Ornaments in practice, higher-order ornaments

Thomas Williams

(in collaboration with Didier Rémy, Pierre-Evariste Dagand)

December 11, 2014

Contents

1 Introduction

Inductive datatypes and parametric polymorphism were two key new features introduced in the ML family of languages in the 1980's. Datatypes stress the algebraic structure of data while parametric polymorphism allows to exploit the generic properties of algorithms working on algebraic structures. Arguably, ML has struck a balance between a precise classifying principle (datatypes) and a powerful abstraction mechanism (parametric polymorphism).

Datatype definitions are inductively defined as labeled sums and products over primitive types. This restricted language allows the programmer to describe, on the one hand, their recursive structures and, on the other hand, how to populate these structures with data of either primitive types or types given as parameters. A quick look at an ML library reveals that datatypes can be factorized through their recursive structures. For example, the type of leaf binary trees and the type of node binary trees both share a common binary-branching structure:

```
type  α ltree  =
  | LLeaf of α
  | LNode of α ltree × α ltree
type  α ntree  =
  | NLeaf
  | NNode of α ntree × α × α ntree
```

This realization is *mutatis mutandis* at the heart of the work on numerical representations [10] in functional settings [12, 9]. Having established the structural ties between two datatypes, one soon realizes that both admit strikingly similar functions, operating similarly over their common recursive structures. The user sometimes feels like repeatedly programming the same operations over and over again with only minor variations. The refactoring process by which one adapts existing code to work on another, similarly-structured datatype requires non-negligible efforts from the programmer. Could this process be automated?

Another tension arises from the recent adoption of indexed types, such as Generalized Algebraic Data Types (GADTs) [4, 16, 14] or refinement types [8, 1]. Indexed datatypes go one step beyond specifying the dynamic structure of data: they introduce a logical information enforcing precise static invariants. For example, while the type of lists is merely classifying data

```
type α list = Nil | Cons of α × α list
```

we can *index* its definition (here, using a GADT) to bake in an invariant over its length, thus obtaining the type of lists indexed by their length:

```
type zero = Zero          type _ succ = Succ
type (_, α) vec =
 | VNil : (zero, α) vec
 | VCons : α × (n, α) vec → (n succ, α) vec
```

Modern ML languages are thus offering novel, more precise datatypes. This puts at risk the fragile balance between classifying power and abstraction mechanism in ML.

Indeed, parametric polymorphism appears too coarse-grained to write program manipulating indifferently lists *and* vectors (but not, say, binary trees). We would like to abstract over the logical invariants (introduced by indexing) without abstracting away the common, underlying structure of datatypes.

The recent theory of ornaments [11] aims at answering these challenges. It defines conditions under which a new datatype definition can be described as an *ornament* of another. In essence, a datatype ornaments another if they both share the same recursive skeleton. Thanks to the structural ties relating a datatype and its ornamented counterpart, the functions that operate only on the structure of the original datatype can be semi-automatically lifted to its ornamented version.

The idea of ornaments is quite appealing but has so far only been explored formally, leaving open the question of whether ornaments are just a theoretician pearl or have real practical applications. This report aims at addressing this very question. Although this is still work in progress and we cannot yet draw firm conclusions at this stage, our preliminary investigation is rather encouraging.

Our contributions are sixfold: first, we present a concrete syntax for describing ornaments of datatypes and specifying the lifting of functions working on bare types to ornamented functions operating on ornamented types (§2); second, we describe the algorithm that given such a lifting specification transforms the definition of a function on bare types to a function operating on ornamented types (§2); third, we present a few typical use cases of ornaments where our semi-automatic lifting performs rather well in Sections §3 and §4; fourth, we define a notion of syntactic ornament that is used to explain the lifting algorithm and its properties in sections §6 and §6.7; fifth, we extend our definition of ornaments to include higher-order functions and nested ornaments (§7); finally, we have identified several interesting issues related to the implementation of ornaments that need to be investigated in future works (§8).

We have a very preliminary prototype implementation of ornaments. It has been used to process the examples presented below, up to some minor syntactical differences. Many type annotations have been omitted to mimic what could be done if we had ML-style type inference; our prototype still requires annotations on all function parameters. In this article, examples are typeset in a stylized, OCaml-like syntax: the actual definitions, as processed by our prototype, are available online[1].

2 Ornaments by examples

Informally, ornaments are relating "similar" datatypes. In this section, we aim at clarifying what we mean by "similar" and justifying why, from a software engineering standpoint, one would benefit from organizing datatypes by their "similarities".

For example, compare Peano's natural numbers and lists:

```
type nat    = Z   | S    of       nat
type α list = Nil | Cons of α × α list
```

[1] http://cristal.inria.fr/~remy/ornaments/

The two datatype definitions have a similar structure, which can be put in close correspondence if we map α `list` to `nat`, `Nil` to `Z`, and `Cons` to `S`. Moreover, the constructor `Cons` takes a recursive argument (of type α `list`) that coincides with the recursive argument of the constructor `S` of type `nat`. The only difference is that the constructor `Cons` takes an extra argument of type α. Indeed, if we take a list, erase the elements, and change the name of the constructor, we get back a natural number that represents the length of the list, as illustrated below:

```
Cons(1, Cons(2, Cons(3, Nil)))
S   (   S   (   S   (   Z  )))
```

This analysis also admits a converse interpretation, which is perhaps more enlightening from a software evolution perspective: lists can be understood as an extension of natural numbers that is obtained by grafting some information to the `S` constructor. To emphasize this correspondence, we say that the type α `list` is an *ornament* of the type `nat` with an extra field of type α on the constructor `S`.

One may then ask whether functions over natural numbers can be lifted to functions over lists [6]. For instance, the addition of Peano-encoded natural numbers

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

is strikingly similar to the `append` function over lists:

```
let rec append xs ys = match xs with
  | Nil → ys
  | Cons(x, xs') → Cons(x, append xs' ys)
```

Intuitively, addition can be recovered from the operation on lists by changing the constructors to their counterpart on natural numbers and simultaneously erasing the head field. [2]. However, this view of *erasing* information is not so interesting from a software engineering perspective, as one must decide in advance on a sufficiently rich datatype from which only a limited number of simpler versions can be derived.

Conversely, our interest lies in being able to lift a function operating on basic types, such as natural numbers to a function operating over some of its ornaments, such as lists.

The example of `append` is not a fortunate coincidence: several functions operating on lists admit a counterpart operating solely on integers. Rather than duplicating these programs, we would like to take advantage of this invariant to lift the code operating on numbers over to lists.

One should hasten to add that not every function over lists admits a counterpart over integers: for example, a function `filter` that takes a predicate `p` and a list `l` and returns the list of all the elements satisfying `p`, has no counterpart on integers, as the length of the returned list is not determined by the length of `l`.

From this informal description of ornaments, we can describe a recipe for programming with ornaments: start with a few basic structures (such as natural numbers, trees, *etc.*), build an ornamented structure by *extending* one of these with additional information and invariants, then (hopefully automatically) lift the functions from the base structure to the ornamented structure.

In this work, ornaments are a primitive language construct and are not definable using polytypic programming on a reflection of the definition of datatypes into the host language as is the case in [5]. Our goal here is to explore programming *with* ornaments and not programming ornaments themselves.

2.1 A syntax for ornaments

Informally, an ornament is *any* transformation of a datatype that preserves its underlying recursive structure, and provides a mapping from the values of the *ornamented type* to the values of the *bare type*. From an operational standpoint, this mapping is able to

- drop the extra information introduced by the ornament,
- transform the arguments of the *ornamented type* down to the *bare type*,
- while leaving untouched the common structure of the datatypes.

Dropping the extra-information can be easily described by a total *projection* function from the ornamented type to the bare type. For the `nat/list` case, the projection is the `length` function:

```
let rec length = function
  | Nil → Z
  | Cons(x, xs) → S(length xs)
```

Instead of providing a language for describing these transformations of types, we assume that both the bare type and the ornamented type are already defined. Then, an ornament is defined by the associated projection function, provided that it respects the structure of the datatypes. Hence, the ornamentation of natural numbers into lists is simply specified by the declaration

```
ornament from length : α list → nat
```

subject to certain conditions that we describe now.

The condition by which a projection "preserves the recursive structure" of its underlying datatype is somewhat harder to characterize syntactically. Let us first clarify what we mean by *recursive* structure. If we limit ourselves to a single, regular recursive type, the fields of each constructor can be divided into two sets: the recursive ones (for example, the tail of a list, or the left and right subtrees of a binary tree), and the non-recursive ones (for example, primitive types or parameters). A function preserves the recursive structure of a pair of datatypes (its domain and codomain) if it bijectively maps the recursive fields of the domain datatype (the ornament) onto the codomain datatype (its bare type).

From this definition, binary trees cannot be ornaments of lists, since trees have a constructor with two recursive fields, while lists only have a constant constructor and a constructor with a single recursive field; thus no function from trees to lists can preserve the recursive structure.

While we have a good semantic understanding of these conditions [5], we aim at giving a syntactic treatment. We are thus faced with the challenge of translating these notions to ML datatypes, which supports, for example, mutually-recursive datatypes.

From the categorical definition of ornaments, we can nonetheless extract a few sufficient syntactic conditions for a projection to define an ornament. For the sake of presentation, we will assume that the arguments of datatypes constructors are always ordered, non-recursive fields coming first, followed by recursive fields. The projection h defining the ornament must immediately pattern match on its argument, and the argument must not be used elsewhere. The constraints are expressed on each clause $p \to e$ of this pattern matching:

1. The pattern p must be of the form $C^{\dagger}(p_1, \ldots, p_m, x_1, \ldots, x_n)$ where $C^{\dagger}$ is a constructor of the ornamented type, the p_i are patterns matching the non-recursive fields, and the x_i's are variables matching the recursive fields.

2. The expression e must be of the form $C(e_1, \ldots, e_q, h\ y_1, ..., h\ y_n)$ where C is a constructor of the base type, the e_i's are expressions that do not use the x_j's, and the y_j's are a permutation of the x_i's.

In particular, a constructor $C^{\dagger}$ of the ornamented type will be mapped to a constructor C of the bare type with the same number of recursive fields.

Remark 1. Unlike the original presentation of ornaments [11], but following the categorical model [5], we allow the recursive arguments to be reordered.

These syntactic restrictions rules out all the following functions as definitions of ornaments:

```
let rec length_div2 = function
  | Nil → Z
  | Cons(_,Nil) → Z
  | Cons(x, Cons(y, xs)) → S(length_div2 xs)
```

The second (recursive) field of `Cons` is not matched by a variable in `length_div2`.

```
let rec length2 = function
  | Nil → Z
  | Cons(x, xs) → S(S(length2 xs))
```

The argument of the outer occurrence of `S` is not a recursive application of the projection `length2`.

```
let rec spine = function
  | NLeaf → Nil
  | NNode(l, x, r) → Cons(x, spine l)

let rec span = function
  | Nil → NLeaf
  | Cons(x, xs) → NNode(span xs, x, span xs)
```

The function `spine` is invalid because it discards the recursive field `r`, and `span` is invalid because it duplicates the recursive field `xs`.

The syntactic restrictions we put on the specification of ornaments make projections incomplete, *i.e.* one may cook up some valid ornaments that cannot be described this way, *e.g.* using arbitrary computation in the projection. However, it seems that interesting ornaments can usually be expressed as valid projections.

As expected, `length` satisfies the conditions imposed on projections and thus defines an ornament from natural numbers to lists.

Perhaps surprisingly, by this definition, the unit type is an ornament of lists (and, in fact, of any type inhabited by a non-recursive value), witnessed by the following function:

```
let nil () = Nil
ornament from nil : unit → α list
```

This example actually belongs to a larger class of ornaments that *removes* constructors from their underlying datatype (see more advanced uses of such examples in §3.3). From a type theoretic perspective, this is unsurprising: in the original presentation of ornaments, *removing* a constructor is simply achieved by *adding* a field to the constructor that belongs to an empty type.

The conditions on the ornament projection can be generalized to work with mutually recursive datatypes. To ornament a mutually recursive family of datatypes, we simply define a mutually recursive family of projections functions, one for each datatype. Individually, each of these projection functions are then subject to the same syntactic conditions.

2.2 Lifting functions: syntax and automation

Using the ornament projection, we can also relate a lifted function operating on some ornamented type with the corresponding function operating on its respective bare type. Intuitively, such a *coherence* property states that the results of the ornamented function are partially determined by the results of the *bare function* (the function on the bare type).

To give a more precise definition, let us define a syntax of functional ornaments, describing how one function is a *lifting* of another, and the coherence property that it defines. Suppose we want to lift a function f of type $\sigma \to \tau$ to the type $\sigma^\dagger \to \tau^\dagger$ using ornaments. More precisely, suppose we want this lifting to use the ornaments defined by the projections $u_\sigma : \sigma^\dagger \to \sigma$ and $u_\tau : \tau^\dagger \to \tau$. We say that $f^\dagger$ is a *coherent lifting* of f with the ornaments u_σ and u_τ if and only if it satisfies the equation:

$$f\ (u_\sigma\ x) = u_\tau\ (f^\dagger\ x)$$

for all arguments x of type $\sigma^\dagger$.

This definition readily generalizes to any number of arguments. For example, lifting the function `add` with the ornament `length` from natural numbers to lists, the property becomes:

$$\texttt{length}\ (f^\dagger\ \texttt{xs ys}) = \texttt{add (length xs) (length ys)}$$

And indeed, taking the function `append` for $f^\dagger$ satisfies this property. Thus, we can say that `append` is a *coherent lifting* of `add` with the ornament `length` used for both the arguments and the result. But is it the only one? Can we find it automatically?

So far, we have only specified when a function is a coherent lifting of another one. However, the whole point of ornaments is to automate the generation of the code of the lifted function. For instance, we would like to write

```
let lifting append from add
  with {length} → {length} → {length}
```

where `{length} → {length} → {length}` specifies the ornaments to be used for the arguments and the result (in the specification of a lifting, ornaments are identified with their projection functions). We then expect the compiler to automatically derive the definition of `append` for us. In practice, we will not get exactly the right definition, but almost.

To achieve this objective, the coherence property appears to be insufficiently discriminating. For instance, there is a plethora of coherent liftings of `add` with the ornament `length` beside `append`. Rather than trying to enumerate all of them, we choose to ignore all solutions whose syntactic form is not close enough to the original function. Our prototype takes hints from the syntactic definition of the bare function, thus sacrificing completeness. The system tries to guess the lifting based on the form of the original function and eventually relies on the programmer to supply code that could not be inferred.

Let us unfold this process on the lifting of `add` along `length`, as described above, where `add` is implemented as:

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

The lifting specification `{length} → {length} → {length}` plays several roles. First, its describes how the type of `add` should be transformed to obtain the type of `append`. Indeed, knowing that `length` has type α `list → nat` and `add` has type `nat → nat → nat`, `append` must have type α `list` → β `list` → γ `list` for some values of α, β, γ to be determined. Second, it describes how each argument and the result should be lifted. During the lifting process, the ornaments of the arguments and of the result play quite different roles: lifting an argument changes the context and thus requires lifting the associated pattern, which introduces additional information in the context; by contrast, lifting the return type requires lifting expressions, which usually needs additional information to be passed to the ornamented constructors; in general, this information cannot be inferred and therefore must be provided by the user.

On our example, the lifting specification says that the arguments `m` and `n` of the function `add` are lifted into some arguments `ml` and `nl` of the function `append` such that `length ml` is `m` and `length nl` is `n`. The matching on `m` can be automatically updated to work on lists instead of numbers, by simply copying the structure of the ornament declaration: the projection returns `Z` only when given `Nil`, while the constructor `S(-)` is returned for every value matching `Cons(x,-)` where `-` stands for the recursive argument.

The variable x is an additional argument to Cons that has no counterpart in S. As a first approximation, we obtain the following skeleton (the left-hand side gray vertical bar is used for code inferred by the prototype):

```
let rec append ml nl = match ml with
  | Nil → $a_1$
  | Cons(x, ml') → $a_2$
```

where the expressions a_1 and a_2 are still to be determined. By inspecting the projection function, it is clear that the variable ml' is the lifting of m'. In order to have a valid lifting, we require a_1 to be a lifting of n and a_2 to be a lifting of S(add m' n), both along length.

Let us focus on a_1. There are several possible ornaments of n: Indeed, we could compute the length of nl and return any list of the same length. However, we choose to return nl because we want to mirror the structure of the original function, and the original function does not destruct n in this case. That is, we restrict lifting of variables so that they are not destructed if they were not destructed in the bare version.

In the other branch we know that the value of a_2 must be an ornament of S(add m' n). To mimic the structure of the code, we must construct an ornament of this value. In this case, it is obvious by inspection of the ornament that there is only one possible constructor. Therefore a_2 must be of the form Cons(a_3,a_4), where a_3 is a term of the type α of the elements of the list and a_4 a lifting of add m' n. Upon encountering a function call to be lifted, the system tries to find a coherent lifting of the function among all previously declared liftings. Here, we know (recursively) that append is lifted from add with ornaments {length} → {length} → {length}. By looking at this specification, we may determine how the arguments must be lifted: Both m and n must be lifted along length and ml' and nl are such coherent liftings—and are the only ones in context.

