

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, Dependent Types, Logical Relations.

ACM Reference Format:
Thomas Williams and Didier Rémy. 2018. A Principled Approach to Ornamentation in ML. *Proc. ACM Program. Lang.* 2, POPL, Article 21 (January 2018), 30 pages. https://doi.org/10.1145/3158109

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 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 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.

Authors' addresses: Thomas Williams, Inria, Paris, France; Didier Rémy, Inria, Paris, France.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/1-ART21
https://doi.org/10.1145/3158109

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 7.6); 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. In §6, we give a formal definition of ornaments based on a logical relation. In §7, 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 §8 and related work in §9.

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

[1] The prototype, available at url http://pauillac.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/

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).

```
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 relation is recursively defined. The first clause relates Const i to Const' i for any integer i. The second clause should be understood as:

$$\mathsf{Add}(u_-,v_-) \Rightarrow \mathsf{Binop'}(\mathsf{Add'}(u_+, v_+)) \textbf{ when } u_- \Rightarrow u_+ \textbf{ and } v_- \Rightarrow v_+ \textbf{ in } \mathsf{oexpr}$$

and means that Mul(u_-, v_-) and Binop'(Mul'(u_+, v_+)) are in the oexpr relation whenever both u_- and u_+ on the one hand and v_- and v_+ on the other hand are 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 a part that is inferred). 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'.

Lifting also works with higher-order types and recursive datatype definitions with negative occurrences (see the library of examples included with the prototype).

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.

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 or as a lifting of and, and the compiler inverts the constructors; it may also do so selectively, only at some given occurrences of the bool type, while carefully rejecting inconsistencies.

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.

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** $Cons(x_0,_) \to x_0$" 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.

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. For compactness, we omit the case of tuples (it is included in the extended version).

```
let bind x f = match x with Some v → f v | None → None
let rec eval env e = function
  | Abs(x, f) → Some (Abs(x, f))
  | App(e1,e2) → bind (eval env e1) (function
      | Abs(x,f) → bind (eval env e2) (fun v → eval (Cons((x,v), env)) f)
      | Tup _ → None              (* Type error *)
      | _ → fail ())              (* Note a value *)
  | Var x → assoc x env | ...
(* 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. Lifting constructors of excluded cases will fail, and pattern matching on excluded cases them will be eliminated. 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.

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 prototype, 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:

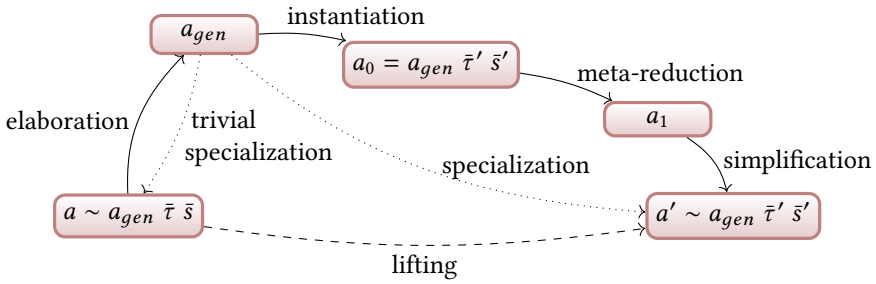


Fig. 1. Overview of the lifting process

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

2.4 More Examples

More examples can be found in [Williams and Rémy 2017] or the online prototype: we can enrich a data structure (turning sets into associative maps), use code refactoring for optimization by changing the data representation (transforming a map with unit as values into a set) or for hiding administrative data such as location information or type annotations in abstract syntax trees, *etc.*

Although it was not our initial goal, we found that ornaments could also be used for generic programming: one can define a type of arbitrary sum of products, then view first-order datatypes as a specialization, represented by an ornament, of this structure. Thus, functions operating on the general structure could be lifted to any datatype. However, this is not practical yet, as the generic functions must be manually unfolded to fit the recursive structure of the ornament, defeating the purpose of defining generic functions—we discuss the problem of unfolding in §8.4.

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. 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 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 §5.3).

The lifting process is formally described in section §7. In the rest of this section, we present it informally on the example of add and append.

3.1 Encoding Ornaments

We are trying to build a function append that has the same structure as add, and operates on constructors Nil and Cons similarly to the way add proceeds with constructors S and Z . Since ornamentation only affects datatypes, it is enough to insert some code to translate from and to lists at occurrences where nat is either constructed or destructed in add.

