwards. When applying this transformation, we keep the evaluation order of the arguments even if they are permuted. Thus our implementation should preserve effects and their ordering. We use a similar strategy for deep pattern matching. During compilation to shallow pattern matching, we annotate the generated matches with tags that are maintained during the elaboration and, whenever possible, we merge back pattern matchings with identical tags after elaboration. This seems to work well, and a primitive treatment of deep pattern matching does not seem necessary andould be more involved, so we currently do not feel the need for such an extension.

Pattern matching clauses with wildcards may be expanded to multiple clauses with different head constructors. For the moment we only factor them back in obvious cases, but we could use tags to try to merge all clauses in the lifted code that originate from the same clause in the base code.

These transformation phases introduce auxiliary variables. Some of these bindings will eventually be expanded, but some will remain in the lifted program. Before printing, we select names derived from the names used in the original program. This seems to be enough to generate readable terms.

## 10.2 Polymorphic Let Bindings

Currently, in a pre-elaboration pass, all local polymorphic let-bindings are duplicated into a sequence of monomorphic let for each usage point. This does not reduce the expressivity of the system —assuming, as [Vytiniotis et al. 2010], that they are sufficiently rare (see to avoid exponential behavior. This transformation requires the user to instantiate what appears to be the same code multiple times. On the other hand, it is useful because it allows lifting a local definition differently for different usage points.

The duplicated local definitions could be tagged and shared back after ornamentation if their instantiations are identical. 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 $\lambda$-lifting and extruding the definition to the toplevel: we can then use our mechanism for lifting references to global definition and fold the definition back in the term before printing it out. It would also be possible to allow a local definition to be ornamented with *polymorphic ornaments*, *i.e.* ornaments polymorphic on a type parameter. This would not solve the problem of using different ornaments for different usage points, but would allow polymorphic recursion—which is not allowed in our current presentation.

## 10.3 Lifting

For convenience, we do not require that all parameters be instantiated when lifting a term: we infer some ornaments and liftings and automatically fill-in patches that return unit. We also provide a way to specify a default ornament for a type. These simple strategies seem to work well for small examples, but it remains to see if they also scale to larger examples with numerous ornamentation points. Our view is that inferring patches 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 [Chambard and Henry 2012; Devriese and Piessens 2011; Scala 2017; White et al. 2014], 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, as illustrated in §2.3.

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 realistic scenarios, programs are written in a modular way. We could generalize and then instantiate whole modules, and store the resulting environments in an ornamentationnterface file describing the relation between a base module and its lifting. Modular ornamentation could be applied to libraries: when releasing a new interface-incompatible version of a library, a maintainer could distribute an ornamentation specification allowing clients of the library to automatically migrate their code, leaving holes only at the points requiring user input.

### 10.4 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 liftings seem to follow this pattern. Syntactic lifting seems to be rather predictable and intuitive and leads to quite natural results. This restriction 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.

We could also consider preprocessing source programs prior to ornamentation. Indeed, $\eta$-expansion or unfolding of recursive definitions could provide more opportunities for ornamentation (*e.g.* in §??). In the cases we observed, the unfolding follows the structure of an ornament. Hence, it would be interesting to perform the unfolding on demand during the instantiation process.

The correctness result we give for lifting only gives weak guarantees with respect to termination: since the logical relation relates a non-terminating term on the right-hand side to any term on the left-hand side, a diverging term is an ornamentation of any term. Using the inverse relation, can prove a stronger property: if the patches always terminate, the lifting terminates exactly when the base term terminates.However, we would like an even more precise result: if a lifting does not terminate while the base term terminates, it can only be because of looping inside the code given by a patch.

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. 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 for the effects performed in patches. Similarly, the complexity of the ornamented program should be proportional to the complexity of the original one, save for the time spent in patches.

### 10.5 Future Work

Programming with generalized algebraic datatypes (GADT) requires writing multiple definitions of the same type holding different invariants. GADT definitions that only add *constraints* could be considered ornaments of regular types, which was one of the main motivations for introducing ornaments in the first place [Dagand and McBride 2014]. It would then be useful to automatically derive, whenever possible, copies of the functions on the original type that preserve the new invariants. Extending our results to the case of GADTs is certainly useful but still challenging future work. Besides issues with type inference, GADTs also make the analysis of dead branches more difficult. 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.

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 for other program transformations or for 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, one might consider exposing *m*ML to the user to let her write generic patches that could be instantiated as needed at many program points.

Ornaments can be composed in multiple ways. Applying the effects of two ornaments consecutively (lifting one type to another one, and lifting the lifted type again to a third type) can be done by re-lifting an already lifted definition. An interesting direction is to combine the information added by two ornaments into a datatype representing the base type and both ornamentations [Dagand and McBride 2013; Ko and Gibbons 2013]. Lifting could then similarly be composed to liftings along the combined ornament. This could be especially useful with GADTs: one could build a new datatype by combining two invariants established independently and obtain liftings by combining two independent liftings. Finally, it would be interesting to consider other transformations of datatypes, and in particular, *deornamentation*: instead of adding information to existing datatypes, deornamentation removes information from a datatype and adapts the functions operating on the original datatype so that they can operate on its impoverished version.