To summarize, our prototype automatically generates the following code:

```
let rec append ml nl = match ml with
  | Nil → nl
  | Cons(x, ml') → Cons($\boxed{?}$, append ml' nl)
```

The notation $\boxed{?}$ represents a hole in the code: this part of the code could not be automatically lifted, since it does not appear in the original code, and it is up to the programmer to say what should be done to generate the element of the list.

To obtain the append function, we can put x in the hole, but there are other solutions that also satisfy the coherence property. For example, we could choose to take the first element of nl if it exists or x otherwise. The resulting function would also be a lifting of add, since whatever is in the hole is discarded by length. Note also that we could transform the list nl, instead of returning it directly in the Nil case, or do something to the list returned by append ml' nl in the Cons case, as long as we do not change the lengths.

While the generated code forces the type of nl to be equal to the return type, we could even imagine valid liftings where the types of the two arguments would be different and the elements of the list would not be equal: for example, filling the hole with () yields a function of type α list → unit list → unit list. To ensure that the obtained function is sufficiently polymorphic, it is possible to add an explicit type annotation when declaring the lifting:

```
let lifting
  append : type a. a list → a list → a list
  from add with {length} → {length} → {length}
```

When provided with such a signature, the behavior of our functions is greatly limited by parametricity: the elements must come from one of the lists. Thus, a possible enhancement to our algorithm is to try, in a post-processing pass, to fill the holes with a term of the right type, if it is unique up to program equivalence—for some appropriate notion of program equivalence. Since we have given up completeness, we can add the additional constraint that the term does not use any lifted value: the reason is that we do not want to destruct lifted values further than in the unlifted version. With this enhancement, the hole in append could be automatically filled with the appropriate value (but our prototype does not do this yet).

Notice that if we had a version of list carrying two elements per node, *e.g.* with a constructor Cons2 of type $\alpha \times \alpha \times \alpha$ list, the Cons branch would be left with two holes:

```
  | Cons2(x1, x2, ml') →
      Cons2( ? , ? , append ml' nl)
```

In this case, the post-processing pass would have no choice but leave the holes to be filled by the user, as each of them requires an expression of type α and there are two variables x_1 and x_2 of type α in the context.

Surprisingly, lifting the tail-recursive version add_bis of add:

```
let rec add_bis m n = match m with
  | Z → n
  | S(m') → add_bis m' (S n)
let lifting append_bis from add_bis
    with {length} → {length} → {length}
```

yields a very different function:

```
let rec append_bis ml nl = match ml with
  | Nil → nl
  | Cons(x, ml') →
      append_bis ml' (Cons( ? ,nl))
```

Filling the hole in the obvious way (whether manually or by post-processing), we get the reverse append function.

This example shows that the result of the lifting process depends not only on the observable behavior of a function (as expressed by the coherence property), but also on its implementation. This renders functional lifting sensitive to syntactic perturbations: one should have a good knowledge of how the bare function is written to have a good understanding of the function obtained by lifting. Conversely, extensionally equivalent definitions of a single bare function might yield observably distinct ornamented functions, as is the case with append and append_bis.

The implementation of automatic lifting stays very close to the syntax of the original function. This has interesting consequences, which we detail (§6 and §6.7) when justifying the correctness of the lifting. In particular, we show that if the projection has a constant cost per recursive call, then the complexity of the lifted function (excluding the complexity of computing what is placed in the holes) is no greater than the complexity of the bare function.

2.3 Patching the generated code

When the lifting leaves a hole in the code because some part of it cannot be automatically lifted, we could rely on a post-processing code inference phase to fill in the missing parts, as mentionned above. This may still fail—or make a wrong choice because of heuristics. In this case, the user can manually edit the lifted code by hand after post-processing. Yet another, perhaps more attractive solution is to provide, along with the lifting declaration, a *patch* for the generated function that will fill in or replace some parts of the generated code.

Our system includes an implementation of a preliminary language of patches for code. Patches follow the structure of the code, except that some parts can be omitted by replacing them with an underscore, and not all patterns have to be provided. New code is inserted by enclosing it in braces. This new code can use the names bound by the patterns of the patch. For example, the following declaration lifts the function `append` from `add` and fills the hole in `Cons`:

```
let append from add
  with {length} → {length} → {length}
  patch fun _ → match _ with Cons(x, _) → Cons({x}, _)
```

Patches can also be used to change a piece of code when the lifting or its post-processing made a wrong choice. In the lifting of `add` to `append`, the system chooses to return `nl` in the base case. The following patch overrides this behavior, and returns `List.rev nl` instead. This is still a valid lifting because `nl` and `List.rev nl` have the same length.

```
let append_rev from add
  with {length} → {length} → {length}
  patch fun ml nl → match _ with
    | Nil → List.rev nl
    | Cons(x, _) → Cons({x}, _)
```

The system then generates the following code:

```
let rec append_rev ml nl = match ml with
  | Nil → List.rev nl
  | Cons(x, ml’) → Cons(x, append_rev ml’ nl)
```

Currently, the implementation does not check that the user-supplied code respects the coherence property, but this would be desirable, at least to issue a warning when the coherence cannot be proved and an error when the code is obviously not coherent. In most cases, we expect the user to only have to insert a constructor or choose a variable from the context, so the coherence proofs should be simple enough.

Open question: We have described a basic language of patches to provide the code that cannot be inferred by the automatic lifting, but this language could certainly be improved. A language of patches may be evaluated on two criterions. It should be sufficiently predictable to allow the programmer to write the patch without looking at the lifted code instead of working interactively. The patches should also be robust to small changes in the original function.

3 Use cases

The examples in the previous sections have been chosen for exposition of the concepts and may seem somewhat contrived. In this section, we present two case studies that exercise ornaments in a practical setting. First, we demonstrate the use of lifting operations on a *larger scale* by transporting a library for sets into a library for maps (§3.1). Second, we show that ornaments can be used to direct *code refactoring* (§3.2 and §3.3), thus interpreting in a novel way the information provided by the ornament as a recipe for software evolution.

3.1 Lifting a library

The idea of lifting functions from one data structure to another one carries to more complex data structures, beyond the toy example of `nat` and `list`. In this section, we lift a (partial) implementation of sets based on unbalanced binary search trees to associative maps. We only illustrate the lifting of the key part of the library:

```
type key
val compare : key → key → int
type set = Empty | Node of key × set × set

let empty : set = Empty

let rec find : key → set → bool =
  fun k → function
    | Empty → false
    | Node(k', l, r) →
      if compare k k' = 0 then true
      else if compare k k' > 0 then find k l
      else find k r
```

Our goal is to lift the two operations `empty` and `find` to associative maps. In this process, we shall change the return type of `find` to α `option` to be able to return the value associated to the key. This is possible because α `option` can be seen as an ornament of `bool` where an extra field has been added to `true`:

```
type α option = None | Some of α
let is_some  =  function
  | Some _ → true
  | None → false
ornament from is_some : α option → bool
```

The interface of the map library should be:

```
type α map =
  | MEmpty
  | MNode of key × α × α map × α map
val mempty : α map
val mfind : key → α map → α option
```

We define the type α `map` as an ornament of `set`:

```
let rec keys  =  function
  | MEmpty → Empty
  | MNode(k, v, l, r) → Node(k, keys l, keys r)
ornament from keys : α map → set
```

We may now ask for a lifting of the two operations:

```
let lifting mempty from empty
  with {keys}
let lifting mfind from find
  with _ → {keys} → {is_some}
```

In the specification of `mfind` the first argument should not be lifted, which is indicated by writing an underscore instead of the name of a projection function, which in this case would be the identity. This information is exploited by the lifting process which can do more automation by knowing that the argument is not lifted.

The lifting of `mfind` is only partial, and the system replies with the lifted code below that contains a hole for the missing piece of information:

```
let mempty = MEmpty
let rec mfind = fun k → function
  | MEmpty → None
  | MNode(k', v, l, r) →
    if compare k k' = 0 then Some( ? )
    else if compare k k' > 0 then mfind k l
    else mfind k r
```

That is, the programmer is left with specifying which value should be included in the map for every key. The solution is of course to fill the hole with `v` (which here could be inferred from its type, as `v` is the only variable of type α in the current context).

Lifting OCaml's `Set` library: As a larger-scale experiment, we tried to automatically lift parts of OCaml's `Set` library to associative maps. Some functions can be lifted but their coherence properties do not capture the desired behavior over maps. For example, the lifting of the `equal` function on sets of keys to an `equal` function on maps would only check for equality of the keys. Indeed, by coherence, applying the lifted version to two maps should be the same as applying `equal` to the sets of keys of the two maps.

Still, for many functions, the lifting makes sense and, as in the `find` example above, the only holes we have to fill are those containing the values associated to keys. This is a straightforward process, at the cost of a few small, manual interventions from the programmer. Moreover, many of these could be avoided by performing some limited form of code inference in a post-processing phase.

Lifting of higher-order functions: Our current theory of ornaments remains first-order. Surprisingly, the syntactic lifting extends seamlessly to higher-order functions. This motivates the formulation of a theory of higher-order ornaments (§7). For example, OCaml's Set library provides the following function to check if a predicate holds for at least one element of the set.

```
let rec exists (p : elt → bool) (s : set) : bool =
  match s with
    | Empty → false
    | Node(l, k, r, _) → p k
              || exists p l || exists p r
```

We want to define a similar function `map_exists` on maps with the type `(elt` → α → `bool)` → α `map` → `bool`. To be able to express this lifting, the syntax of lifting specifications is extended to allow higher-order liftings.

```
let lifting map_exists from exists
  with (_ → +_ → _) → {keys} → bool
```

The syntactic lifting yields the following definition:

```
let rec map_exists p m =
  match m with
    | Empty → false
    | Node(l, k, v, r, _) → p k  ?
              || map_exists p l || map_exists p r
```

which happens to be exactly the function we are expecting if we plug the value `v` into the hole.

3.2 Refactoring

Another application of ornaments is related to code refactoring: upon reorganizing a datatype definition, without adding or removing any information, we would like to automatically update programs that manipulate that datatype.

For instance, consider the abstract syntax of a small programming language:

```
type expr =
  | Const of int
  | Add of expr × expr
  | Mul of expr × expr
let rec eval  = function
  | Const(i) → i
  | Add(u, v) → eval u + eval v
  | Mul(u, v) → eval u × eval v
```

As code evolves and the language gets bigger, a typical refactoring is to use a single constructor for all binary operations and have a separate datatype of operations, as follows:

```
type binop = Add' | Mul'
type expr' =
  | Const' of int
  | BinOp' of binop × expr' × expr'
```

By defining the `expr'` datatype as an ornament of `expr`, we get access to the lifting machinery to transport programs operating over `expr` to programs operating over `expr'`. This ornament is defined as follows:

```
let rec convert  = function
  | Const'(i) → Const(i)
  | BinOp(Add', u, v) → Add(convert u, convert v)
  | BinOp(Mul', u, v) → Mul(convert u, convert v)
ornament from convert : expr' → expr
```

We may now lift the `eval` function to the new representation:

```
let lifting eval' from eval
  with {convert} → _
```

In this case, the lifting is total and returns the following code:

```
let rec eval'  = function
  | Const'(i) → i
  | BinOp'(Add', u, v) → eval' u + eval' v
  | BinOp'(Mul', u, v) → eval' u × eval' v
```

Quite interestingly, the lifting is completely determined by the coherence property for strict refactoring applications because the ornament defines a bijection between the two types (here, `expr` and `expr'`). Here, we have hit a sweet spot where the ornament is sufficiently simple to be reversible on each constructor. This allows our system to lift the source program in totality.

3.3 Removing constructors

Another subclass of ornaments consists of those that remove some constructors from an existing type. Perhaps surprisingly, there are some interesting uses of this pattern: for example, in a compiler, the abstract syntax may have explicit nodes to represent syntactic sugar since the early passes of the compiler may need to maintain the difference between the sugared and desugared forms. However, one may later want to flatten out these differences and reason in the subset of the language that does not include the desugared forms—thus ensuring the stronger invariant that the sugared forms do not appear as inputs or ouputs.

Concretely, the language of expressions defined in the previous section (§3.2) could have been defined with a let construct (denoted by `lexpr`). The type `expr` is a subset of `lexpr`: we have an ornament of `lexpr` whose projection `to_lexpr` injects `expr` into `lexpr` in the obvious way:

```
type lexpr =
  | LConst of int
  | LAdd of lexpr × lexpr
  | LMul of lexpr × lexpr
  | Let of string × lexpr × lexpr
  | Var of string
let rec to_lexpr : expr → lexpr = function
  | Const n → LConst n
  | Add(e$_1$, e$_2$) → LAdd(to_lexpr e$_1$, to_lexpr e$_2$)
  | Mul(e1, e2) → LMul(to_lexpr e1, to_lexpr e2)
ornament from to_lexpr : expr → lexpr
```

As with the refactoring, lifting a function f operating on lexpr over to expr is completely determined by the coherence property. Still for the lifting to exist, the function f must verify the coherence property, namely that the images of f without sugared inputs are expressions without sugared outputs, and the lifting will fail whenever the system cannot verify this property, either because the property is false or because of the incompleteness of the verification. For example, the function mul_to_add introduces a let:

```
let mul_to_add  = function
  | LMul(LConst 2, x) →
    let n = gen_name() in
    Let(n, x, Add(Var n, Var n))
  | y → y
```

Hence, it is rejected (the left-hand side double bar is used to signal incorrect code):

```
let lifting mul_to_add' from mul_to_add
  with {to_lexpr} → {to_lexpr}
```

The system throws an error message and prints the partially lifted code to indicate the error location:

```
let mul_to_add'  = function
  | Mul(Const 2, x) →
    let n = gen_name() in
    !
  | y → y
```

4 GADTs as ornaments of ADTs

GADTs allow to express more precise invariants on datatypes. In most cases, a GADT is obtained by *indexing* the definition of another type with additional information. Depending on the invariants needed in the code, multiple indexings of the same bare type can coexist. But this expressiveness comes at a cost: for each indexing, many operations available over the bare type must be reimplemented over the finely-indexed types. Indeed, a well-typed function between two GADTs describes not only a process for transforming the data, but also a proof that the invariants of the result follow from the invariants carried by the input arguments. We would like to automatically generate these functions instead of first duplicating the code and then editing the differences, which is tedious and hinders maintainability.

The key idea is that indexing a type is an example of ornament. Indeed, to transport a value of the indexed type back to the bare type, it is only necessary to drop both the indices and the constraints embedded in values. The projection will thus map every indexed constructor back to its unindexed equivalent.

Let us consider the example of lists indexed by their length (or *vectors*) mentioned in the introduction:

```
type α list = Nil | Cons of α × α list
type zero = Zero          type _ succ = Succ
type (_, α) vec =
 | VNil : (zero, α) vec
 | VCons : α × (n, α) vec → (n succ, α) vec
```

We may define an ornament `to_list` returning the list of the elements of a vector (a type signature is required because `to_list` uses polymorphic recursion on the index parameter).

```
let rec to_list : type n. (n, α) vec → α list =
  function
  | VNil → Nil
  | VCons(x, xs) → Cons(x, xs)
ornament from to_list : (γ, α) vec → α list
```

This ornament maps, for all n, the type `(n, α) vec` to the type `α list`. In most cases of indexing ornaments, the function projecting the types is not injective: the additional constraints given by the indexing are forgotten. However, the projection of the values is injective. As for refactoring, the lifting of a function is thus unique. For more complex GADTs, the projection may forget some fields that only serve as a representation of a proof. Since proofs should not influence the results of the program, this ambiguity should not cause any issue.

In practice, lifting seems to work well for many functions. Take for example the `zip` function on lists:

```
let rec zip xs ys = match xs, ys with
  | Nil, Nil → Nil
  | Cons(x, xs), Cons(y, ys) → Cons((x, y), zip xs ys)
  | _ → failwith "different length"
```

When specifying the lifting of `zip`, we must also give the type of `vzip` to express the relation between the length of the arguments. It cannot be inferred automatically because the obtained function will be polymorphic recursive.

```
let lifting vzip :
  type n. (n, α) vec → (n, β) vec → (n, α × β) vec
  from zip with {to_list} → {to_list} → {to_list}
```

This lifting is fully automatic, thus generating the following code:

```
let rec vzip :
  type n. (n, α) vec → (n, β) vec → (n, α × β) vec
  = fun xs ys → match xs, ys with
  | VNil, VNil → VNil
  | VCons(x, xs), VCons(y, ys) →
     VCons((x, y), vzip xs ys)
  | _ → failwith "different length"
```

Observe that the structure of the lifted function is identical to the original. Indeed, the function on vectors could have been obtained simply by adding a type annotation and replacing each constructor by its vector equivalent. The last case of the pattern matching is now redundant, it could be removed in a subsequent pass.