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

In the second branch, we must construct a list out of the hybrid list-nat skeleton S' (append₁ m' n). We use a helper function to inject an 'a list nat_skel into an 'a list :

```
  | S' m' → inj_nat_list₁ (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 append₂ m n = match proj_nat_list # m with
  | Z' → n
  | S' m' → inj_nat_list # (S'(append₂ m' n)) # (match m with Cons (x, _) → x)
```

This version is correct, but not final yet, as it still contains the intermediate hybrid structure, which will eventually be eliminated. Besides, for more complex examples, we cannot give a valid ML type to the analog of inj_nat_list, as the argument x takes different types in different branches. 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.

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 append₂, which removes all helper functions:

```
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, append₃ 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 append₃, 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.

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. 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 append₂ 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 append₂
```

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 p₁ #⇒
  let rec add_gen' m n = match m_orn.proj # m with
    | Z' → n
    | S' m' → n_orn.inj # S'(add_gen' m' n) # (p₁ # 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 §7), 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 p₁ #⇒
  let rec add_gen' m n = match m_orn.proj # m with
    | Z' → n
    | S' m' → n_orn.inj # S'(add_gen' m' n) # (p₁ # 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 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 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 append_5 gives append_2 which can then be simplified to append.

```
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
```

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

Fig. 2. Syntax of ML

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 §7.

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 are available. In such situations, lifting information must also be provided as additional rules. See the full version for such examples. 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 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 5.5). Then, in an environment without equalities, they can be simplified into ML terms (§5.3). For space reasons, we omit the definitions that are not essential for understanding the elaboration. We refer the reader to Williams and Rémy [2017] for a complete presentation.

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; and just $\overline{Q}$ if we do not have to explicitly mention the components; $\overline{Q}$ stands for $(Q, .. Q)$ in syntax definitions. We write $Q[z_i \leftarrow Q_i]^i$ for the simultaneous substitution of z_i by Q_i in Q for all i in I.

4.1 ML

We consider an explicitly typed version of ML. In practice, the user writes programs with implicit types that are elaborated into the explicit language, but we leave out type inference here for sake of simplicity[3]. The programmer's language is core ML with recursion and datatypes. Its syntax is described in Figure 2, ignoring the gray which is not part of the ML definition. To prepare for

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

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

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

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

$$\textsc{Conv}\ \frac{\Gamma \vdash \tau_1 \simeq \tau_2 \quad \Gamma \vdash a : \tau_1}{\Gamma \vdash a : \tau_2} \qquad \textsc{Con}\ \frac{d : \forall(\alpha_j : \mathsf{Typ})^j\, (\tau_i)^i \to \tau \quad (\Gamma \vdash \tau_j : \mathsf{Typ})^j \quad (\Gamma \vdash a_i : \tau_i[\alpha_j \leftarrow \tau_j]^j)^i}{\Gamma \vdash d(\tau_j)^j (a_i)^i : \tau[\alpha_j \leftarrow \tau_j]^j}$$

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

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

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

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

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

Fig. 4. Reduction rules of ML

extensions, we slightly depart from traditional presentations and introduce monotypes of kind Typ as a restriction of type schemes of super kind Sch. Still, type schemes are not first-class, since polymorphic type variables range only over monomorphic types, *i.e.* those of kind Typ.

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

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

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 $\mathsf{fix}\,(f : \tau_1 \to \tau_2)\, x.\, a$.

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 4. This reduction is written $\longrightarrow_\beta$.

Typing environments Γ contain term variables $x : \tau$ and type variables α : Typ. The typing rules are standard and given in Figure 3. Typing judgments are of the form $\Gamma \vdash a : \tau$ where $\Gamma \vdash \tau$: Sch. Although we do not have references, we still have a form of value restriction: Rule Let-Poly restricts polymorphic binding to *non-expansive terms* u, defined in Figure 2, that do not contain application—and whose reduction always terminate. This will be important once we add equalities. Binding of an expansive term is still allowed, 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. 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. 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:

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

We revisit the rules Let-Mono, Let-Poly, and Match-eML of Figure 3, now reading the gray: a let binding introduces an equality in the typing context witnessing that the new variable is equal to its definition (rules Let-Mono and Let-Poly); similarly, both type-level and term-level pattern matching introduce equalities witnessing the branch under selection (Match). 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 $\Gamma \vdash \tau_1 \simeq \tau_2$.