## 11 RELATED WORK

Ornaments have been recently introduced by [Dagand and McBride 2013, 2014; McBride 2011] in the context of dependently typed languages, where they can be encoded instead of treated as primitive. In this context, Ko and Gibbons [2016] describe a different way to handle higher-order ornaments without using logical relations. We first considered applying ornaments to an ML-like language in [Williams et al. 2014].

Type-Theory in Color [Bernardy and Guilhem 2013] is another way to understand the link between a base type and a richer type. 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, which would correspond to the base term. 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

Programming with GADTs may require defining one base structure and several structures with some additional invariants, along with new functions for each invariant. Ghostbuster [McDonell et al. 2016] proposes a *gradual* approach to porting functions to GADTs with richer invariants, by allowing as a temporary measure to write a function against the base structure and dynamically checking that it respects the invariant of the richer structure, until the appropriate function is written.

Najd and Peyton-Jones [2016] also observe that one often needs many variants of a given data structure (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.

Our *m*ML language is equipped with rudimentary meta-programming facilities, by separating a meta-abstraction from the ML abstraction. Its main distinguishing features from usual approaches to meta-programming in ML (such as Kiselyov [2014]) is its ability to embed *e*ML and transform equalities, allowing simplification of partial pattern matchings.

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 [Wadler 1986] and later reformulated by Okasaki [Okasaki 1998] 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 [Foster et al. 2007] 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 seem 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?

Our principled approach with a posteriori abstraction of the source term revealed very beneficial for ornaments and we imagine that it could also be used for other forms of program transformations beyond ornaments that remain to be explored.

## REFERENCES

Jean-Philippe Bernardy and Moulin Guilhem. 2013. Type-theory in color. In *International Conference on Functional Programming*. 61–72. https://doi.org/10.1145/2500365.2500577

Pierre Chambard and Grégoire Henry. 2012. Experiments in generic programming: runtime type representation and implicit values. Presentation at the OCaml Users and Developers meeting, Copenhagen, Denmark. (sep 2012). http://oud.ocaml.org/2012/slides/oud2012-paper4-slides.pdf

Pierre-Évariste Dagand and Conor McBride. 2013. 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*. IEEE Computer Society, 530–539. https://doi.org/10.1109/LICS.2013.60

Pierre-Évariste Dagand and Conor McBride. 2014. Transporting functions across ornaments. *J. Funct. Program.* 24, 2-3 (2014), 316–383. https://doi.org/10.1017/S0956796814000069

Dominique Devriese and Frank Piessens. 2011. 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)*. 143–155. https://doi.org/10.1145/2034773.2034796

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems* 29, 3 (May 2007), 17. https://doi.org/citation.cfm?doid=1232420.1232424

Oleg Kiselyov. 2014. *The Design and Implementation of BER MetaOCaml*. Springer International Publishing, Cham, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

Hsiang-Shang Ko. 2014. *Analysis and synthesis of inductive families*. DPhil dissertation. University of Oxford.

Hsiang-Shang Ko and Jeremy Gibbons. 2013. Modularising inductive families. *Progress in Informatics* 10 (2013). https://doi.org/doi:10.2201/NiiPi.2013.10.5

Hsiang-Shang Ko and Jeremy Gibbons. 2016. Programming with ornaments. *Journal of Functional Programming* 27 (2016). https://doi.org/10.1017/S0956796816000307

Conor McBride. 2011. Ornamental Algebras, Algebraic Ornaments. (2011). https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf

Trevor L. McDonell, Timothy A. K. Zakian, Matteo Cimini, and Ryan R. Newton. 2016. Ghostbuster: A Tool for Simplifying and Converting GADTs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 338–350. https://doi.org/10.1145/2951913.2951914

Shayan Najd and Simon Peyton-Jones. 2016. Trees that grow. *JUCS* (2016). https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/trees-that-grow-2.pdf

Chris Okasaki. 1998. Views for Standard ML. In *In SIGPLAN Workshop on ML*. 14–23.

Scala. 2017. Implicit Parameters. Scala documentation. (2017). https://docs.scala-lang.org/tour/implicit-parameters.html

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let Should Not Be Generalised, In Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation. https://www.microsoft.com/en-us/research/publication/let-should-not-be-generalised/

Philip Wadler. 1986. Views: A way for pattern matching to cohabit with data abstraction. (1986).

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*. 22–63. https://doi.org/10.4204/EPTCS.198.2

Thomas Williams, Pierre-Évariste Dagand, and Didier Rémy. 2014. Ornaments in practice. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, José Pedro Magalhães and Tiark Rompf (Eds.). ACM, 15–24. https://doi.org/10.1145/2633628.2633631

Thomas Williams and Didier Rémy. 2017. *A Principled Approach to Ornamentation in ML*. Research Report. Inria. https://hal.inria.fr/hal-01628060