The automatic lifting ignores the indices: the proofs of the invariants enforced by indexing is left to the typechecker. In the case of `vzip`, the type annotations provide enough information for OCaml's type inference to accept the program. However, this is not always the case. Take for example the function `zipm` that behaves like `zip` but truncates one list to match the length of the other:

```
let rec zipm xs ys = match xs, ys with
  | Nil, _ → Nil
  | _, Nil → Nil
  | Cons(x, xs), Cons(y, ys) → Cons((x, y), zipm xs ys)
```

To lift it to vectors, we need to encode the fact that one type-level natural number is the minimum of two others. This is encoded in the type `min`.

```
type (_, _, _) min =
  | MinS : (α, β, γ) min → (α su, β su, γ su) min
  | MinZl : (ze, α, ze) min
  | MinZr : (α, ze, ze) min
```

The lifting of `zipm` needs to take an additional argument that containts a witness of type `min`: this is indicated by adding a "+" sign in front of the corresponding argument in the lifting specification.

```
let lifting vzipm :
  type n1 n2 nmin.
    (n1, n2, nmin) min →
        (n1, α) vec → (n2, β) vec → (nmin, α × β) vec
  from zipm
  with +_ → {to_list} → {to_list} → {to_list}
```

This lifting is partial, and actually fails:

```
let rec vzipm :
  type n1 n2 nmin. (n1, n2, nmin) min
    → (n1, α) vec → (n2, β) vec → (nmin, α × β) vec
  = fun m xs ys → match xs, ys with
  | VNil, VNil → VNil
  | VCons(x, xs), VCons(y, ys) →
      VCons((x, y), vzipm [?] xs ys)
  | _, _ → failwith "different length"
```

Even though it behaves correctly, this function does not typecheck, even if we put a correct witness inside the hole: some type equalities need to be extracted from the witness `min`. This amounts to writing the following code:

```
let rec vzipm :
  type l1 l2 lm. (l1, l2, lm) min →
    (α, l1) vec → (β, l2) vec → (α × β, lm) vec =
  fun m xs ys → match xs, ys with
    | VNil, _ →
      (match m with MinZl → VNil | MinZr → VNil)
    | _, VNil →
      (match m with MinZr → VNil | MinZl → VNil)
    | VCons(x, xs), VCons(y, ys) →
```

```
(match m with
 | MinS m' → VCons((x, y), vzipm m' xs ys))
```

Generating such a code is out of reach of our current prototype. Besides, it contradicts our simplification hypothesis that ornaments should not (automatically) inspect arguments deeper than in the original code.

Instead of attempting to directly generate this code, a possible extension to our work would be to automatically search, in a post-processing phase, for a proof of the required equalities to generate code that typechecks, *i.e.* to generate the above code from the ouput of the partial lifting.

5 System F with recursive datatypes

We'll now give a more formal presentation of the lifting algorithm described in (§2.2). We therefore to precisely define the language on which our lifting algorithm operates: it can be seen as a call-by-value System F with type constructors or, alternatively, as an explicitly typed version of ML with unrestricted polymorphism. The results do not seem too dependent on the particular features of the language. In particular, they should adapt without difficulties to a language with a ML type system and GADTs. On the other hand, we anticipate some problems if we add references or similar side-effects to the language.

5.1 Syntax of terms and types

The syntax of types τ includes type variables α, universal quantification $\forall\alpha.\,\tau$ and application of a *type constructor* ε to a sequence of types. In the description of the syntax, we will use the notation $\widehat{\ }$ to refer to a sequence of objects.

$$\begin{aligned}
\tau &::= \alpha \mid \tau \to \tau \mid \forall\alpha.\,\tau \mid \varepsilon\,\widehat{\tau}\\
M &::= x \mid \lambda x{:}\tau.\,M \mid M\;M \mid \Lambda\alpha.\,M \mid M\;\tau \mid C(\widehat{\tau};\widehat{M}) \mid \mathsf{match}\;M\;\mathsf{with}\;\widehat{m} \mid \mathsf{fix}\,x{:}\tau.\,M\\
m &::= p \Rightarrow M\\
p &::= C(\widehat{x})
\end{aligned}$$

A term M can be a variable, the application of a term to a term or a type, a type (Λ) or value (λ) abstraction, a fixed point construction $\mathsf{fix}\,x{:}\tau.\,M$. If C is a constructor of the datatype ε, $C(\widehat{\tau};\widehat{M})$ constructs a value of $\varepsilon\,\widehat{\tau}$ tagged by the constructor C, with fields $\widehat{M}$. Expressions can be filtered by *pattern matching*: $\mathsf{match}\;M\;\mathsf{with}\;\widehat{m}$ examines the value of M and tries to find a matching clause in $\widehat{m}$ (the clauses are separated by vertical bars "|").

5.2 Datatypes, type and value constructors

Our base language allows the definition of ML-style *algebraic datatypes*. The syntax only describes the terms, datatypes are supposed predefined. In the typing rules, they will be given as a *datatype environment* describing the type and value constructors. A datatype environment D defines three functions:

1. The arity of a type constructor ε is written $\mathsf{arity}(D;\varepsilon)$.
2. Value constructors C construct a value of a given datatype $\mathsf{tcon}(D;C)$.
3. Each value constructor C takes a sequence $\widehat{\tau}$ of type arguments of length the arity of $\mathsf{tcon}(D;C)$, and has a number of values for its fields. To construct a value of type $\varepsilon\ \widehat{\tau}$, the fields must be filled with values of types given by the sequence of types $\mathsf{conty}(D;C;\widehat{\tau})$.

All type constructors are in scope when giving the types of the fields of a constructor. Thus, mutually recursive definitions are supported.

For example, to the following ML definition of lists:

```
type α list = Nil | Cons of α × α list
```

we associate the following datatype environment.

$$\mathsf{arity}(D;\mathtt{list}) \triangleq 1 \qquad \mathsf{tcon}(D;\mathtt{Nil}) \triangleq \mathtt{list} \qquad \mathsf{tcon}(D;\mathtt{Cons}) \triangleq \mathtt{list}$$

$$\mathsf{conty}(D;\mathtt{Nil};\alpha) \triangleq () \qquad \mathsf{conty}(D;\mathtt{Cons};\alpha) \triangleq (\alpha, \mathtt{list}\ \alpha)$$

5.3 Contexts, reduction

We are interested in call-by-value reduction in order to apply the results to ML. Substitution is written $s[x \leftarrow t]$ and multisubstitution, the simultaneous substitution of objects $\widehat{t}$ for variables $\widehat{x}$ in s, is written $s[\widehat{x} \leftarrow \widehat{t}]$. In multisubstitution, it is assumed that the lengths of the tuples match.

5.4 Environments, typing

The typing rules, given in figure 2, are of the form $D;\Gamma \vdash M : \tau$ and express that in the datatype environment D and the typing context Γ, the term M is well-typed of type τ. While Γ changes during the typing derivation, D is fixed: this reflects the fact that the datatype environment is considered fixed. The rule Match uses an additional predicate $\mathsf{valid}(\widehat{m})$ that checks whether a set of patterns is complete (all constructors of a datatype appear) and non-overlapping (no constructor appears twice).

6 Syntactic ornaments in the binary language

The principle of our lifting algorithm is to preserve the structure of the bare term as much as possible while generating the ornamented term. We make this common structure explicit, justifying the lifting process by a term that embeds both the base and the ornamented terms. These lifting witnesses are said to be *binary* terms because they

Values and evaluation contexts:

$$
\begin{aligned}
v &::= \lambda x{:}\tau.\, M \mid \Lambda\alpha.\, v \mid C(\widehat{\tau};\widehat{v}) \\
E &::= [\,] \mid M\ E \mid E\ v \mid E\ \tau \mid \Lambda\alpha.\, E \mid C(\widehat{\tau};\widehat{v}, E, \widehat{M}) \mid \mathsf{match}\ [\,]\ \mathsf{with}\ \widehat{m}
\end{aligned}
$$

Reducible expressions:

$$(\lambda x{:}\tau.\, M)\ v \hookrightarrow M[x \leftarrow v] \qquad\qquad (\Lambda\alpha.\, v)\ \tau \hookrightarrow v[\alpha \leftarrow \tau]$$

$$\mathsf{match}\ C(\widehat{\tau};\widehat{v})\ \mathsf{with} \cdots \mid C(\widehat{x}) \to M \mid \cdots \hookrightarrow M[\widehat{x} \leftarrow \widehat{v}] \qquad \mathsf{fix}\, x{:}\tau.\, M \hookrightarrow M[x \leftarrow \mathsf{fix}\, x{:}\tau.\, M]$$

Reduction in context:

$$\frac{M \hookrightarrow M'}{E[M] \longrightarrow E[M']}$$

Figure 1: Call-by-value semantics for the base language

contain two different terms: they can be *projected* both to a bare term and to an ornamented term. The two projected terms are said to related by *syntactic lifting.* For instance, the lifting of add to append is witnessed by the following term. The symbol & has no special meaning, but is a visual hint to form indentifiers that stand for the superposition of a base and an ornamented term. Braces are used to indicate that a piece of code only exists on the ornamented side.

```
let rec add&append m&ml n&nl = match m&ml with
  | Z&Nil → n&nl
  | S&Cons({x},m'&ml') → S&Cons({x}, add&append m'&ml' n&nl)
```

Indeed, we can recover either add or append from this term by the following *projections*: we obtain add by erasing all terms between braces and renaming the variables; we obtain append by erasing just the braces around terms and renaming the identifiers.

The binary language is not a mere syntactic artifact of the lifting process. We can equip it with a type system expressing the correctness of a lifting: the lifting specifications become the types of this system, and the meaning of the typing judgments is that a binary term expresses a valid lifting for a given specification. The lifting witnesses are also executable: the reductions of the binary terms translate to reductions of the bare and ornamented terms. The binary type system is sound with respect to these dynamic semantics. Together, the binary language along with its static and dynamic semantics give us a very precise account of the relation between an automatically lifted term and its bare version in terms of syntax, typing, and operational behavior.

Still, the binary language is only used as a formal tool: the user writes a term of the base language and an ornament specification. Using the specification as a guide, the lifting algorithm elaborates the bare term to a binary term of the correct type, which is projected to an ornamented term before being returned to the user.

Contexts:

$$\Gamma ::= \varnothing \mid \Gamma, x : \tau \mid \Gamma, \alpha$$

Well-formedness of contexts $D \vdash \Gamma$

$$D \vdash \varnothing \qquad \frac{D \vdash \Gamma \qquad x \notin \mathrm{dom}(\Gamma) \qquad D;\Gamma \vdash \tau}{D \vdash \Gamma, x : \tau} \qquad \frac{D \vdash \Gamma \qquad \alpha \notin \mathrm{dom}(\Gamma)}{D \vdash \Gamma, \alpha}$$

Well-kindedness of types $D;\Gamma \vdash \tau$:

$$\frac{D \vdash \Gamma \qquad \alpha \in \Gamma}{D;\Gamma \vdash \alpha} \quad \frac{D;\Gamma \vdash \tau_1 \qquad D;\Gamma \vdash \tau_2}{D;\Gamma \vdash \tau_1 \to \tau_2} \quad \frac{D;\Gamma, \alpha \vdash \tau}{D;\Gamma \vdash \forall \alpha.\, \tau} \quad \frac{\mathsf{arity}(D;\varepsilon) = n \qquad D;\Gamma \vdash \tau_i}{D;\Gamma \vdash \varepsilon\ \tau_1\ \ldots\ \tau_n}$$

Typing of terms $D;\Gamma \vdash M : \tau$:

$$\textsc{Var}\ \frac{\Gamma(x) = \tau}{D;\Gamma \vdash x : \tau} \qquad \textsc{Lam}\ \frac{D;\Gamma, x : \tau_1 \vdash M : \tau_2}{D;\Gamma \vdash \lambda x{:}\tau_1.\, M : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{D;\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad D;\Gamma \vdash M_2 : \tau_1}{D;\Gamma \vdash M_1\ M_2 : \tau_2}$$

$$\textsc{TLam}\ \frac{D;\Gamma, \alpha \vdash M : \tau}{D;\Gamma \vdash \Lambda \alpha.\, M : \forall \alpha.\, \tau} \qquad \textsc{TApp}\ \frac{D;\Gamma \vdash M : \forall \alpha.\, \tau_2 \qquad D;\Gamma \vdash \tau_1}{D;\Gamma \vdash M\ \tau_1 : \tau_2[\alpha := \tau_1]}$$

$$\textsc{Cons}\ \frac{\mathsf{tcon}(D;C) = \varepsilon \wedge \mathsf{conty}(D;C;\widehat{\tau}) = \widehat{\tau'} \qquad D;\Gamma \vdash M_i : \tau'_i}{D;\Gamma \vdash C(\widehat{\tau};\widehat{M}) : \varepsilon\,\widehat{\tau}}$$

$$\textsc{Match}\ \frac{D;\Gamma \vdash M : \varepsilon\,\widehat{\tau} \qquad \Gamma \vdash m_i : \varepsilon\,\widehat{\tau} \to \tau' \qquad \mathsf{valid}(\widehat{m})}{D;\Gamma \vdash \mathsf{match}\ M\ \mathsf{with}\ \widehat{m} : \tau'} \qquad \textsc{Fix}\ \frac{D;\Gamma, x : \tau \vdash M : \tau}{D;\Gamma \vdash \mathsf{fix}\, x{:}\tau.\, M : \tau}$$

Typing of patterns $D;\Gamma \vdash m : \tau \to \tau'$:

$$\textsc{Pat}\ \frac{\mathsf{tcon}(D;C) = \varepsilon \wedge \mathsf{conty}(D;C;\widehat{\tau}) = \widehat{\tau'} \qquad D;\Gamma, \widehat{x} : \widehat{\tau'} \vdash M : \tau''}{D;\Gamma \vdash C(\widehat{x}) \to M : \varepsilon\,\widehat{\tau} \to \tau''}$$

Figure 2: Typing of the base language

6.1 From ornaments to projection

The binary language, whose precise syntax is described in §6.2, is linked to the base language by two projection functions: one extracts the bare term and the other the ornamented term. Since ornamentation is first concerned by transformations on datatypes, the behavior of the projection will be defined by its image on type and data constructors of the binary language. The datatypes of the binary language and the projections on constructors are derived from the definitions of the ornaments: the constructors in the binary language represent the relation between a constructor in the base and in the ornamented language.

In order to have a simple translation from the binary language to the unary language, we sacrifice some convenience in ornaments definitions and limit them to some very basic transformation. We hope that the results will generalize gracefully to more complex transformations. Namely, we limit our study to ornaments that use a shallow pattern matching, do not reorder values in constructors, and *injectively* map the constructors of the ornamented type to those of the base type. Thus, constructors of the binary type represent either pairs of constructors of the ornamented type and the base type, or constructors of the base type that do not match a constructor of the ornamented type.