Although the equality judgment plays a key role, its formal definition is technical and only given in [Williams and Rémy 2017], as the rest of the presentation does not depend on it. It is defined generically on terms, types, and kinds: even if equality of terms does not appear in typing derivations, terms do appear in types. Equality assumptions between non-expansive terms are injected into the equality. A controlled form of reduction is also allowed into the equality: type-level and term-level pattern matching, let binding, and type abstraction and application can be reduced to check for equality. This is enough to reduce all closed non-expansive terms to values.

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 *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 5. 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 ♯ 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 and applications.

The restriction of meta-applications to the non-expansive subset of terms is to ensure that non-expansive terms are closed under meta-reduction. It is important that a non-expansive term remains non-expansive after substitution. Therefore, we may only allow substitution by non-expansive terms.

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

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

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

$$
\textsc{App-Meta}\ \frac{\Gamma \vdash a : \Pi(x : \tau_1).\, \tau_2 \quad \Gamma \vdash u : \tau_1}{\Gamma \vdash a \sharp u : \tau_2[x \leftarrow u]}
\qquad
\textsc{EApp}\ \frac{\Gamma \vdash a_1 \simeq a_2 \quad \Gamma \vdash b : \Pi(\diamond : a_1 =_{\tau'} a_2).\, \tau}{\Gamma \vdash b \sharp \diamond : \tau}
$$

$$
\textsc{EAbs}\ \frac{\Gamma \vdash \tau : \mathsf{Sch} \quad \Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau \quad \Gamma, (a_1 =_\tau a_2) \vdash b : \tau'}{\Gamma \vdash \lambda^\sharp(\diamond : a_1 =_\tau a_2).\, b : \Pi(\diamond : a_1 =_\tau a_2).\, \tau'}
$$

Fig. 5. Syntax, reduction and typing of *m*ML

In particular, arguments of redexes in Figure 5 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. We add meta-abstractions, but not meta-applications, to the class of non-expansive terms u. The meta-reduction, written $\longrightarrow_\sharp$, is defined in Figure 5. It is a strong reduction, allowed under arbitrary contexts C.

The new typing rules of *m*ML are also given in Figure 5. Meta-abstractions are dependently typed: the value of the argument can be used in the type of the result. The types $\forall^\sharp(\alpha : \kappa).\, \tau$, $\Pi(x : \tau).\, \tau$, and $\Pi(\diamond : a =_\tau a).\, \tau$ are classified in a superkind Met of Sch. This prevents them from appearing anywhere but at the toplevel of a Met-kinded type, which is essential for ensuring that *e*ML-typed *m*ML expressions can be reduced to *e*ML. Type abstractions $\Lambda^\sharp(\alpha : \kappa).\, \tau$ and type applications $\tau \sharp \tau$ are classified using dependent function kinds $\forall(\alpha : \kappa)\, \kappa$. Term abstractions $\lambda^\sharp(x : \tau).\, \tau$ and applications in types $\tau \sharp a$ are classified using term-to-type kind functions $\tau \to \kappa$.

5 THE METATHEORY OF *M*ML

In this section we present the main results on the metatheory of *m*ML that we later use to prove the correctness of the encoding of ornaments. More results and detailed proofs can be found in Williams and Rémy [2017].

We write $\longrightarrow$ for the union of $\longrightarrow_\beta$, and $\longrightarrow_\sharp$, and $\longrightarrow^*$ for its transitive closure. We also write $\longrightarrow_0$ for $\longrightarrow$ without reduction of ML applications and $\longrightarrow_0^*$ for its transitive closure. This reduction terminates on well-typed terms. Thanks to the careful design of meta-reduction and the grammar of non-expansive terms, any combination of the reduction relations $\longrightarrow_0$, $\longrightarrow_\beta$, $\longrightarrow_\sharp$ is confluent. Below we show that meta-reduction can always be performed first—hence at ornamentation time.