For example, the ornaments from naturals to lists is valid.

```
type nat = Z | S of nat
type α list = Nil | Cons of α × α list

let rec length = function
  | Nil → Z
  | Cons(x,xs) → S(length xs)
```

It can be described by the following type definition and projections in the binary language:

```
type α natlist = ZNil | SCons of α × α natlist
```

$$
\begin{array}{lcllcl}
\llbracket \mathtt{ZNil} \rrbracket^{\downarrow} & \triangleq & \mathtt{Z} & \llbracket \mathtt{SCons}(x, xs) \rrbracket^{\downarrow} & \triangleq & \mathtt{S}(\llbracket xs \rrbracket^{\downarrow}) \\
\llbracket \mathtt{ZNil} \rrbracket^{\uparrow} & \triangleq & \mathtt{Nil} & \llbracket \mathtt{SCons}(x, xs) \rrbracket^{\uparrow} & \triangleq & \mathtt{Cons}(\llbracket x \rrbracket^{\uparrow}, \llbracket xs \rrbracket^{\uparrow})
\end{array}
$$

As a counterexample, the ornament from α `list` to α `list2`, with binary type α `blist2` defined as

```
type α list2 = Nil2 | Cons1 of α × α list2 | Cons2 of α × α list2
type α blist2 = BNil | BCons1 of α × α blist2 | BCons2 of α × α blist2
```

$$
\begin{array}{rcllcl}
\llbracket \mathtt{BNil} \rrbracket^{\downarrow} & \triangleq & \mathtt{Nil} & \llbracket \mathtt{BNil} \rrbracket^{\uparrow} & \triangleq & \mathtt{Nil} \\
\llbracket \mathtt{BCons1}(x, xs) \rrbracket^{\downarrow} & \triangleq & \mathtt{Cons}(\llbracket x \rrbracket^{\downarrow}, \llbracket xs \rrbracket^{\downarrow}) & \llbracket \mathtt{BCons1}(x, xs) \rrbracket^{\uparrow} & \triangleq & \mathtt{Cons1}(\llbracket x \rrbracket^{\uparrow}, \llbracket xs \rrbracket^{\uparrow}) \\
\llbracket \mathtt{BCons2}(x, xs) \rrbracket^{\downarrow} & \triangleq & \mathtt{Cons}(\llbracket x \rrbracket^{\downarrow}, \llbracket xs \rrbracket^{\downarrow}) & \llbracket \mathtt{BCons2}(x, xs) \rrbracket^{\uparrow} & \triangleq & \mathtt{Cons2}(\llbracket x \rrbracket^{\uparrow}, \llbracket xs \rrbracket^{\uparrow})
\end{array}
$$

is invalid, as the base projection from α `blist2` to α `list` is not injective, since it maps both `Bcon1` and `Bcon2` to `Cons`—which would lead to overlapping cases in pattern matchings of the base functions. The coherence property requires that the information that we are in the case `Cons1` or `Cons2` should only be exploited on the ornamented side of the code. While our system could be extended to allow this kind of ornaments by syntactically comparing the code on both branches, it is possible to express a similar intent with the current restrictions by adding a boolean field to the ornamented version of `Cons`: this field will only be usable in the ornamented code.

```
type α list2 = Nil2 | Cons2 of bool × α × α list2
type α blist2 = BNil | BCons of {bool} × α × α blist2
```

$$\begin{aligned}
[\![\mathtt{BNil}]\!]^{\downarrow} &\triangleq \mathtt{Nil} \\
[\![\mathtt{BCons}(b, x, xs)]\!]^{\downarrow} &\triangleq \mathtt{Cons}([\![x]\!]^{\downarrow}, [\![xs]\!]^{\downarrow})
\end{aligned}$$

$$\begin{aligned}
[\![\mathtt{BNil}]\!]^{\uparrow} &\triangleq \mathtt{Nil} \\
[\![\mathtt{BCons}(b, x, xs)]\!]^{\uparrow} &\triangleq \mathtt{Cons2}(b, [\![x]\!]^{\uparrow}, [\![xs]\!]^{\uparrow})
\end{aligned}$$

The price we pay for this simplification is that most examples of refactoring on types are not supported by this presentation. This constraint could be relaxed at the cost of a more complex syntax, forcing branches to be put in parallel. But this kind of refactoring is orthogonal to the feature of ornaments we are concerned about here, which is the introduction of new information in the constructors. Since the refactoring transformations are bijective, we expect their theory not to be problematic.

6.2 Syntax of the binary language

The syntax of the binary language is quite closely related to the syntax of the unary language: indeed we want the projection functions to be as straightforward as possible. There are still some important changes. To emphasize the similarities between the two, meta-variables used for the binary language are underlined versions of meta-variables used for the unary language. The syntax of the binary language also allows the insertion of constructs of the unary language between single or double braces at certain occurrences. The meaning of these insertions will be expanded upon later. The corresponding meta-variables are either subscripted with an arrow pointing up $a_{\uparrow}$ if an ornamented term is allowed, or pointing down $a_{\downarrow}$ if a bare term is accepted. Another change is that functions now take *tuples* of arguments, noted between angle brackets $\langle \_ \rangle$ in types, values and patterns. The kinds κ are used for kinding type constructors: a type variable with kind ☆ must be instanciated with a binary type, while a variable with kind {☆} requires an ornamented type. Apart from these changes we will come back to later, the syntax is identical to the syntax of the base language.

$$\begin{aligned}
\kappa &::= \star \mid \{\star\} \\
\alpha_\uparrow &::= \alpha \mid \{\alpha\} \\
\underline{\tau} &::= \alpha \mid \langle\widehat{\tau_\uparrow}\rangle \to \underline{\tau} \mid \forall\alpha_\uparrow.\, \underline{\tau} \mid \varepsilon\, \widehat{\tau_\uparrow} \\
\tau_\uparrow &::= \underline{\tau} \mid \{\tau\} \\
\underline{M} &::= x \mid \lambda\langle\widehat{b_\uparrow}\rangle.\, \underline{M} \mid \underline{M}\ \langle\widehat{M_\uparrow}\rangle \mid \Lambda\alpha_\uparrow.\, \underline{M} \mid \underline{M}\ \tau_\uparrow \mid C(\widehat{\underline{\tau}}; \widehat{M_\uparrow}) \\
&\quad \mid \mathsf{match}\ \underline{M}\ \mathsf{with}\ \widehat{m_\downarrow} \mid \mathsf{fix}\, x\,{:}\,\underline{\tau}.\, \underline{M} \\
M_\uparrow &::= \underline{M} \mid \{M\} \\
b_\uparrow &::= x : \underline{\tau} \mid \{x : \tau\} \\
m_\downarrow &::= \underline{p} \Rightarrow \underline{M} \mid \{\{p \Rightarrow M\}\} \\
\underline{p} &::= \overline{C}(\widehat{x_\uparrow}) \\
x_\uparrow &::= x \mid \{x\}
\end{aligned}$$

Let's also define the evaluation contexts and the values. The contexts $\underline{E}$ can only take a normal binary term, while the contexts $E_\uparrow$ can only take an unary term (appearing between brackets):

$$\begin{aligned}
\underline{v} &::= \lambda\langle\widehat{b_\uparrow}\rangle.\, \underline{M} \mid \Lambda\alpha_\uparrow.\, \underline{M} \mid C(\widehat{\underline{\tau}}; \widehat{v_\uparrow}) \\
v_\uparrow &::= \underline{v} \mid \{v\} \\
\underline{E} &::= [\,]\ \langle\widehat{v_\uparrow}\rangle \mid [\,]\ \underline{\tau} \mid \Lambda\alpha_\uparrow.\, [\,] \mid \mathsf{match}\ [\,]\ \mathsf{with}\ \widehat{m_\downarrow} \mid \underline{M}\ \langle\widehat{M_\uparrow}, \{[\,]\}, \widehat{v_\uparrow}\rangle \mid C(\widehat{\tau_\uparrow}; \widehat{v_\uparrow}, \{[\,]\}x, \widehat{t_\uparrow}) \\
E_\uparrow &::= \underline{M}\ \langle\widehat{M_\uparrow}, \{[\,]\}, \widehat{v_\uparrow}\rangle \mid C(\widehat{\tau_\uparrow}; \widehat{v_\uparrow}, \{[\,]\}, \widehat{t_\uparrow})
\end{aligned}$$

6.2.1 Escape to the unary language

The constructors of the ornamented type will hold some additional information (*e.g.* the elements of a list, ...). This information can be used in later computations but the results of these computations are not available to the bare code: once extracted, the bare code will not have access to the data used to carry out the computation. This separation is enforced in the syntax: the bindings of variables only available in the ornamented code are written between braces, and these variables can only be used by code written between braces. Since such code only appears on the ornamented side, it is written directly in the syntax of the unary language. This construction has already been demonstrated in the `add&append` example in the introduction of this section.

Symmetrically, if we erase some constructors from a datatype, some branches will only exist in the base version of a function. In this case, they are surrounded in double braces and written in the syntax of the unary language. For example, we have seen that unit is an ornament of lists (in §2.1); then, the following code implements both the zero constant function and the list length function.

```
let rec const_length l = match l with
  | Unit_Nil → 0
{{| Cons(x,l') → 1 + const_length l' }}
```

Single braces can also occur in types. Their meaning is the same as for terms: the parts between braces will be erased when projecting to a bare type. To ensure that the projection is correct, the syntax of types limits the places where braces are valid. While quantifying on a variable they serve to introduce a quantifier exclusive to the ornamented side. Braced types can also be used as arguments of type constructors and as types for function arguments. In particular, this prevents erasing the codomain of a function.

6.2.2 Multi-application

It is possible to add new arguments to functions in the ornamented version of a function that have no counterpart in the original version. Following our already established notation, these arguments are written between braces and will only be available to the code in the *ornamented* version. For example, `myfun` erases to `myfun_base` and `myfun_ornamented`.

```
let myfun x {y} = ... g {f y} z ...
let myfun_base x = ... g z ...
let myfun_ornamented x y = ... g (f y) z
```

Unfortunately, this erasure of extra arguments may not always preserve the intended call-by-value semantics, since a unary function could be erased to a nullary function, *i.e.* an expression that would be evaluated earlier than intended, and may thus lead to non-termination of the base code while the ornamented code would terminate.

A simple solution to avoid this problem is to allow multiple abstractions and applications, and require that all erased arguments are so grouped with a non-erased argument. Hence, all grouped arguments must be applied at once, thus preserving expressions evaluation order. In practice, we defined a *tuple abstraction* and a *tuple application*, under the condition that all tuples contain at least one non-erased term.

To maintain a clear distinction between binary and base terms, we always use multi-applications (in angle brackets) in binary terms even if only one argument is applied.

6.3 Projection

Before defining the typing and evaluation rules for the binary language, we need to define the projection of types, terms, *etc.* We assume that the projections $\llbracket \_ \rrbracket^{\downarrow}$ and $\llbracket \_ \rrbracket^{\uparrow}$ are defined on the types and data constructors (this definition will be extracted from the ornament definitions). The intuition of this projection has already given on examples: to generate a bare term, the expressions, types, and bindings between single braces are erased, while the pattern matching clauses between double braces are kept. Conversely, to generate an ornamented term, the constructions between single brackets are kept and the clauses between double braces are erased. Moreover, tuple abstractions and applications are translated back to sequences of single abstractions and applications. Finally, type and data constructors are projected to their base or ornamented versions.

When defining the translation, we often need to filter a sequence to keep only the members that are not between single or double braces, and thus can be projected to the base or ornamented version. The projection on sequences is defined in this way, with p, q denoting the concatenation of sequences and $\varnothing$ the empty sequence. We use η as a

Projection for types $\llbracket \underline{\tau} \rrbracket^\eta$

$$\llbracket \alpha \rrbracket^\eta = \alpha \qquad \llbracket \langle \widehat{\tau_\uparrow} \rangle \to \underline{\tau}' \rrbracket^\eta = \llbracket \widehat{\tau_\uparrow} \rrbracket^\eta \to \llbracket \underline{\tau}' \rrbracket^\eta \qquad \llbracket \forall \alpha.\, \underline{\tau} \rrbracket^\eta = \forall \alpha.\, \llbracket \underline{\tau} \rrbracket^\eta \qquad \llbracket \varepsilon\ \widehat{\tau_\uparrow} \rrbracket^\eta = \llbracket \varepsilon \rrbracket^\eta\ \llbracket \widehat{\tau_\uparrow} \rrbracket^\eta$$

$$\llbracket \forall \{\alpha\}.\, \underline{\tau} \rrbracket^\downarrow = \llbracket \underline{\tau} \rrbracket^\downarrow \qquad \llbracket \forall \{\alpha\}.\, \underline{\tau} \rrbracket^\uparrow = \forall \alpha.\, \llbracket \underline{\tau} \rrbracket^\uparrow$$

Projection for terms

$$\llbracket x \rrbracket^\eta = x \qquad \llbracket \lambda \langle \widehat{b_\uparrow} \rangle.\, \underline{M} \rrbracket^\eta = \lambda \llbracket \widehat{b_\uparrow} \rrbracket^\eta.\, \llbracket \underline{M} \rrbracket^\eta \qquad \llbracket \underline{M}\ \langle \widehat{N_\uparrow} \rangle \rrbracket^\eta = \llbracket \underline{M} \rrbracket^\eta\ \llbracket \widehat{N_\uparrow} \rrbracket^\eta$$

$$\llbracket \Lambda \alpha.\, \underline{M} \rrbracket^\eta = \Lambda \alpha.\, \llbracket \underline{M} \rrbracket^\eta \qquad \llbracket \Lambda \{\alpha\}.\, \underline{M} \rrbracket^\downarrow = \llbracket \underline{M} \rrbracket^\downarrow \qquad \llbracket \Lambda \{\alpha\}.\, \underline{M} \rrbracket^\uparrow = \Lambda \alpha.\, \llbracket \underline{M} \rrbracket^\uparrow$$

$$\llbracket \underline{M}\ \underline{\tau} \rrbracket^\eta = \llbracket \underline{M} \rrbracket^\eta\ \llbracket \underline{\tau} \rrbracket^\eta \qquad \llbracket \underline{M}\ \{\tau\} \rrbracket^\downarrow = \llbracket M \rrbracket^\downarrow \qquad \llbracket \underline{M}\ \{\tau\} \rrbracket^\uparrow = \llbracket \underline{M} \rrbracket^\uparrow\ \tau$$

$$\llbracket \mathsf{match}\ \underline{M}\ \mathsf{with}\ \widehat{m_\downarrow} \rrbracket^\eta = \mathsf{match}\ \llbracket \underline{M} \rrbracket^\eta\ \mathsf{with}\ \llbracket \widehat{m_\downarrow} \rrbracket^\eta \qquad \llbracket C(\widehat{\tau_\uparrow}; \widehat{M_\uparrow}) \rrbracket^\eta = \llbracket C \rrbracket^\eta(\llbracket \widehat{\tau_\uparrow} \rrbracket^\eta; \llbracket \widehat{M_\uparrow} \rrbracket^\eta)$$

$$\llbracket \mathsf{fix}\, x : \underline{\tau}.\, \underline{M} \rrbracket^\eta = \mathsf{fix}\, x : \llbracket \underline{\tau} \rrbracket^\eta.\, \llbracket \underline{M} \rrbracket^\eta$$

Projection for bindings and patterns

$$\llbracket C(\widehat{x_\uparrow}) \rrbracket^\eta = \llbracket C \rrbracket^\eta(\llbracket \widehat{x_\uparrow} \rrbracket^\eta) \qquad \llbracket x : \underline{\tau} \rrbracket^\eta = x : \llbracket \underline{\tau} \rrbracket^\eta \qquad \llbracket p \to M \rrbracket^\eta = \llbracket p \rrbracket^\eta \to \llbracket M \rrbracket^\eta$$

meta-variable for ↑ and ↓.

$$\llbracket \{p\} \rrbracket^\uparrow = \llbracket p \rrbracket^\uparrow \qquad \llbracket \{\{p\}\} \rrbracket^\uparrow = \varnothing \qquad \llbracket \{q\} \rrbracket^\downarrow = \varnothing \qquad \llbracket \{\{q\}\} \rrbracket^\downarrow = \llbracket q \rrbracket^\downarrow$$

$$\llbracket p, q \rrbracket^\eta = \llbracket p \rrbracket^\eta, \llbracket q \rrbracket^\eta \qquad \llbracket \varnothing \rrbracket^\eta = \varnothing$$

For convenience, we write $\tau_1, \ldots, \tau_n \to \tau'$ for $\tau_1 \to \ldots \to \tau_n \to \tau'$, $\lambda b_1, \ldots, b_n.\, t$ for $\lambda b_1. \ldots \lambda b_n.\, t$, etc.

The transformation is the same on terms and on values. The projection should map binary values to values, but this is not the case: $\lambda \langle \{x : \tau\} \rangle.\, \underline{M}$ projects on the base language to $\llbracket \underline{M} \rrbracket^\downarrow$. However, under the hypothesis that multi-abstractions must contain at least one argument that is not braced, each multi-abstraction yields at least one λ-abstraction in each of the projected terms *i.e.* a value.

6.4 Datatype environment

The datatype environment for the binary language is defined similarly to its unary counterpart. Instead of being a number of arguments, the arity of a type constructor is a list of tags: ☆ or {☆}. The fields of a constructor now have a type in $\tau_\uparrow$: either it exists in both the bare and ornamented versions, or only in the ornamented version. The constructors that do not exist in the ornamented version are not defined in D, but in the datatype environment of the bare terms.

For example, to the definition of lists in the unary language:

```
type α list = Nil | Cons of α × α list
```

we associate the following datatype environment:

$$\mathsf{arity}(D; \mathtt{list}) \triangleq 1 \qquad \mathsf{tcon}(D; \mathtt{Nil}) \triangleq \mathtt{list} \qquad \mathsf{tcon}(D; \mathtt{Cons}) \triangleq \mathtt{list}$$

$$\mathsf{conty}(D; \mathtt{Nil}; \alpha) \triangleq () \qquad \mathsf{conty}(D; \mathtt{Cons}; \alpha) \triangleq (\alpha, \mathtt{list}\ \alpha)$$

and to the definition of the ornaments between lists and naturals in the binary language:

```
type α natlist = ZNil | SCons of α × α natlist
```

we associate the following the environment

$$\mathsf{arity}(D; \mathtt{natlist}) \triangleq (\{☆\}) \qquad \mathsf{tcon}(D; \mathtt{ZNil}) \triangleq \mathsf{tcon}(D; \mathtt{SCons}) \triangleq \mathtt{natlist}$$

$$\mathsf{conty}(D; \mathtt{Nil}; \{\alpha\}) \triangleq () \qquad \mathsf{conty}(D; \mathtt{Cons}; \{\alpha\}) \triangleq (\{\alpha\}, \mathtt{natlist}\ \{\alpha\})$$

The datatype environment D in the binary language must be *coherent* with the bare environment $D^{\downarrow}$ and the ornamented environment $D^{\uparrow}$. The coherence property ensures that the evaluations and typing derivations will translate faithfully.