5.1 Type Soundness

Using a reducibility argument, we prove the following result:

Theorem 5.1 (Normalization for meta-reduction). *The reduction $\longrightarrow_\sharp$ is strongly normalizing.*

$$
\begin{array}{lcl}
\mathcal{G}[\emptyset] & = & \{\emptyset\} \\
\mathcal{G}[\Gamma, x : \tau] & = & \{\gamma[x \leftarrow (u_1, u_2)] \mid (u_1, u_2) \in \mathcal{E}[\tau]_\gamma \wedge \gamma \in \mathcal{G}[\Gamma]\} \\
\mathcal{G}[\Gamma, \alpha : \kappa] & = & \{\gamma[\alpha \leftarrow (\tau_1, R, \tau_2)] \mid (\tau_1, R, \tau_2) \in \mathcal{K}[\kappa]_{\gamma_1,\gamma_2} \wedge \gamma \in \mathcal{G}[\Gamma]\} \\
\mathcal{G}[\Gamma, (a_1 =_\tau a_2)] & = & \{\gamma \in \mathcal{G}[\Gamma] \mid (\vdash \gamma_1(a_1) \simeq \gamma_1(a_2)) \wedge (\vdash \gamma_2(a_1) \simeq \gamma_2(a_2))\} \\
\mathcal{E}[\tau]_\gamma & = & \{(a_1, a_2) \mid \forall v_2,\ (a_2 \longrightarrow^* v_2) \implies \exists v_1,\ (a_1 \longrightarrow^* v_1) \wedge (v_1, v_2) \in \mathcal{V}[\tau]_\gamma\} \\
\mathcal{V}[\alpha]_\gamma & = & \gamma(\alpha) \\
\mathcal{V}[\tau_1 \to \tau_2]_\gamma & = & \{(v_1, v_2) \mid (v_i = \mathsf{fix}\,(y : \tau')\,x.\,a_i)^i \\
 & & \qquad \wedge\ \ \forall (v_1', v_2') \in \mathcal{V}[\tau_1]_\gamma,\ \left(a_i'[y \leftarrow v_i, x \leftarrow v_i']\right)^i \in \mathcal{E}[\tau_2]_\gamma\} \\
\mathcal{V}[\zeta\ (\tau_i)^i]_\gamma & = & \{(d(v_j)^j, d(w_j)^j) \mid (d : \forall(\alpha_i : \mathsf{Typ})^i\ (\tau_j)^j \to \zeta\ (\alpha_i)^i) \\
 & & \qquad \wedge\ \ \forall\,(j)\ (v_j, w_j) \in \mathcal{V}[\tau_j[\alpha_i \leftarrow \tau_i]^i]_\gamma\} \\
\mathcal{V}\left[\begin{array}{l}\mathsf{match}\ a\ \mathsf{with} \\ (d_i(x_{ij})^j \to \tau_i)^i\end{array}\right]_\gamma & = & \begin{cases} \mathcal{V}[\tau_j]_{\gamma[x_{ij} \leftarrow (v_j, v_j')]^j} & \text{if } \gamma_1(a) \longrightarrow_0^* d_i(v_j)^j \wedge \gamma_2(a) \longrightarrow_0^* d_i(v_j')^j \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 6. Definition of the logical relation (excerpt)

We have the usual subject reduction and progress properties. We only state them for the reduction of terms (while they are actually recursively defined with the reduction of types and kinds).

Lemma 5.2 (Subject reduction). *If $\Gamma \vdash a : \tau$ in mML and $a \longrightarrow a'$, then $\Gamma \vdash a' : \tau$ in mML.*

We have a progress property for $\longrightarrow_\sharp$ alone and for *e*ML terms:

Theorem 5.3 (Soundness for *e*ML). *Well-typed closed eML terms are values or reduce.*

Theorem 5.4 (Reduction of *m*ML terms). *Consider an mML typing judgment $\Gamma \vdash a : \tau$ where Γ is an eML typing environment and τ an eML type. Then, a has a meta-normal form a' which, moreover, is an eML term.*

Moreover, the resulting *e*ML term has an *e*ML typing derivation:

Theorem 5.5 (*m*ML does not type new *e*ML terms). *If the judgment $\Gamma \vdash a : \tau$ is provable in mML, and Γ, a, and τ are eML environment, term, and type, then it also has an eML derivation.*

This is a key for the elimination of meta-abstractions and meta-applications after instantiation of generic liftings.

5.2 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 define a deterministic reduction $\mapsto$ that interleaves the ordinary and meta-reductions while still guaranteeing reduction of closed terms to values. We prove that terms normalize for $\mapsto$ if and only if they normalize for $\longrightarrow$. We also define an analogue $\mapsto_0$ of $\longrightarrow_0$. However, in this simplified presentation, we keep the full reduction $\longrightarrow$, and ignore the details of the step-indexing[4] An excerpt of the definition is given in Figure 6. We again refer the reader to Williams and Rémy [2017] for the exact definition and technical details.