Once the projection is defined, we say the environments $D, D^{\downarrow}, D^{\uparrow}$ are coherent if and only if the following properties hold, where the projections $\llbracket \_ \rrbracket^{\downarrow}$ and $\llbracket \_ \rrbracket^{\uparrow}$ on arities count the number of arguments that are not erased by the projections (*i.e.* all arguments for the ornamented code and all non-$\{☆\}$ arguments for the base code).

$$\mathsf{arity}(D^{\downarrow}; \llbracket \varepsilon \rrbracket^{\downarrow}) = \llbracket \mathsf{arity}(D; \varepsilon) \rrbracket^{\downarrow} \qquad \mathsf{tcon}(D^{\downarrow}; \llbracket C \rrbracket^{\downarrow}) = \llbracket \mathsf{tcon}(D; C) \rrbracket^{\downarrow}$$

$$\mathsf{arity}(D^{\uparrow}; \llbracket \varepsilon \rrbracket^{\uparrow}) = \llbracket \mathsf{arity}(D; \varepsilon) \rrbracket^{\uparrow} \qquad \mathsf{tcon}(D^{\uparrow}; \llbracket C \rrbracket^{\uparrow}) = \llbracket \mathsf{tcon}(D; C) \rrbracket^{\uparrow}$$

$$\mathsf{conty}(D^{\downarrow}; \llbracket C \rrbracket^{\downarrow}; \llbracket \widehat{\tau_{\uparrow}} \rrbracket^{\downarrow}) = \llbracket \mathsf{conty}(D; C; \widehat{\tau_{\uparrow}}) \rrbracket^{\downarrow} \qquad \mathsf{conty}(D^{\uparrow}; \llbracket C \rrbracket^{\uparrow}; \llbracket \widehat{\tau_{\uparrow}} \rrbracket^{\uparrow}) = \llbracket \mathsf{conty}(D; C; \widehat{\tau_{\uparrow}}) \rrbracket^{\uparrow}$$

Moreover, the projection should verify the following properties:

1. the base and ornamented projections do not identify two constructors of the same type, *i.e.* if $\mathsf{tcon}(D; C) = \mathsf{tcon}(D; C')$ and $\llbracket C \rrbracket^{\downarrow} = \llbracket C' \rrbracket^{\downarrow}$ or $\llbracket C \rrbracket^{\uparrow} = \llbracket C' \rrbracket^{\uparrow}$, then $C = C'$;

2. the ornamented projection is surjective: if $\mathsf{tcon}(D^{\uparrow}; C') = \llbracket \varepsilon \rrbracket^{\uparrow}$, then there exists a binary constructor C such that $\mathsf{tcon}(D; C) = \varepsilon$ and $\llbracket C \rrbracket^{\uparrow} = C'$.

These conditions ensure that the translated pattern matchings are both complete and non-ambiguous. Breaking the first condition leads to cases where the projected term has overlapping patterns even if the binary term does not, as in the `list2` case seen in §6.1. The second restriction ensures that we cannot create new constructors from thin air when building an ornament: the ornamented constructors have to be extensions of a base constructor. Cases handling constructors present in the base version but missing in the ornamented version have to be specified between double braces.

6.5 Dynamic semantics

The dynamic semantics of the binary language is defined to match the reduction steps of the dynamic semantics of the base language. The restrictions on the flow of information between ornamented and non-ornamented parts are enforced by the substitution: when substituting a variable bound between binding braces, the value is only substituted inside term braces: the variable is not in scope anywhere else. Thus, the substitution $x \leftarrow \underline{M}$ substitutes $\underline{M}$ for x outside braces and $[\![\underline{M}]\!]^{\uparrow}$ for x inside braces , while $\{x\} \leftarrow \{M\}$ substitutes M for x, but only inside braces *i.e.* in code that will appear only on the ornamented side. The definition of the substitution thus has some atypical rules:

$$\begin{array}{cc} x[\{x\} \leftarrow \{M\}] = x & \{N\}[x \leftarrow \underline{M}] = \{N[x \leftarrow [\![\underline{M}]\!]^{\uparrow}]\} \\ \{N\}[\{x\} \leftarrow \{M\}] = \{N[x \leftarrow M]\} & \{\{p \Rightarrow N\}\}[x \leftarrow \underline{M}] = \{\{p \Rightarrow N[x \leftarrow [\![\underline{M}]\!]^{\downarrow}]\}\} \end{array}$$

No syntax is needed for substituting variables only inside double braces because no variable is bound specifically for the non-ornamented part. Also note that in well-typed terms, the first case does never occur since x is out of scope at this point. For example, anticipating on the reduction rules that will be stated below, the term $(\lambda\langle x, y\rangle.\,\mathtt{SCons}(\{x\}, y))\ \langle\mathtt{ZNil}, \mathtt{ZNil}\rangle$ reduces to $\mathtt{SCons}(\{\mathtt{Nil}\}, \mathtt{ZNil})$: $\mathtt{ZNil}$ is projected to $\mathtt{Nil}$. The term $(\lambda\langle\{x\}, y\rangle.\,\mathtt{SCons}(\{x\}, y))\ \langle\{\mathtt{Nil}\}, \mathtt{ZNil}\rangle$ also reduces to $\mathtt{SCons}(\{\mathtt{Nil}\}, \mathtt{ZNil})$.

The dynamic semantics of the binary language are designed to match as closely as possible the dynamics of the unary language. In defining the evaluation, we will carefully annotate the reductions to distinguish two categories of reduction: some reductions translate to matching reductions on the bare and ornamented terms, while some other reductions yield only a reduction on the ornamented term. This is the case when the reduction involves brackets. Binary reductions are annotated in order to precisely measure the number of reductions: a reduction $\longrightarrow_{n,p}$ translates to n reductions on the bare term and $n+p$ reductions on the ornamented term. This precise accounting of reductions allows to prove the conjecture stated in §2.2 that the complexity of the ornamented term is the complexity of the base term plus the complexity of whatever terms were added in the holes of the ornamented code, which correspond to terms between braces in the binary code.

The relation $\longrightarrow_{n,p}$ defines the top-level reduction of binary terms. The tuple application reduces in one step, but it is translated to multiple applications in the base language. Thus, it reduces in a number of steps equal to the number of non-erased arguments. There are two types of type applications: either we apply a normal type or an ornamented type. The first reduction translates to one reduction on each side, while the second reduction only yields a reduction on the ornamented side. Reduction of a pattern matching yields one reduction on each side.

$$(\lambda\langle\widehat{x_\uparrow} : \widehat{\tau_\uparrow}\rangle.\, \underline{M})\ \langle\widehat{N_\uparrow}\rangle \hookrightarrow_{|[\![\widehat{x_\uparrow}]\!]^\downarrow|, |[\![\widehat{x_\uparrow}]\!]^\uparrow| - |[\![\widehat{x_\uparrow}]\!]^\downarrow|} M[\widehat{x_\uparrow} \leftarrow \widehat{N_\uparrow}]$$

$$\mathsf{match}\ C(\widehat{\tau_\uparrow}; \widehat{v_\uparrow})\ \mathsf{with} \cdots \mid C(\widehat{x_\uparrow}) \to \underline{M} \mid \cdots \hookrightarrow_{1,0} \underline{M}[\widehat{x_\uparrow} \leftarrow \widehat{v_\uparrow}]$$

$$\mathsf{fix}\, x : \underline{\tau}.\, \underline{M} \hookrightarrow_{1,0} \underline{M}[x \leftarrow \mathsf{fix}\, x : \underline{\tau}.\, \underline{M}] \qquad\qquad (\Lambda\alpha.\, \underline{v})\ \underline{\tau} \hookrightarrow_{1,0} \underline{v}[\alpha \leftarrow \underline{\tau}]$$

$$(\Lambda\{\alpha\}.\, \underline{v})\ \{\tau\} \hookrightarrow_{0,1} \underline{v}[\{\alpha\} \leftarrow \{\tau\}]$$

Double-braced clauses in pattern matching cannot be reduced, because they do not match a binary term.

The actual reduction is defined as the closure under evaluation context of the head reduction $\hookrightarrow$. Reduction under braces is counted as one ornamented reduction and no common reduction.

$$\textsc{Base}\ \frac{\underline{M} \hookrightarrow_{n,p} \underline{M'}}{\underline{M} \longrightarrow_{n,p} \underline{M'}} \qquad \textsc{Unary-Context}\ \frac{M \longrightarrow M'\ \text{(unary)}}{E_\uparrow[M] \longrightarrow_{0,1} E_\uparrow[M']} \qquad \textsc{Binary-Context}\ \frac{\underline{M} \longrightarrow_{n,p} \underline{M'}}{\underline{E}[\underline{M}] \longrightarrow_{n,p} \underline{E}[\underline{M'}]}$$

As expected, our accounting of reductions is accurate:

Theorem (Projection of reductions). Let $\underline{M}$ and $\underline{M'}$ be binary terms such that $\underline{M} \longrightarrow_{n,p} \underline{M'}$. Then $[\![\underline{M}]\!]^\downarrow \longrightarrow^n [\![\underline{M'}]\!]^\downarrow$ and $[\![\underline{M}]\!]^\uparrow \longrightarrow^{n+p} [\![\underline{M'}]\!]^\uparrow$.

The following corollary gives a coarser view of the dynamic semantics of the bare and ornamented terms:

Theorem. Let $\underline{M}$ be a binary term. If $[\![\underline{M}]\!]^\uparrow$ terminates, then $[\![\underline{M}]\!]^\downarrow$ terminates.

The converse is false: an ornamented term can fail to terminate whils its bare term terminates because of an infinite loop in code that only appears on the ornamented side.

6.6 Well-formedness, typing

The typing environments of the binary language can contain both normal type variables and type variables between braces, only usable in ornamented contexts. In the same way, it contains both binary bindings, giving the binary type of a variable usable in every context, and ornamented bindings, giving the type of a variable usable only in an ornamented context. The projection of the environments is defined by the rules for projection of bindings and sequences.

$$\underline{\Gamma} ::= \varnothing \mid \underline{\Gamma}, b_\uparrow \mid \underline{\Gamma}, \alpha_\uparrow$$

Typing rules for the binary language are given in Fig. 4. They are mostly a reformulation of the rules for System F, with special care taken to handle the bare and ornamented code: the two rules {{BPat}} and BOrn allow to switch to bare terms in patterns and to ornamented terms in terms. The validity of a pattern matching is defined as follows: all constructors of the binary type should appear once and only once, and all constructors of

Well-formedness of contexts $D \vdash \underline{\Gamma}$:

$$D \vdash \varnothing \qquad \frac{D \vdash \underline{\Gamma} \qquad \alpha, \{\alpha\} \notin \underline{\Gamma}}{D \vdash \underline{\Gamma}, \alpha} \qquad \frac{D \vdash \underline{\Gamma} \qquad \alpha, \{\alpha\} \notin \underline{\Gamma}}{D \vdash \underline{\Gamma}, \{\alpha\}}$$

Well-kindedness of types $D; \underline{\Gamma} \vdash \underline{\tau}$:

$$\frac{D \vdash \underline{\Gamma} \qquad x \notin \mathsf{dom}(\underline{\Gamma}) \qquad D; \underline{\Gamma} \vdash \underline{\tau}}{D \vdash \underline{\Gamma}, x : \underline{\tau}} \qquad \frac{D \vdash \underline{\Gamma} \qquad x \notin \mathsf{dom}(\underline{\Gamma}) \qquad D^{\uparrow}; [\![\underline{\Gamma}]\!]^{\uparrow} \vdash \tau}{D \vdash \underline{\Gamma}, \{x : \tau\}}$$

$$\frac{D^{\uparrow}; [\![\underline{\Gamma}]\!]^{\uparrow} \vdash \tau}{D; \underline{\Gamma} \vdash \{\tau\} : \{\star\}} \qquad \frac{D \vdash \underline{\Gamma} \qquad \alpha \in \underline{\Gamma}}{D; \underline{\Gamma} \vdash \alpha : \star} \qquad \frac{D; \underline{\Gamma}, \alpha \vdash \tau : \star}{D; \underline{\Gamma} \vdash \forall \alpha.\, \tau : \star} \qquad \frac{D; \underline{\Gamma} \vdash \widehat{\tau_{\uparrow}} : \widehat{\kappa} \qquad D; \underline{\Gamma} \vdash \underline{\tau}'}{D; \underline{\Gamma} \vdash \langle \widehat{\tau_{\uparrow}} \rangle \to \underline{\tau}' : \star}$$

$$\frac{\mathsf{arity}(D; \varepsilon) = \kappa \qquad D; \underline{\Gamma} \vdash \widehat{\tau_{\uparrow}} : \widehat{\kappa} \qquad D \vdash \underline{\Gamma}}{D; \underline{\Gamma} \vdash \varepsilon\, \widehat{\tau_{\uparrow}} : \star}$$

Figure 3: Well-formedness for the binary language

the bare type that are not projections of a constructor of the binary type should appear once and only once between double braces.

The type system thus defined is sound with respect to the dynamic semantics:

Theorem (Soundness). A closed, well-typed term is either a value, or reduces to a closed term of the same type.

As promised, the typing judgments can be projected down to the base language:

Theorem (Projection). Let D, $D^{\downarrow}$, $D^{\uparrow}$ be coherent constructor environments. Then, then if $\underline{M}$ has type $\underline{\tau}$ in context $\underline{\Gamma}$, the projections $[\![\underline{M}]\!]^{\downarrow}$ and $[\![\underline{M}]\!]^{\uparrow}$ have types (respectively) $[\![\underline{\tau}]\!]^{\downarrow}$ and $[\![\underline{\tau}]\!]^{\uparrow}$ in contexts $[\![\underline{\Gamma}]\!]^{\downarrow}$ and $[\![\underline{\Gamma}]\!]^{\uparrow}$.

6.7 Automatic lifting by elaboration of terms

The automatic lifting is based on the binary language: instead of directly generating an ornamented term, the lifting generates (at least conceptually) a well-typed binary term that can be projected to both the bare term and the ornamented term. This elaboration can itself be split into two parts: annotating the bare term with ornament types, and actual ornamentation.

To annotate the bare terms with ornament types, it is necessary to rewrite the typing rules for the binary language into *elaboration* rules that only mention the projection of a term instead of the binary term itself. The contexts remain binary: we need to know how each variable is ornamented. From these derivations we can extract a binary term—uniquely, modulo the ornamented code. We write the transformed term directly in the rules, but it is uniquely defined by the inference rules: the difficult part is in generating the rest of the derivation. The binary types are necessary to disambiguate

Typing $D; \underline{\Gamma} \vdash \underline{M} : \underline{\tau}$

$$\textsc{BOrn}\quad \frac{D^{\uparrow}; \llbracket \underline{\Gamma} \rrbracket^{\uparrow} \vdash M : \tau}{D; \underline{\Gamma} \vdash \{M\} : \{\tau\}}$$

$$\textsc{BVar}\quad \frac{\underline{\Gamma}(x) = \underline{\tau} \qquad D \vdash \underline{\Gamma}}{D; \underline{\Gamma} \vdash x : \underline{\tau}}$$

$$\textsc{BLam}\quad \frac{D; \underline{\Gamma}, \widehat{x_\uparrow} : \widehat{\tau_\uparrow} \vdash \underline{M} : \underline{\tau}'}{D; \underline{\Gamma} \vdash \lambda \widehat{x_\uparrow} : \widehat{\tau_\uparrow}.\, \underline{M} : \langle \widehat{\tau_\uparrow} \rangle \to \underline{\tau}'}$$

$$\textsc{BApp}\quad \frac{D; \underline{\Gamma} \vdash \underline{N} : \langle \widehat{\tau_\uparrow} \rangle \to \underline{\tau}' \qquad D; \underline{\Gamma} \vdash \widehat{M_\uparrow} : \widehat{\tau_\uparrow}}{D; \underline{\Gamma} \vdash \underline{N}\ \langle \widehat{M_\uparrow} \rangle : \underline{\tau}'}$$

$$\textsc{BTLam}\quad \frac{D; \underline{\Gamma}, \alpha \vdash \underline{M} : \underline{\tau}}{D; \underline{\Gamma} \vdash \Lambda\alpha.\, \underline{M} : \forall\alpha.\, \underline{\tau}}$$

$$\textsc{BTOrnLam}\quad \frac{D; \underline{\Gamma}, \{\alpha\} \vdash \underline{M} : \underline{\tau}}{D; \underline{\Gamma} \vdash \Lambda\{\alpha\}.\, \underline{M} : \forall\{\alpha\}.\, \underline{\tau}}$$

$$\textsc{BTApp}\quad \frac{D; \underline{\Gamma} \vdash \underline{M} : \forall\alpha.\, \underline{\tau} \qquad D; \underline{\Gamma} \vdash \underline{\tau}' : \star}{D; \underline{\Gamma} \vdash \underline{M}\ \underline{\tau}' : \tau[\alpha \leftarrow \underline{\tau}']}$$

$$\textsc{BTOrnApp}\quad \frac{D; \underline{\Gamma} \vdash \underline{M} : \forall\{\alpha\}.\, \underline{\tau} \qquad D; \underline{\Gamma} \vdash \{\tau'\} : \{\star\}}{D; \underline{\Gamma} \vdash \underline{M}\ \{\tau'\} : \tau[\{\alpha\} \leftarrow \{\tau'\}]}$$

$$\textsc{BCons}\quad \frac{\mathsf{tcon}(D; C) = \varepsilon \wedge \mathsf{conty}(D; C; \widehat{\tau_\uparrow}) = \widehat{\tau'_\uparrow} \qquad D; \underline{\Gamma} \vdash \widehat{M_\uparrow} : \widehat{\tau'_\uparrow}}{D; \underline{\Gamma} \vdash C(\widehat{\tau_\uparrow}; \widehat{M_\uparrow}) : \varepsilon\ \widehat{\tau_\uparrow}}$$

$$\textsc{BMatch}\quad \frac{D; \underline{\Gamma} \vdash \underline{M} : \varepsilon\ \widehat{\tau_\uparrow} \qquad \underline{\Gamma} \vdash \widehat{m_\downarrow} : \varepsilon\ \widehat{\tau_\uparrow} \to \underline{\tau}' \qquad \mathsf{valid}(\widehat{m_\downarrow})}{D; \underline{\Gamma} \vdash \mathsf{match}\ \underline{M}\ \mathsf{with}\ \widehat{m_\downarrow} : \underline{\tau}'}$$

$$\textsc{Fix}\quad \frac{D; \underline{\Gamma}, x : \underline{\tau} \vdash \underline{M} : \underline{\tau}}{D; \underline{\Gamma} \vdash \mathsf{fix}\, x{:}\underline{\tau}.\, \underline{M} : \underline{\tau}}$$

Typing of patterns $D; \underline{\Gamma} \vdash m_\downarrow : \underline{\tau} \to \underline{\tau}'$

$$\textsc{BPat'}\quad \frac{\mathsf{tcon}(D; C) = \varepsilon \wedge \mathsf{conty}(D; C; \widehat{\tau_\uparrow}) = \widehat{\tau'_\uparrow} \qquad D; \underline{\Gamma}, \widehat{x_\uparrow} : \widehat{\tau'_\uparrow} \vdash \underline{M} : \underline{\tau}''}{D; \underline{\Gamma} \vdash C(\widehat{x_\uparrow}) \to \underline{M} : \varepsilon\ \widehat{\tau_\uparrow} \to \underline{\tau}''}$$

$$\{\{\textsc{BPat}\}\}\quad \frac{\mathsf{tcon}(D^{\downarrow}; C) = \llbracket \varepsilon \rrbracket^{\downarrow} \wedge \mathsf{conty}(D^{\downarrow}; C; \llbracket \widehat{\tau_\uparrow} \rrbracket^{\downarrow}) = \widehat{\tau'} \qquad D^{\downarrow}; \llbracket \underline{\Gamma} \rrbracket^{\downarrow}, \widehat{x} : \widehat{\tau'} \vdash M : \llbracket \underline{\tau}'' \rrbracket^{\downarrow}}{D; \underline{\Gamma} \vdash \{\{C(\widehat{x}) \to M\}\} : \varepsilon\ \widehat{\tau_\uparrow} \to \underline{\tau}''}$$

Figure 4: Typing for the binary language

some constructs, for example how arguments should be grouped in an application or an abstraction. The new rules are not entirely syntax-directed: implicit type abstractions and applications, corresponding to code that only appears on the ornamented side, can be inserted everywhere.

To allow the insertion of new, ornamented code, we define a rule Hole that replaces the truly new parts of the code (*i.e.* the terms that would only appear in the ornamented code) by a special construct $\boxed{?}$ that indicates that user-supplied code should appear here. The full set of rules are given in Fig. 6.7.

Then, lifting algorithms are given by a type inference algorithm for the one-sided typing. In the current implementation, the expected type (given by the ornament specification) is propagated downwards in the term using bidirectional type inference. Some constraints are collected to instantiate the type variables. When a type can't be determined, or a term would normally need a type annotation in a bidirectional type system (*e.g.* lambda abstractions, constructors), the heuristic is to *default* to a non-ornamented version of the term (this assume that all the bare datatypes embed into the binary language).

This algorithm is largely incomplete. In particular, it doesn't handle well the definition of local functions. The inner functions should be lifted at types that can only be determined by looking at how they are used. A solution would be to allow to give an ornament specification for inner declarations via a patch. For a type system limited to prenex polymorphism (for instance in ML), it is likely that a complete type inference algorithm could be devised.

7 From syntactic ornaments to semantic ornaments

Our goal is to define an "ornamentation" relation between terms, that allows us to state that a term is an ornament of another and relate it to the original definition given in §2.2. In the two previous sections, we considered a syntactic view of ornaments: two termes are considered to be related by ornamentation if they are a projection of a single binary term. But the syntactic definition is incomplete and very restrictive: in particular, it is not stable by any reasonable program equivalence, and only relates program that have comparable runtimes. We define in this section a semantic definition of ornaments based on the syntactic definition, and relate closely this semantic definition to the original definition of ornaments.

7.1 Semantic ornament by contextual equivalence

Similarly to what could be done to define an equivalence of terms, we use (as in [13]) a definition of semantic ornamentation based on contextual equivalence. In essence, we want to consider two terms to be in ornamentation if they give similar results in related contexts. For a term equivalence, we would take equal contexts but the relation we want to define is heterogeneous. A first approach would be to consider contexts in syntactical ornament (extending the projection to contexts), but it is difficult to prove

Holes in the code:

$$\dfrac{\textsc{Hole}}{D;\underline{\Gamma} \vdash \varnothing : \{\tau\} \Rightarrow \{\boxed{\;?\;}\}}$$

Elaboration rules $D;\underline{\Gamma} \vdash M : \underline{\tau} \Rightarrow \underline{M}$

$$\textsc{OVar}\quad \frac{\underline{\Gamma}(x) = \underline{\tau} \qquad D \vdash \underline{\Gamma}}{D;\underline{\Gamma} \vdash x : \underline{\tau} \Rightarrow x}$$

$$\textsc{OAbs}\quad \frac{D;\underline{\Gamma}, \widehat{x}_{\uparrow} : \widehat{\tau}_{\uparrow} \vdash M : \underline{\tau}' \Rightarrow \underline{M}}{D;\underline{\Gamma} \vdash \lambda\llbracket \widehat{x}_{\uparrow} : \widehat{\tau}_{\uparrow}\rrbracket^{\downarrow}.\, M : \langle \widehat{\tau}_{\uparrow}\rangle \to \underline{\tau}' \Rightarrow \lambda\langle \widehat{x}_{\uparrow} : \widehat{\tau}_{\uparrow}\rangle.\, \underline{M}}$$

$$\textsc{OApp}\quad \frac{D;\underline{\Gamma} \vdash N : \langle \widehat{\tau}_{\uparrow}\rangle \to \underline{\tau}' \Rightarrow \underline{N} \qquad D;\underline{\Gamma} \vdash \widehat{M} : \widehat{\tau}_{\uparrow} \Rightarrow \widehat{M}_{\uparrow}}{D;\underline{\Gamma} \vdash N\ \widehat{M} : \underline{\tau}' \Rightarrow N\ \langle \widehat{M}_{\uparrow}\rangle}$$

$$\textsc{OTAbs}\quad \frac{D;\underline{\Gamma}, \alpha \vdash M : \underline{\tau} \Rightarrow \underline{M}}{D;\underline{\Gamma} \vdash \Lambda\alpha.\, M : \forall\alpha.\, \underline{\tau} \Rightarrow \Lambda\alpha.\, \underline{M}}$$

$$\textsc{OTAbsImplicit}\quad \frac{D;\underline{\Gamma}, \{\alpha\} \vdash M : \underline{\tau} \Rightarrow \underline{M}}{D;\underline{\Gamma} \vdash M : \forall\{\alpha\}.\, \underline{\tau} \Rightarrow \Lambda\{\alpha\}.\, \underline{M}}$$

$$\textsc{OTApp}\quad \frac{D;\underline{\Gamma} \vdash M : \forall\alpha.\, \underline{\tau} \Rightarrow \underline{M} \qquad D;\underline{\Gamma} \vdash \underline{\tau}' : \star \qquad \llbracket \underline{\tau}'\rrbracket^{\downarrow} = \tau'}{D;\underline{\Gamma} \vdash M\ \tau' : \tau[\alpha \leftarrow \underline{\tau}'] \Rightarrow \underline{M}\ \underline{\tau}'}$$

$$\textsc{OTAppImplicit}\quad \frac{D;\underline{\Gamma} \vdash M : \forall\{\alpha\}.\, \underline{\tau} \qquad D;\underline{\Gamma} \vdash \{\tau'\} : \{\star\}}{D;\underline{\Gamma} \vdash M : \tau[\{\alpha\} \leftarrow \{\tau'\}] \Rightarrow \underline{M}\ \{\tau'\}}$$

$$\textsc{OCons}\quad \frac{\mathsf{tcon}(D;C) = \varepsilon \land \mathsf{conty}(D;C;\widehat{\tau}_{\uparrow}) = \widehat{\tau}'_{\uparrow} \qquad D;\underline{\Gamma} \vdash \widehat{M} : \widehat{\tau}'_{\uparrow} \Rightarrow \widehat{M}_{\uparrow}}{D;\underline{\Gamma} \vdash \llbracket C\rrbracket^{\downarrow}(\widehat{\tau};\widehat{M}) : \varepsilon\ \widehat{\tau}_{\uparrow} \Rightarrow C(\widehat{\tau}_{\uparrow};\widehat{M}_{\uparrow})}$$

$$\textsc{OMatch}\quad \frac{D;\underline{\Gamma} \vdash M : \varepsilon\ \widehat{\tau}_{\uparrow} \Rightarrow \underline{M} \qquad \underline{\Gamma} \vdash \widehat{m} : \varepsilon\ \widehat{\tau}_{\uparrow} \to \underline{\tau}' \Rightarrow \widehat{m_{\downarrow}} \qquad \mathsf{valid}(\widehat{m_{\downarrow}})}{D;\underline{\Gamma} \vdash \mathsf{match}\ M\ \mathsf{with}\ \widehat{m} : \underline{\tau}' \Rightarrow \mathsf{match}\ \underline{M}\ \mathsf{with}\ \widehat{m_{\downarrow}}}$$

Elaboration of patterns $D;\underline{\Gamma} \vdash m : \underline{\tau} \to \underline{\tau}' \Rightarrow m_{\downarrow}$

$$\textsc{OPat}\quad \frac{\mathsf{tcon}(D;C) = \varepsilon \land \mathsf{conty}(D;C;\widehat{\tau}_{\uparrow}) = \widehat{\tau}'_{\uparrow} \qquad D;\underline{\Gamma}, \widehat{x}_{\uparrow} : \widehat{\tau}'_{\uparrow} \vdash \underline{M} : \underline{\tau}'' \Rightarrow \underline{M}}{D;\underline{\Gamma} \vdash \llbracket C\rrbracket^{\downarrow}(\llbracket \widehat{x}_{\uparrow}\rrbracket^{\downarrow}) \to M : \varepsilon\ \widehat{\tau}_{\uparrow} \to \underline{\tau}'' \Rightarrow C(\widehat{x}_{\uparrow}) \to \underline{M}}$$

$$\textsc{O}\{\{\textsc{Pat}\}\}\quad \frac{\mathsf{tcon}(D^{\downarrow};C) = \llbracket\varepsilon\rrbracket^{\downarrow} \land \mathsf{conty}(D^{\downarrow};C;\llbracket\widehat{\tau}_{\uparrow}\rrbracket^{\downarrow}) = \widehat{\tau'} \qquad D^{\downarrow};\llbracket\underline{\Gamma}\rrbracket^{\downarrow}, \widehat{x} : \widehat{\tau'} \vdash M : \llbracket\underline{\tau}''\rrbracket^{\downarrow}}{D;\underline{\Gamma} \vdash C(\widehat{x}) \to M : \varepsilon\ \widehat{\tau}_{\uparrow} \to \underline{\tau}'' \Rightarrow \{\{C(\widehat{x}) \to M\}\}}$$

Figure 5: Elaboration rules for lifting

many properties of the relation with this definition. Instead, taking inspiration from semantic approaches to typing [17], we take the largest relation on contexts for which all syntactically related terms are related. In essence, we form the *closure* of our syntactic relation by double orthogonalization.

In practice, we start by defining a syntactic ornamentation relation based on the binary language. Then, we define related contexts as contexts mapping terms in syntactic ornamentation to related results.

Definition (Syntactic ornamentation of terms). A term $M^{\uparrow}$ is an ornement of a term $M^{\downarrow}$ in (binary) context $\underline{\Gamma}$ at (binary) type τ, which we write $\underline{\Gamma} \vdash M \lesssim M' : \underline{\tau}$, if there exists a binary term $\underline{M}$ such that $\underline{\Gamma} \vdash \underline{M} : \underline{\tau}$ and both $\llbracket \underline{M} \rrbracket^{\downarrow}$ and $\llbracket \underline{M} \rrbracket^{\uparrow}$ are equal to $M^{\downarrow}$ and $M^{\uparrow}$.

Notice, by projection of typing $D; \underline{\Gamma} \vdash M^{\downarrow} \lesssim M^{\uparrow} : \underline{\tau}$ implies $D^{\eta}; \llbracket \underline{\Gamma} \rrbracket^{\eta} \vdash M^{\eta} : \llbracket \underline{\tau} \rrbracket^{\eta}$. That is, only related well-typed terms may be in an ornament relation.

From this relation on terms we define a relation on contexts. Let us start by defining a typing judgment for contexts of the base language: a context $\mathcal{E}$ has type $(\Gamma \vdash \tau) \rightsquigarrow (\Gamma' \vdash \tau')$ if for any term M such that $\Gamma \vdash M : \tau$, $\Gamma' \vdash \mathcal{E}[M] : \tau'$. Assume given a type `unit` with a single unary constructor `Unit`. We only consider the contexts returning a value of type `unit`. To simplify the notations, we write $\mathcal{E} : \Gamma \vdash \tau$ when $\mathcal{E} : (\Gamma \vdash \tau) \rightsquigarrow (\varnothing \vdash \text{unit})$. We define a relation $\leq_{\perp}$ between applications of a term to a context :

Definition. We say that $\mathcal{E}^{\downarrow}[M^{\downarrow}]$ terminates more than $\mathcal{E}^{\uparrow}[M^{\uparrow}]$, noted $\mathcal{E}^{\downarrow}[M^{\downarrow}] \leq_{\perp} \mathcal{E}^{\uparrow}[M^{\uparrow}]$, if when $\mathcal{E}^{\uparrow}[M^{\uparrow}]$ terminates, $\mathcal{E}^{\downarrow}[M^{\downarrow}]$ also terminates.

The relation $\leq_{\perp}$ can also be viewed as a relation between pairs of terms and pairs of contexts: $(\mathcal{E}^{\downarrow}, \mathcal{E}^{\uparrow}) \perp (M^{\downarrow}, M^{\uparrow})$ if and only if $\mathcal{E}^{\downarrow}[M^{\downarrow}] \leq_{\perp} \mathcal{E}^{\uparrow}[M^{\uparrow}]$. From the relation of syntactic ornamentation on terms, we can deduce, by orthogonality, a relation on contexts. We give here the expanded definition:

Definition (Ornamentation of contexts). Two contexts $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$ are said to be in ornament at type $\underline{\tau}$ in context $\underline{\Gamma}$ if $\mathcal{E}^{\eta} : \llbracket \underline{\Gamma} \rrbracket^{\eta} \vdash \llbracket \underline{\tau} \rrbracket^{\eta}$ and for all terms $M^{\downarrow}$, $M^{\uparrow}$, if $\underline{\Gamma} \vdash M^{\downarrow} \lesssim M^{\uparrow} : \underline{\tau}$, then $\mathcal{E}^{\downarrow}[M^{\downarrow}] \leq_{\perp} \mathcal{E}^{\uparrow}[M^{\uparrow}]$. We write this $\underline{\Gamma} \vdash \mathcal{E}^{\downarrow} \lessapprox \mathcal{E}^{\uparrow} : \underline{\tau}$.

In the definition of $\leq_{\perp}$, we require only that the bare term terminates more than the ornamented term. This is because, as noted in 6, when two terms are in syntactic ornament, the bare term terminates if the ornamented term terminates. The converse does not hold: the ornamented term can fail to terminate because of some code exclusive to the ornamented term, written between brackets in the binary term.