The logical relation gives an interpretation of types. It depends on an environment γ that associates type variables to triples composed of two types and a relation between them (for parametric polymorphism), and term variables to pairs of related terms (for dependent types).

[4]Step-indexing is needed to cope with type definitions with recursive occurrences in negative positions.

Typing environments $\mathcal{G}[\Gamma]$ are interpreted as sets of environments. Equalities are interpreted as restricting environments to those where two closed terms are equal for the equality of mML in the empty context. Kinds $\mathcal{K}[\kappa]_{\gamma_1,\gamma_2}$ are interpreted as sets of relations for kinds of types, and as functions for higher order kinds (this interpretation is given in the long version). Types τ are interpreted in an environment γ as a relation on terms $\mathcal{E}[\tau]_\gamma$. This interpretation is done through a relation on values, written $\mathcal{V}[\tau]_\gamma$: two terms a_1, a_2 are related if they evaluate to related values v_1, v_2, or if a_2 does not terminate. Notice the asymmetry of $\mathcal{E}[\tau]_\gamma$, which allows the right-hand-side term to diverge while the left-hand-side term converges. This allows comparing programs for termination. To relate only programs that converge exactly as often, we may instead consider the intersection of $\mathcal{E}[\tau]_\gamma$ with its symmetric relation $\mathcal{E}[\tau]_\gamma^{-1}$.

The interpretation of function types $\tau_1 \to \tau_2$ is standard: two functions (v_1, v_2) are related if, for a pair of arguments (w_1, w_2) related at type τ_1, the terms $(v_1\ w_1, v_2\ w_2)$ are related at type τ_2. Similarly, two terms (a_1, a_2) of a given datatype are related if they evaluate to terms starting with the same constructor, and the fields of the constructor are related at their respective types. Type-level pattern matching is interpreted by evaluating the argument in the environment to determine the correct branch, and then evaluating the type in the branch in an environment extended with the variables bound by the pattern. We use the reduction $\longrightarrow_0$ because reduction of applications is forbidden in equalities, and because $\longrightarrow_0$ terminates. The interpretations of the other type-level computations (not included in this version) are defined so as to be compatible with type-level evaluation: type-level abstraction is interpreted as a function from interpretations to interpretations and type-level application as application of the interpretation of the function to the interpretation of the argument.

The logical relation has the expected properties:

Theorem 5.6 (Fundamental lemma). *If* $\Gamma \vdash a : \tau$ *and* $\gamma \in \mathcal{G}[\Gamma]$, *then* $(\gamma_1(a), \gamma_2(a)) \in \mathcal{E}[\tau]_\gamma$.

Theorem 5.7 (Equality). *If* $\Gamma \vdash a \simeq a'$, $\Gamma \vdash b \simeq b'$, *and* $\gamma \in \mathcal{G}[\Gamma]$, *then* $(\gamma_1(a), \gamma_2(b)) \in \mathcal{E}[\tau]_\gamma$ *if and only if* $(\gamma_1(a'), \gamma_2(b')) \in \mathcal{E}[\tau]_\gamma$.

5.3 Simplification from *e*ML to ML

The terms obtained from meta-reducing a meta-closed mML term are usually not typeable in ML, as the types appearing in them may contain type-level pattern matching and the typing derivation may use equalities. However, we are able to transform eML terms typeable in an environment without equalities into programs typeable in ML.

The transformation proceeds in two steps. In a first step, we ensure that all the equalities that are used are of the form $x = d(y_i)^i$. For simplicity, we start by reducing all let-bindings of non-expansive terms. Since equalities on expansive terms cannot be used for substitution, we only have to consider equalities introduced by pattern matching on non-expansive terms. The only non-expansive terms that can appear in a well-typed pattern matching are variables, type constructors, and pattern matching (since let bindings have been reduced). Pattern matching on a type constructor reduces. For the case of nested pattern matching, we extrude the inner pattern matchings by repeatedly applying the following transformation:

$$\begin{pmatrix}\mathsf{match}\ (\mathsf{match}\ u\ \mathsf{with}\ (d_k\ \overline{\sigma}\ (x_{kj})^j \to a_k)^k)\\ \mathsf{with}\ (d_i\ \overline{\tau}\ (x_{ij})^j \to b_i)^i\end{pmatrix} \longmapsto \begin{pmatrix}\mathsf{match}\ u\ \mathsf{with}\ (d_k\ \overline{\sigma}\ (x_{kj})^j \to\\ \qquad(\mathsf{match}\ a_k\ \mathsf{with}\ (d_i\ \overline{\tau}\ (x_{ij})^j \to b_i)^i)^k\end{pmatrix}$$

In a second step, we get rid of all the implicit uses of equalities in conversions. All equalities between non-expansive terms are of the form $x = d(y_i)^i$ and introduced in the branch of a pattern matching. Then, in this branch, we can replace y by $d(y_i)^i$. If a substitution occurs in the argument of a pattern

matching, we reduce the pattern matching. Formally, we apply the following transformation:

$$\left(\text{match } x \text{ with } (d_k\ \overline{\sigma}\ (x_{kj})^j \to a_k)^k\right) \longmapsto \left(\text{match } x \text{ with } (d_k\ \overline{\sigma}\ (x_{kj})^j \to a_k[x \leftarrow d_k\ \overline{\sigma}\ (x_{kj})^j])^k\right)$$

Thus, all uses of these equalities in conversions become uses of reflexivity, and conversions are provable in a context without equalities. Then, the conversions are not necessary:

Lemma 5.8 (ML conversions are trivial). *If $\Gamma \vdash \tau_1 \simeq \tau_2$ in eML where Γ is equality-free and τ_1 and τ_2 are ML types, then $\tau_1 = \tau_2$.*

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

6 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 α from natural numbers to lists:

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

The ornament natlist α defines, for all types α, a relation between values of its *base type* nat, which we write $(\text{natlist } \alpha)^-$, and its *lifted type* list α, written $(\text{natlist } \alpha)^+$: 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.

A higher-order ornament natlist $\alpha \longrightarrow$ natlist α relates two functions f_- of type nat $\longrightarrow$ nat and f_+ of type list $\tau \longrightarrow$ list τ when for related inputs v_- and v_+, the outputs $f_-\ v_-$ and $f_+\ v_+$ are related.

6.1 Ornamentation as a Logical Relation

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 7, mirrors the syntax of types. An ornament type, written ω, may be an ornament variable φ, a datatype ornament $\chi\ \overline{\omega}$, a higher-order ornament $\omega_1 \longrightarrow \omega_2$, or an *identity* ornament $\zeta\ (\omega)^i$, which is automatically defined for any datatype of the same name (ω_i indicates how the i-th type argument of the datatype is ornamented). An ornament type ω is interpreted as a relation between terms of type ω^- and ω^+. The projection operation, defined on Figure 7, depends on the projections of the datatype ornaments: they are given by the global judgment $\chi\ \overline{\alpha} : \tau \Rightarrow \tau$. 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 $(\text{list (natlist nat)})^-$ equal to list nat and $(\text{list (natlist nat)})^+$ equal to list (list nat).

Interestingly, our tentative interpretation of a functional ornament type $\omega_1 \longrightarrow \omega_2$ as a pair of functions (f_1, f_2) taking related arguments (v_1, v_2) at ornament type ω_1 to related results $(f_1\ v_1, f_2\ v_2)$ at ornament type ω_2 corresponds exactly to the interpretation of function types in the logical relation, with types replaced by ornament types. Likewise, interpretation of the identity ornament $\zeta\ (\omega_i)^i$ is similar to the interpretation of the datatype $\zeta\ (\tau_i)^i$, with the type parameters replaced by

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

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

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

Fig. 7. Ornament types

ornaments. Thus, we identify the interpretations of function types and functional ornaments, and the interpretation of datatype with the interpretation of the identity ornament of this datatype. From the point of view of the logical relation, ornament types are a mere extension of the syntax of types with non-identity datatype ornaments. An immediate consequence is that a well-typed term is in an ornamentation relation with itself at its type, seen as an ornament type (*i.e.* as the identity ornament corresponding to its type). These properties are keys to the lifting correctness proof.