With one more step of orthogonalization, we recover the closure of the relation of syntactic ornamentation:

Definition (Semantic ornamentation of terms). Two terms $M^{\downarrow}$ and $M^{\uparrow}$ are said to be in ornament at type $\underline{\tau}$, context $\underline{\Gamma}$ if $\llbracket \underline{\Gamma} \rrbracket^{\eta} \vdash M^{\eta} : \llbracket \underline{\tau} \rrbracket^{\eta}$ and for all contexts $\mathcal{E}^{\downarrow}$, $\mathcal{E}^{\uparrow}$, if $\underline{\Gamma} \vdash \mathcal{E}^{\downarrow} \lessapprox \mathcal{E}^{\uparrow} : \underline{\tau}$, then $\mathcal{E}^{\downarrow}[M^{\downarrow}] \leq_{\perp} \mathcal{E}^{\uparrow}[M^{\uparrow}]$. We write this $\underline{\Gamma} \vdash M^{\downarrow} \lessapprox M^{\uparrow} : \underline{\tau}$.

In these definitions, we only consider termination in contexts that return unit. This does not limit the observations that can be made by the contexts: for example, if one wants to test whether two terms $M^{\downarrow}$ and $M^{\uparrow}$ return `true` in two contexts $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$, both contexts can be composed with the context $\mathsf{match}\ [\,]\ \mathsf{with}\ \text{true} \Rightarrow \text{unit} \mid \text{false} \Rightarrow \Omega$ where Ω is non-terminating. Then, returning `true` with the original contexts is equivalent to returning `unit` in the new contexts.

The relation thus defined enjoys some good properties (proved in §A.3):

Theorem (Adequacy). Terms in syntactic ornament relation are also in semantic ornament relation.

Moreover, it is indeed *semantic*: it is preserved by transformations on the bare or ornamented terms that preserve contextual equivalence. Our semantic relation is also *compatible*: each typing rule of the binary language translates to a deduction rule on the semantic ornament relation. For example, the rule for application (limited to unary application for simplicity) UnApp gives the rule Rel-UnApp below.

$$\textsc{UnApp}\ \frac{D;\underline{\Gamma} \vdash N : \langle \underline{\tau} \rangle \to \underline{\tau}' \qquad D;\underline{\Gamma} \vdash M : \underline{\tau}}{D;\underline{\Gamma} \vdash N\ \langle M \rangle : \underline{\tau}'} \qquad \textsc{Rel-UnApp}\ \frac{\underline{\Gamma} \vdash N^{\downarrow} \lessapprox N^{\uparrow} : \langle \underline{\tau} \rangle \to \underline{\tau}' \qquad \underline{\Gamma} \vdash M^{\downarrow} \lessapprox M^{\uparrow} : \underline{\tau}}{\underline{\Gamma} \vdash N^{\downarrow}\ M^{\downarrow} \lessapprox N^{\uparrow}\ M^{\uparrow} : \underline{\tau}'}$$

The definition behaves as expected. For example, it properly distinguishes the natural number $M^{\downarrow} = \mathtt{S(Z)}$ and the empty list $M^{\uparrow} = \mathtt{Nil}(\tau)$: consider the binary context $\underline{\mathcal{E}} = \mathsf{match}\ [\,]\ \mathsf{with}\ \mathtt{ZNil} \to \mathtt{Unit} \mid \mathtt{SCons}(x, y) \to \Omega$ where Ω is a non-terminating term. We have $\vdash \mathcal{E}^{\downarrow} \lessapprox \mathcal{E}^{\uparrow} : \text{natlist}\ \{\tau\}$, but $\mathcal{E}^{\downarrow}[M^{\downarrow}]$ diverges while $\mathcal{E}^{\uparrow}[M^{\uparrow}]$ terminates. On the other hand, terms that should be considered in ornament relation really are: for example, `rev_append` is considered an ornament of the non tail-recursive version of `add` (defined in §2.2): indeed, `add` is contextually equivalent to its tail-recursive version `add_bis` and `add_append` is a syntactic lifting of `add_bis`.

Finally, a problem of this definition is that it accepts a non-terminating term as an ornament of anything (see §A.3). This is not surprising, as even with syntactic lifting, it is possible to introduce non-terminating code wherever code exclusive to the ornamented version can appear.

7.2 First-order ornaments

Let us recall the original definition of first-order ornaments given in §2.2: ornaments of datatypes are defined by a projection function, and a function $f^{\uparrow}$ is an ornament of $f^{\downarrow}$ at ornament type $\tau_1 \to \tau_2 \to \cdots \to \tau$ if and only if $f^{\downarrow}\ (\mathsf{proj}_{\tau_1}\ x_1)\ \ldots\ (\mathsf{proj}_{\tau_n}\ x_n) = \mathsf{proj}_{\tau}\ (f^{\uparrow}\ x_1\ \ldots\ x_n)$ for all $x_i : [\![\tau_i]\!]^{\uparrow}$, where proj_{τ} is the projection function at type τ, projecting values of type $[\![\tau]\!]^{\uparrow}$ into values of type $[\![\tau]\!]^{\downarrow}$ related by ornamentation. We may recover a definition close to it, although we have to be careful about the termination: we change equality (that could be interpreted as contextual equivalence) with *contextual ordering* as defined below.

Definition (Contextual ordering). Let $M, M' : \tau$ be two (closed) terms. They are said to be contextually ordered, and we write $M <^{\downarrow}_{\tau} M'$, if for all $\mathcal{E} : (\Gamma \vdash \tau) \rightsquigarrow (\varnothing \vdash \mathtt{unit})$, if $\mathcal{E}[M']$ terminates then $\mathcal{E}[M]$ terminates.

7.2.1 Projection

The notion of projection function does not generalize to all higher-order ornaments. Indeed, functions for example cannot be projected in the general case: projecting the result may be possible, but the arguments would have to be unprojected, creating information from nothing. We thus need to identify a class of *projectible* ornaments for which it is possible to define a projection function.

A first interesting class of projectible ornaments is the class of *identity ornaments*: a type is lifted to itself and its values remain unchanged. To define identity ornaments, we assume that there is an embedding $|| - ||$ of the type and value constructors of the base language into the binary language. This embedding generalizes to all types and terms (see §A.4). Then, it respects a property of reflexivity: if M has type τ in a context Γ, $||M||$ has type $||\tau||$ in context $||\Gamma||$. The projection function for this class of ornaments is simply the identity. A second class of projectible ornaments is the class of *data ornaments*. They are datatypes whose arguments are identity ornaments, and whose constructors only have projectible ornaments as field. These types admit a projection function, defined more precisely in §A.4: the ornamented constructor can simply be changed to the bare constructor, all fields that only exist in the ornamented type are erased, and the other fields are projected or left unchanged. We thus define a class $\mathcal{I}$ of identity ornaments, including type variables because we make sure they are instantiated with identity ornaments, a class $\mathcal{D}$ of data ornament constructors, and the class $\mathcal{P}$ of projectible ornaments as the greatest fixed point of the following rules:

$$\begin{array}{c}\textsc{Lift}\\ ||\tau|| \in \mathcal{I}\end{array} \qquad \begin{array}{c}\textsc{Orn}\\ \{\tau\} \in \mathcal{I}\end{array} \qquad \begin{array}{c}\textsc{Invariant}\\ \dfrac{\tau_{\uparrow} \in \mathcal{I}}{\tau_{\uparrow} \in \mathcal{P}}\end{array}$$

$$\begin{array}{c}\textsc{Con}\\ \dfrac{\forall C, \widehat{\alpha}.\, \mathsf{tcon}(D; C) = \varepsilon \Rightarrow \mathsf{conty}(D; C; \widehat{\alpha}) \in \mathcal{P}}{\varepsilon \in \mathcal{D}}\end{array} \qquad \begin{array}{c}\textsc{Ornament}\\ \dfrac{\varepsilon \in \mathcal{D} \qquad \widehat{\tau_{\uparrow}} \in \mathcal{I}}{\varepsilon\ \widehat{\tau_{\uparrow}} \in \mathcal{P}}\end{array}$$

Then, the projection and ornamentation are linked by the following property. Its meaning is that, modulo termination, the two definitions of ornamentation by contextual equivalence and by projection functions agree on projectible ornaments.

Theorem (Projection and ornamentation).

1. Let $M^{\eta} : [\![\tau]\!]^{\eta}$ be closed terms such that $M^{\downarrow} <^{\perp}_{[\![\underline{\tau}]\!]^{\downarrow}} \mathsf{proj}_{\underline{\tau}}\, M^{\uparrow}$. Then $\varnothing \vdash M^{\downarrow} \lessapprox M^{\uparrow} : \underline{\tau}$.

2. Suppose $\varnothing \vdash M^{\downarrow} \lessapprox M^{\uparrow} : \underline{\tau}$. Then, $M^{\downarrow} <^{\perp}_{[\![\underline{\tau}]\!]^{\downarrow}} \mathsf{proj}_{\underline{\tau}}\, M^{\uparrow}$.

7.2.2 First order

We can now try to recover the original definition of functional ornaments. Consider two functions $f^{\downarrow}$ and $f^{\uparrow}$ that are semantic ornaments of each other. Then, one obtains related results by applying related arguments to them. In particular, we know that

$\mathsf{proj}_{\underline{\tau}}\ x$ and x are related. It is thus easy to see that functions in the semantic ornament relation are also in first-order ornament relation. The other side of the equivalence gives us more trouble: if we take $f^{\downarrow} = \bot$ and $f^{\uparrow} = \lambda x.\,\bot$, at ornament specification $\underline{\tau} \to \underline{\tau}'$ they are related by the first-order definition but not by semantic ornamentation. But if the definition of each function starts by as much abstractions as it takes arguments, the problem of termination disappears and the two definitions are indeed equivalent.

8 Discussion

8.1 Implementation

Our preliminary implementation of ornaments is based on a small, explicitly typed language. Once types are erased, it is a strict subset of OCaml: in particular, it does not feature modules, objects, *etc.*, but these are orthogonal to ornaments.

The lifting of ornaments does not depend on any type annotations: it is purely directed by the ornament specifications provided by the user. In our language with explicit types, ornaments have explicit type parameters, but they are only used to generate type annotations in the lifted code. Hence, our implementation could be used, with very few modifications, in a language with type inference such as OCaml or Haskell: we could ignore everything related to types and work directly on untyped terms, before running the host language type inferencer on the lifted terms. Another solution would be to run the type inference first to get explicitly typed terms (including types on ornament declarations), lift these terms, erase the types and run the host language type inferencer on the lifted functions.

The theory of ornaments assumes no side effects. However, as our implementation of lifting preserves the structure of functions, the ornamented code should largely behave as the bare code with respect to the order of computations. Still, we would have to be more careful not to duplicate or delete computations, which could be observed if side-effecting functions can be received as arguments. Of course, it would also be safer to have some effect type system to guard the programmer against indirect side-effecting performed by lifted functions—but this would already be very useful for bare programs.

8.2 Related works

Implicit arguments When the lifting process is partial, it returns code with holes that have to be filled manually. One direction for improvement is to add a post-processing pass to fill in some of the holes by using code inference techniques such as implicit parameters [3, 15], which could return three kinds of answers: a unique solution, a default solution, *i.e.* letting the user know that the solution is perhaps not unique, or failure. In fact, it seems that a very simple form of code inference might be pertinent in many cases, similar to Agda's instance arguments [7], which only inserts variable present in the context. However, code inference remains an orthogonal issue that should be studied on its own.

Colored Type Systems Another approach to the problem of code reuse in languages with dependent types is the notion of colored type system [2]. Compared to ornaments, the point of view is reversed: one can mark some parts of types with *colors* that can be erased to yield other types and functions.

This notion of erasure is backed by a specific type theory. The theory of colors provides properties about the erased functions that are similar to the coherence property given by ornaments, but does not define any form of lifting.

Polytypism & datatype-generic programming The presentation of ornaments given in this report is *orthogonal* to any form of polytypic or datatype-generic programming facility. We have chosen to study ornaments as a primitive object in order to focus on the practical, syntactic aspects of the formalism.

In a datatype-generic framework, ornaments can be coded through an indexed family, as demonstrated by the original presentation of ornaments [11]. In such a system, ornaments are thus first-class citizens that can be inspected or defined at run-time: besides datatype-genericity, we can also write ornament-generic programs. Being a primitive notion, the ornaments offered by our system do not support these techniques.

Compared to polytypic programming, ornaments offer a more fine-grained form of generic programming. Polytypic programming makes no proviso of the *recursive structure* of types: a polytypic program is defined at once over the entire grammar of types. With ornaments, we can single out a particular data-structure, ornament it into another datatype and take advantage of the structural ties when lifting functions.

8.3 Future work

The lifting algorithm uses rules inspired from bidirectional type inference to propagate the type information inside the term. Although this handles a good number of simple cases, an implementation of full ornament inference would make the lifting process more robust.

We have introduced a minimal language of patches to avoid having to manually edit the lifted function once the code has been generated. While it is sufficient for our small examples, users would probably benefit from more powerful patches that could, for example, transform function calls in many places in the code at once.

Another direction for improvement is to enable the definition of new ornaments by combination of existing ornaments of the same type. This would be particularly useful for GADTs: an indexed type could then be built from a bare type and a library of useful properties expressed as GADTs.

On the theoretical side, a possible improvement is to adapt the theory of higher-order ornament to a language closer to OCaml. A first step would be to switch to an implicitly typed base language, and then to add GADTs to the type system. Another useful improvement would be the addition of effectful operations, for example with references.

9 Conclusion

We have explored a non-intrusive extension of an ML-like language with ornaments. The description of ornaments by their projection seems quite convenient in most cases. Although our lifting algorithm is syntax-directed and thus largely incomplete, it seems to be rather predictable and intuitive, and it already covers a few interesting applications. In fact, incompleteness improves automation: by reducing the search space, much more code can be inferred before reaching a point where there are multiple choices. Moreover, these restrictions lead to quite natural results (*e.g.* in the lifting of `add` in §2.2). Still, it would be interesting to have a more semantic characterization of our restricted form of lifting.

Our results are promising, if still preliminary. This invites us to pursue the exploration of ornaments both on the practical and theoretical sides, but more experience is really needed before we can draw definite conclusions.

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

References

[1] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2), 2011.

[2] Jean-Philippe Bernardy and Moulin Guilhem. Type-theory in color. In *International Conference on Functional Programming*, pages 61–72, 2013.

[3] Pierre Chambard and Grégoire Henry. Experiments in generic programming: runtime type representation and implicit values. Presentation at the OCaml Users and Developers meeting, Copenhagen, Denmark, sep 2012.

[4] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[5] Pierre-Évariste Dagand and Conor McBride. A categorical treatment of ornaments. In *Logics in Computer Science*, 2013.

[6] Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 2014.

[7] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 143–155, 2011.

[8] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, pages 268–277, 1991.

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

[10] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley, 1981.

[11] Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*, 2014. To appear.

[12] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1998.

[13] Andrew Pitts. Typed operational reasoning. In Benjamin C Pierce, editor, *Advanced topics in types and programming languages.* MIT press, 2005.

[14] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Principles of Programming Languages*, pages 232–244, 2006.

[15] Scala. Implicit parameters. Scala documentation.

[16] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *International Conference on Functional Programming*, pages 341–352, 2009.

[17] Jérôme Vouillon and Paul-André Melliès. Semantic types: A fresh look at the ideal model for types. In *ACM SIGPLAN Notices*, volume 39, pages 52–63. ACM, 2004.

A Selected proofs

A.1 Projection of the evaluation

Lemma (Values are projected to values). If $\underline{v}$ is a value, $\llbracket\underline{v}\rrbracket^{\downarrow}$ and $\llbracket\underline{v}\rrbracket^{\uparrow}$ are values.

Proof. By induction on the structure of $\underline{v}$:

- If $\underline{v} = \lambda\langle\widehat{b_{\uparrow}}\rangle.\, \underline{M}$, given our syntactic restrictions on bindings, $\llbracket\widehat{b_{\uparrow}}\rrbracket^{\eta}$ is non-empty. Then, $\llbracket\underline{v}\rrbracket^{\eta} = \lambda\llbracket\widehat{b_{\uparrow}}\rrbracket^{\eta}.\, \llbracket M\rrbracket^{\eta}$ is a value.
- If $\underline{v} = \Lambda\{\alpha\}.\, \underline{v'}$, by induction hypothesis, $\llbracket\underline{v'}\rrbracket^{\eta}$ is a value. Thus, $\llbracket\underline{v}\rrbracket^{\downarrow} = \llbracket\underline{v'}\rrbracket^{\downarrow}$ and $\llbracket\underline{v}\rrbracket^{\uparrow} = \Lambda\alpha.\, \llbracket\underline{v'}\rrbracket^{\uparrow}$ are both values.
- If $\underline{v} = \Lambda\alpha.\, \underline{v'}$, by induction hypothesis $\llbracket\underline{v'}\rrbracket^{\eta}$ is a value. Thus, $\llbracket\underline{v}\rrbracket^{\eta} = \Lambda\alpha.\, \llbracket\underline{v'}\rrbracket^{\eta}$ is a value.
- If $\underline{v} = C(\widehat{\underline{\tau}}; \widehat{v'_{\uparrow}})$, by induction hypothesis, $\llbracket\widehat{v'_{\uparrow}}\rrbracket^{\eta}$ are values (apply the induction hypothesis on arguments that are not between braces, and the fact that w is a value if $\{w\}$ is a value). Thus, $\llbracket\underline{v}\rrbracket^{\eta} = C(\llbracket\widehat{\underline{\tau}}\rrbracket^{\eta}; \llbracket\widehat{v'_{\uparrow}}\rrbracket^{\eta})$ is a value.

□

Theorem. Let $\underline{M}$ and $\underline{M'}$ be binary terms such that $\underline{M} \longrightarrow_{n,p} \underline{M'}$. Then $\llbracket\underline{M}\rrbracket^{\downarrow} \longrightarrow^{n} \llbracket\underline{M'}\rrbracket^{\downarrow}$ and $\llbracket\underline{M}\rrbracket^{\uparrow} \longrightarrow^{n+p} \llbracket\underline{M'}\rrbracket^{\uparrow}$.

Proof. By structural induction on $\underline{M} \longrightarrow_{n,p} \underline{M'}$:

- If the last rule was Base, let's consider the base reduction:
  - If we reduced a pattern matching, a fixed-point, or an application of a type without brackets, it is easy to check that $\llbracket\underline{M}\rrbracket^{\eta} \longrightarrow \llbracket\underline{M'}\rrbracket^{\eta}$.
  - If we reduced a type application with brackets, the bare projections of $\underline{M}$ and $\underline{M'}$ are equal, and the ornamented case is easy to check.
  - For the case of the tuple application, the evaluation is translated into the n steps reduction:

$$\begin{aligned}
&(\lambda x_1 \ldots x_n.\, M)\; M_1\; \ldots\; M_n \\
&\quad\longrightarrow (\lambda x_2 \ldots x_n.\, M[x_1 \leftarrow M_1])\; M_2\; \ldots\; M_n \\
&\quad\longrightarrow \ldots \\
&\quad\longrightarrow M[x_1 \leftarrow M_1, \ldots, x_n \leftarrow M_n]
\end{aligned}$$

- If the last rule was Binary-Context, it is enough to check that the translation of an evaluation context is still an evaluation context (using the preservation of values by projection).

- If the last rule was Unary-Context, it is enough to check that the hole of the context is erased by translation (thus $[\![\underline{M}]\!]^\downarrow = [\![\underline{M'}]\!]^\downarrow$) and that the ornamented projection of a unary context is still an evaluation context.

□

Lemma. Suppose $\underline{M} \longrightarrow_{n+p} \underline{M'}$. Then, $n + p \geq 1$.

Proof. By structural induction. □

Theorem. Let $\underline{M}$ be a binary term. If $[\![\underline{M}]\!]^\uparrow$ terminates, then $[\![\underline{M}]\!]^\downarrow$ terminates.

Proof. We only need to prove that $\underline{M}$ terminates. If it did not terminate, then $[\![\underline{M}]\!]^\uparrow$ would not terminate since each reduction on $\underline{M}$ translates to at least one reduction on $[\![\underline{M}]\!]^\uparrow$. □

A.2 Projection of typing

Theorem. Let D, $D^\downarrow$, $D^\uparrow$ be coherent constructor environments. Then,

$$D; \Gamma \vdash M : \tau \Rightarrow D^\downarrow; [\![\Gamma]\!]^\downarrow \vdash [\![M]\!]^\downarrow : [\![\tau]\!]^\downarrow \wedge D^\uparrow; [\![\Gamma]\!]^\uparrow \vdash [\![M]\!]^\uparrow : [\![\tau]\!]^\uparrow$$

Proof. By structural induction on the derivation of $\underline{\Gamma} \vdash \underline{M} : \underline{\tau}$, constructing a translation of the binary rules to unary derivations on the projected terms. The rules BVar, BTLam, BTApp, BCons, BMatch, BFix translate directly to (respectively) Var, TLam, TApp, Cons, Match, Fix. The rules BTLam and BTApp are erased in the bare derivation and translated to TLam and TApp in the ornamented derivation. Each of the rules BApp and BLam translates to a number of nested applications of App and Lam, respectively. The rule BOrn needs only be translated to yield an ornamented derivation; the necessary derivation appears in its hypothesis.

Last, it is easy to check that the completeness and non-overlapping of clauses in pattern matching is preserved by the translation. □

A.3 Syntactic and semantic ornamentation

Theorem (Adequacy). Let $M^\downarrow$ and $M^\uparrow$ be two terms of the base language. If $\underline{\Gamma} \vdash M^\downarrow \lesssim M^\uparrow : \underline{\tau}$, then $\underline{\Gamma} \vdash M^\downarrow \lessapprox M^\uparrow : \underline{\tau}$.

Proof. Consider two contexts $\mathcal{E}^\downarrow$ and $\mathcal{E}^\uparrow$ such that $\underline{\Gamma} \vdash \mathcal{E}^\downarrow \lessapprox \mathcal{E}^\uparrow : \tau$. By definition of $\cdot \vdash \cdot \lessapprox \cdot : \cdot$ for contexts, and since $\underline{\Gamma} \vdash M^\downarrow \lesssim M^\uparrow : \underline{\tau}$, we have $\mathcal{E}^\downarrow[M^\downarrow] \leq_\perp \mathcal{E}^\uparrow[M^\uparrow]$. □

Lemma (Ornamentation and non-termination). Let $M^\downarrow$ and $M^\uparrow$ be two closed terms of the base language. If $\varnothing \vdash M^\eta : [\![\underline{\tau}]\!]^\eta$ and the evaluation of $M^\uparrow$ does not terminate, then $\varnothing \vdash M^\downarrow \lessapprox M^\uparrow : \underline{\tau}$.

Proof. Consider $\mathcal{E}^{\downarrow}$, $\mathcal{E}^{\uparrow}$ such that $\underline{\Gamma} \vdash \mathcal{E}^{\downarrow} \lessapprox \mathcal{E}^{\uparrow} : \underline{\tau}$. Suppose $\mathcal{E}^{\uparrow}[M^{\uparrow}]$ terminates. Then $\mathcal{E}^{\uparrow}[M'^{\uparrow}]$ terminates for all terms M', because $\mathcal{E}^{\uparrow}$ must not evaluate its hole. Consider the binary term $\underline{\Omega}_{\underline{\tau}} = \mathsf{fix}\, x : \underline{\tau}.\, x$. The evaluation of $\underline{\Omega}_{\underline{\tau}}$ does not terminate, neither does the evaluation of its projections. Since $\mathcal{E}^{\uparrow}[[\![\Omega_{\underline{\tau}}]\!]^{\uparrow}]$ terminates, $\mathcal{E}^{\downarrow}[[\![\Omega_{\underline{\tau}}]\!]^{\downarrow}]$ must terminate. By the same argument, $\mathcal{E}^{\downarrow}[M'^{\downarrow}]$ terminates for all terms M'. In particular, $\mathcal{E}^{\downarrow}[M^{\downarrow}]$ terminates. □

Compatibility asserts that all the rules of the binary type system translate to deduction rules on the relation $\underline{\Gamma} \vdash - \lessapprox - : \underline{\tau}$. As an example, we show how it is proved for the case of unary application. The proof is tedious but very mechanical. This style of proof generalizes to all the other rules.

Lemma (Compatibility for unary application). The following rule Rel-UnApp is admissible.

$$\frac{\underline{\Gamma} \vdash N^{\downarrow} \lessapprox N^{\uparrow} : \langle \underline{\tau} \rangle \to \underline{\tau}' \qquad \underline{\Gamma} \vdash M^{\downarrow} \lessapprox M^{\uparrow} : \underline{\tau}}{\underline{\Gamma} \vdash N^{\downarrow}\ M^{\downarrow} \lessapprox N^{\uparrow}\ M^{\uparrow} : \underline{\tau}'}\ \textsc{Rel-UnApp}$$

Proof. Suppose we have $\underline{\Gamma} \vdash N^{\downarrow} \lessapprox N^{\uparrow} : \langle \underline{\tau} \rangle \to \underline{\tau}'$ and $\underline{\Gamma} \vdash M^{\downarrow} \lessapprox M^{\uparrow} : \underline{\tau}$. We need to prove that for all contexts $\mathcal{E}^{\downarrow}$, $\mathcal{E}^{\uparrow}$ such that $\underline{\Gamma} \vdash \mathcal{E}^{\downarrow} \lessapprox \mathcal{E}^{\uparrow} : \underline{\tau}'$, $\mathcal{E}^{\downarrow}[N^{\downarrow}\ M^{\downarrow}] \leq_{\perp} \mathcal{E}^{\uparrow}[N^{\uparrow}\ M^{\uparrow}]$. Let us consider two such contexts. Consider $\mathcal{E}'^{\eta} = \mathcal{E}^{\eta}[M^{\eta}\ []]$. It is sufficient to prove that $\underline{\Gamma} \vdash \mathcal{E}'^{\downarrow} \lessapprox \mathcal{E}'^{\uparrow} : \underline{\tau}$: consider $M'^{\downarrow}$ and $M'^{\uparrow}$ such that $\underline{\Gamma} \vdash M'^{\downarrow} \lesssim M'^{\uparrow} : \underline{\tau}$. We need to prove $\mathcal{E}^{\downarrow}[N^{\downarrow}\ M'^{\downarrow}] \leq_{\perp} \mathcal{E}^{\uparrow}[N^{\uparrow}\ M'^{\uparrow}]$. This is true if we have $\underline{\Gamma} \vdash \mathcal{E}''^{\downarrow} \lessapprox \mathcal{E}''^{\uparrow} : \langle \underline{\tau} \rangle \to \underline{\tau}'$ with $\mathcal{E}''^{\eta} = \mathcal{E}^{\eta}[[]\ M'^{\eta}]$. Thus, we need to prove that for all $N'^{\downarrow}$ and $N'^{\uparrow}$ such that $\underline{\Gamma} \vdash N'^{\downarrow} \lesssim N'^{\uparrow} : \langle \underline{\tau} \rangle \to \underline{\tau}'$, we have $\mathcal{E}^{\downarrow}[N'^{\downarrow}\ M'^{\downarrow}] \leq_{\perp} \mathcal{E}^{\uparrow}[N'^{\uparrow}\ M'^{\uparrow}]$. But this is true because $\underline{\Gamma} \vdash N'^{\downarrow}\ M'^{\downarrow} \lesssim N'^{\uparrow}\ M'^{\uparrow} : \underline{\tau}'$ (from the typing rules) and $\underline{\Gamma} \vdash \mathcal{E}^{\downarrow} \lessapprox \mathcal{E}^{\uparrow} : \underline{\tau}'$. □

A.4 Projection and ornamentation

The embedding $\|-\|$ is required to be a section of the projection: for all type constructors ε and data constructors C, $[\![\|X\|]\!]^{\eta} = X$. The embedding extends naturally to types and terms:

$$\|\alpha\| = \alpha \qquad \|\tau_1 \to \tau_2\| = \langle \|\tau_1\| \rangle \to \|\tau_2\| \qquad \|\forall \alpha.\, \tau\| = \forall \alpha.\, \|\tau\| \qquad \|\varepsilon\ \widehat{\tau}\| = \|\varepsilon\|\ \|\widehat{\tau}\|$$

$$\|x\| = x \qquad \|\lambda x : \tau.\, M\| = \lambda \langle x : \|\tau\| \rangle.\, \|M\| \qquad \|M_1\ M_2\| = \|M_1\|\ \langle \|M_2\| \rangle$$

$$\|\Lambda \alpha.\, M\| = \Lambda \alpha.\, \|M\| \qquad \|M\ \tau\| = \|M\|\ \|\tau\| \qquad \|C(\widehat{\tau}; \widehat{M})\| = \|C\|(\|\widehat{\tau}\|; \|\widehat{M}\|)$$

$$\|\mathsf{match}\ M\ \mathsf{with}\ \widehat{m}\| = \mathsf{match}\ \|M\|\ \mathsf{with}\ \|\widehat{m}\| \qquad \|\mathsf{fix}\, x : \tau.\, M\| = \mathsf{fix}\, x : \|\tau\| \|M\|.$$

$$\|C(\widehat{x}) \to M\| = \|C\|(\widehat{x}) \to \|M\|$$

We first give a few helping lemmas:

Lemma (Section). Let X be a value v, a term M, a type τ, or a clause m. We have $[\![\|X\|]\!]^{\eta} = X$.

Lemma (Lifting of typing). Suppose $\Gamma \vdash M : \tau$. Then $||\Gamma|| \vdash ||M|| : ||\tau||$.

Lemma (Reflexivity). Suppose $\Gamma \vdash t : \tau$. Then, $||\Gamma|| \vdash t \lessapprox t : ||\tau||$.

Proof. Section is proved by structural induction.

Lifting of typing derivations is proved by induction on the derivation, translating the rules.

For reflexivity: the term $||t||$ projects to t on the bare and ornamented sides, and $||\Gamma|| \vdash ||t|| : ||\tau||$ by the previous lemma. □

Let's start by giving a detailed definition of the projection: a projection function is defined (in the base language) for each type constructor $\varepsilon \in \mathcal{O}$ as: $\mathsf{proj}_\varepsilon : \forall \widehat{\alpha_\uparrow}. [\![\varepsilon]\!]^\uparrow\ [\![\widehat{\alpha_\uparrow}]\!]^\uparrow \to [\![\varepsilon]\!]^\downarrow\ [\![\widehat{\alpha_\uparrow}]\!]^\downarrow$, with the following definition:

$$\begin{aligned}
&\mathsf{proj}_\varepsilon = \Lambda [\![\widehat{\alpha_\uparrow}]\!]^\uparrow.\ \lambda x{:}\varepsilon\ [\![\widehat{\alpha_\uparrow}]\!]^\uparrow.\ \mathsf{match}\ x\ \mathsf{with} \\
&\quad \mid \ldots \\
&\quad \mid [\![C]\!]^\uparrow([\![\widehat{x_\uparrow}]\!]^\uparrow) \to [\![C]\!]^\downarrow([\![\widehat{\alpha_\uparrow}]\!]^\downarrow; \mathsf{proj}_{[\![\mathsf{conty}(D;C;\widehat{\alpha_\uparrow})]\!]^\downarrow} [\![\widehat{x_\uparrow}]\!]^\downarrow) \\
&\quad \mid \ldots
\end{aligned}$$

For the types in $\mathcal{I}$, the projection function is $\mathsf{id} : \forall \alpha.\ \alpha \to \alpha$. For the other types in $\mathcal{D}$, $\mathsf{proj}_{\varepsilon\ \widehat{\alpha_\uparrow}}$ is a notation for $\mathsf{proj}_\varepsilon\ \widehat{\alpha_\uparrow}$.

We must prove the correction of the projection:

Theorem (Projection and ornamentation).

1. Let $M^\eta : [\![\tau]\!]^\eta$ be closed terms such that $M^\downarrow <^{\perp}_{[\![\underline{\tau}]\!]^\downarrow} \mathsf{proj}_{\underline{\tau}}\ M^\uparrow$. Then $\varnothing \vdash M^\downarrow \lessapprox M^\uparrow : \underline{\tau}$.

2. Suppose $\varnothing \vdash M^\downarrow \lessapprox M^\uparrow : \underline{\tau}$. Then, $M^\downarrow <^{\perp}_{[\![\underline{\tau}]\!]^\downarrow} \mathsf{proj}_{\underline{\tau}}\ M^\uparrow$.

Proof. We will start by showing that we only need to prove the theorem for values.

1. If $M^\uparrow$ does not terminate, the conclusion is obviously true. Suppose $M^\uparrow$ does evaluate to a value $v^\uparrow$. Then $M^\downarrow$ evaluates to a value $v^\downarrow$. The terms $v^\downarrow$ and $M^\downarrow$ are contextually equivalent, and the same is true for $v^\uparrow$ and $M^\uparrow$: it is enough to prove the theorem for $v^\downarrow$ and $v^\uparrow$.

2. If $M^\uparrow$ does not terminate, the conclusion is obviously true. Suppose $M^\uparrow$ evaluates to a value $v^\uparrow$. Consider the binary context $\underline{\mathcal{E}} = (\lambda\langle x : \tau\rangle.\mathtt{Unit})[]$. $M^\uparrow$ terminates in this context, so $M^\downarrow$ must terminate too. This proves that $M^\downarrow$ evaluates to a value.

Then, the theorem is proved for values by induction on values and using reflexivity to discharge the cases of the types in $\mathcal{I}$. □