6.2 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 ζ 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 ζ, 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 χ the name of the datatype ornament, $\zeta\,(\tau_k)^k$ the base type, and σ the lifted type. The base and ornamented types must be such that ζ 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 (they do not contain alternative patterns, wildcards, and the empty pattern, thus they are also expressions). The patterns $(Q_i)^i$ must form a well-typed partition of σ 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 ζ) 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, and the wildcards on the right-hand side can be replaced by any value of the correct type. As a notation shortcut, we insert patterns into the relation instead of the set of values matched by this pattern. Then, the interpretation is:

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

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

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

6.3 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 σ : Typ is the lifted type; δ 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 δ of kind $\hat{\tau}_+ \to$ 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 natlist α, we need no additional information when the skeleton is $\hat{\mathsf{Z}}$, but a value of type α when the skeleton starts with $\hat{\mathsf{S}}$, as explained at the end of §3.1. The encoding works 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[5]. 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

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

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 $\tau_1 + \tau_2$.

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

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:

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

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:

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

The 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 δ for $\delta_{\chi\,(\omega_j)^j}$):

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

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

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

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

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

6.4 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 τ_- to the ornamented type $\sigma_{\chi\,(\omega_j)^j}$ are related to the functions defining the identity ornament of τ_-. 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 α, $\hat{\omega}$ is equal to $\widehat{\mathsf{nat}}$ (natlist α): the field in $\hat{\mathsf{S}}$ must have already been ornamented with natlist α 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 6.1 (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 χ if:

- the ornamentation functions have the correct types defined above;
- $(\hat{\omega}_m^-)^m = \mathcal{A}_\zeta(\tau_i)^i$, which implies that the left projection of the skeleton is isomorphic to the base type;
- for all $\gamma \in \mathcal{G}[(\alpha_j : \mathsf{Typ})^j]$, $\mathcal{V}[\![\chi\,(\alpha_j)^j]\!]_\gamma$ is a relation between $\zeta\,(\gamma_1(\tau_i))^i$ and $\gamma_2(\sigma)$ and:
  - $\Big(\gamma_1(\mathsf{proj}_{\zeta\,(\tau_i)^i}), \gamma_2(\mathsf{proj})\Big) \in \mathcal{V}[\![\Pi(x : \chi\,(\alpha_j)^j).\ \hat{\zeta}\,(\omega_m)^m]\!]_\gamma$;
  - $\Big(\gamma_1(\mathsf{inj}_{\zeta\,(\tau_i)^i}), \gamma_2(\mathsf{inj})\Big) \in \mathcal{V}[\![\Pi(x : \hat{\zeta}\,(\omega_m)^m)).\ \Pi(y : \delta\,\sharp\,x).\ \chi\,(\alpha_j)^j]\!]_{\gamma[\delta \leftarrow \lambda_.\ \mathsf{Top}]}$

The ornaments defined in this section are valid. Together, these properties allow us to take a term that uses the encoding of a yet-unspecified ornament φ and relate the terms obtained by instantiating 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.

7 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 Γ 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 Δ 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 §8.2 for a discussion of this point).

To save notation, we just write α instead of $\alpha : \mathsf{Typ}$ 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.

7.1 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 10). The elaboration environment Γ is actually of the form $G, \overline{\alpha}, S, R, \Delta$, as described in Figure 8. The local environment Δ is initially empty, as shown in Rule Elab-Decl (Figure 11) 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

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

Fig. 8. Environments

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

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

Fig. 9. Environment projections

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

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

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

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

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

Fig. 10. Elaboration to a generalized term

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

Fig. 11. Elaborating a declaration

side. We use capital letter A for elaborated terms to help distinguish them from base terms; ω 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 11). 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 φ that can only be instantiated by a valid ornament (see Definition 6.1) of base type $\zeta\,(\tau_i)^i$ with skeleton $\hat{\zeta}\,(\omega_k)^k$; it also binds the target type β and the ornament type extension, projection, and injection functions to the variables δ, proj, and inj, respecting the types of valid ornaments.

An ornament type ω 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.

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 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 §7.2).

When encountering a variable x corresponding to a global definition (Rule E-VarGlobal in Figure 10), 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 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 σ corresponds to meta-functions of multiple arguments returning a value of type $\delta \sharp A$ where δ 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 7.1. 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 9. Most rules are unsurprising, once noticed that the projections of an ornament ω require an ornament specification S in order to project ornament variables. For convenience, we may also use the superset Γ instead of S in the projection. The right-hand side projection of ornament specifications is the *ordered* concatenation of two environments:

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

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 β 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 δ, proj, inj whose types depend on the sequence of variables introduced first and make the typing constraints carried by S explicit.

Well-formedness judgments have been omitted by lack of space—see Williams and Rémy [2017] for details.

The main elaboration judgment $\Gamma \vdash a \rightsquigarrow A : \omega$, described in Figure 10, follows the structure of the original term. When used for type inference, we need to expand Γ as $G, \overline{\alpha}, S, R, \Delta$ to see the flow of information: G, $\overline{\alpha}$, and Δ are inputs, while S and R are outputs, and added to on demand. The term a is an input while A and ω 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 φ and the type of the projection proj by looking up φ in S. The type ω 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 α_j and the type of the skeleton $\hat{d}$; then, an abstract ornament φ is introduced in the S subset of Γ and a patch variable is introduced in the R subset of Γ. 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:

Lemma 7.1. *If $\Gamma \vdash a \rightsquigarrow A : \omega$ holds then both $\Gamma^- \vdash a : \omega_\Gamma^-$ and $\Gamma^+ \vdash A : \omega_\Gamma^+$ hold.*

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.

7.2 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 5.1 and 5.3, can be done afterwards.

A lifting declaration of the form "let $y\ \overline{\beta}$ = lifting $x\ (\omega_j)^j$ 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 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 φ to some ornament type $\chi\ (\omega_i)^i$ where χ is a concrete ornament such that $(\chi\ (\beta_i)^i \mapsto _ \lhd \hat{\omega}'' : _ \Rightarrow _)$, the $(\omega_i)^i$ are well-formed, 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^+.

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, 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 9. 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 θ, and check that $\overline{\beta} \vdash s : S\theta$ 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.

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.

7.3 Correctness of the Lifting

We use the logical relation from §5.1 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 7.2. 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 β, δ, proj, and 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 7.3. *If* $\vdash G, \overline{\alpha}, S, R$, *then* $\mathsf{id}_{S,R}$ *exists and* $(G, \overline{\alpha})^- \vdash \mathsf{id}_{S,R} : (S, R)^+$.

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 7.4. *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.*

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 ω for all choices of the type variables. Formally, we use the logical relation (§5.1): 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 $\gamma'(\alpha) = \mathcal{V}[s(\varphi)]_\gamma$, $\gamma'(\delta) = \lambda_$. 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 6.1. For patches $y :^\sharp \Delta \to \delta\ \sharp\ A$, 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 7.5. *Consider $\gamma \in \mathcal{G}[\overline{\alpha}]$, and suppose G, S, R, I are well-formed. Then, $\gamma' \in \mathcal{G}[(\overline{\alpha}, S, R)^+]$, $\gamma'_1 = G^- \circ \gamma_1 \circ \mathrm{id}_{S,R}$, and $\gamma'_2 = I^+ \circ \gamma_2 \circ s_S^+ \circ r$.*

Then, we can instantiate the generic term with γ'. 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 7.6 (Correctness of lifting). *Suppose I is appropriate, and consider a lifting request s. If $G; I \vdash s \Rightarrow I'$, then I' is appropriate.*

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 α. Then, the relation expressed by natlist α 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 α 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).

8 DISCUSSION

8.1 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. 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. We could 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 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 would be more involved.

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.

8.2 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 λ-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.

8.3 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], as illustrated in §2.3.

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 interface file describing the relation between a base module and its lifting. Then, 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.

8.4 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, η-expansion or unfolding of recursive definitions could provide more opportunities for ornamentation

(*e.g.* in §2.4). 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. We can prove a stronger property: if the patches always terminate, the lifting terminates exactly when the base term terminates.

We have described ornaments in 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.

8.5 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.

The meta-language *m*ML 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.

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, 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.

9 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. Terms operating on a colored type can then be erased to terms operating on the uncolored version, which would correspond to the base term.

Ghostbuster [McDonell et al. 2016] proposes a *gradual* approach to porting functions to GADTs with richer invariants, by allowing to write a function against the base structure and dynamically checking that it respects the invariant of the richer structure.

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 an idiom to encode extensible datatypes (including GADTs) using existing Haskell features. This approach requires that the programmer adds in extension points from the start, instead of allowing unforeseen extensions. The encoding is not eliminated, leaving runtime and readability costs.

Views, first proposed by Wadler [1986] and later reformulated by Okasaki [1998] have some resemblance with isomorphic ornaments. They allow several interchangeable representations for the same data, using isomorphism to switch between views, but, again, the isomorphisms persist at runtime. Lenses [Foster et al. 2007] also focus on switching representations at runtime.

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.

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/10.1145/1232420.1232424

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 RR-9117. Inria. https://hal.inria.fr/hal-01628060