Ornaments in Practice

Thomas Williams Pierre-Évariste Dagand Didier Rémy

INRIA

{Thomas.Williams,Pierre-Evariste.Dagand,Didier.Remy}@inria.fr

Abstract

Ornaments have been introduced as a way to describe some changes in datatype definitions that preserve their recursive structure, reorganizing, adding, or dropping some pieces of data. After a new data structure has been described as an ornament of older one, some functions operating on the bare structure can be partially or sometimes totally lifted into functions operating on the ornamented structure. We explore the feasibility and the interest of using ornaments in practice by applying these notions in an ML-like programming language. We propose a concrete syntax for defining ornaments of datatypes and the lifting of bare functions to their ornamented counterparts, describe the lifting process, and present several interesting use cases of ornaments.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Data types and structures

General Terms Design; Languages; Experimentation

Keywords Ornament; Datatypes; Code inference; Refactoring; Dependent types; Generalized Algebraic Datatypes

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 universal 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 (Knuth 1981) in functional settings (Okasaki 1998; Hinze 1998). 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) (Cheney and Hinze 2003; Schrijvers et al. 2009; Pottier and Régis-Gianas 2006) or refinement types (Freeman and Pfenning 1991; Bengtson et al. 2011). 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 (McBride 2014) 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

WGP'14, August 31, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2633628.2633631

paper 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 fourfold: 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; finally, we have identified several interesting issues related to the implementation of ornaments that need to be investigated in future works (§5).

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
```

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 (Dagand and McBride 2014). 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. (Bernardy and Guilhem 2013). 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 ornament, 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 (Dagand and McBride 2013). 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

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

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 (Dagand and McBride 2013), this paper aims 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 $p \rightarrow e$ 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 (McBride 2014), but following the categorical model (Dagand and McBride 2013), we allow the recursive arguments to be reordered.

This rules out all the following functions in the definition 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 *inserting* a field asking for an element of the empty set.

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 usual syntactic conditions.

2.2 Lifting functions: syntax and automation

Using the ornament projection, we can also relate a lifted function operating on some ornamented types with the corresponding function operating on their respective bare types. Intuitively, such a *coherence* property states that the results of the ornamented function are partially determined by the result 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 $\sigma \rightarrow \tau$ to the type $\sigma^\dagger \rightarrow \tau^\dagger$ using the ornaments. More precisely, suppose we want this lifting to use the ornaments defined by the projections $u_\sigma : \sigma^\dagger \rightarrow \sigma$ and $u_\tau : \tau^\dagger \rightarrow \tau$. 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 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:

```
length (f† xs ys) = 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. 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 very 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 → a1
  | Cons(x, ml') → a2
```

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([?], 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: we conjecture that if the projection has a constant cost per recursive call, then the (asymptotic) 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. To stay close to the intended meaning of the function, the automatic lifting will not call the projection functions defining the ornaments in the generated code.

Open question: In this section, we have shown how our prototype exploits the syntactic structure of the bare function to generate coherent liftings. While our heuristics seem "reasonable", we lack a formal understanding of what "reasonable transformations" are. In particular, parametricity falls short of providing such a mechanism.

2.3 Patching the generated code

When the lifting leaves a hole in the code because some part of it can't be automatically lifted, we could rely on a post-processing code inference phase to fill in the missing parts, as we have already mentioned. This may still fail—or make a wrong choice because of heuristics. In this case, the user can play the post-processor and edit the lifted code by hand. 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 took 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 case, we expect the user will 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: Surprisingly, even if the theory of ornaments remains first-order, our syntactic lifting extends seamlessly to higher-order functions. 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 function on maps map_exists with 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.

Open question: While higher-order liftings work in practice, their theory is not well understood yet: what exactly should the coherence property for higher-order lifting be?

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.

Open question: In order to fully automate the refactoring tasks, we crucially rely on the good behavior of the ornament under inversion. However, we cannot hope to give a *complete* syntactic criterion for such a class of ornaments. We still have to devise a syntactic presentation that would delineate a sufficiently expressive subclass of reversible ornaments while being intuitive.

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(e1, e2) → LAdd(to_lexpr e1, to_lexpr e2)
  | 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 $(n,\ \alpha)$ `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 contaits 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. Discussion

5.1 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.

5.2 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 (Chambard and Henry 2012; Scala), 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 (Devriese and Piessens 2011), 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 dependently-typed languages is the notion of colored type system (Bernardy and Guilhem 2013). 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 article 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 (McBride 2014). 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.

5.3 Future work

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.

Also, even if we are able to generate useful liftings for a number of higher-order functions such as `exists` and `filter`, we are still missing a theory of higher-order ornaments that would explain the expected behavior of these liftings.

6. 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

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems*, 33(2), 2011. doi: 10.1145/1890028.1890031.

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

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

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

P.-É. Dagand and C. McBride. A categorical treatment of ornaments. In *Logics in Computer Science*, 2013. doi: 10.1109/LICS.2013.60.

P.-É. Dagand and C. McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 2014. doi: 10.1017/S0956796814000069.

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

T. Freeman and F. Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, pages 268–277, 1991. doi: 10.1145/113445.113468.

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

D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. ISBN 0-201-03822-6.

C. McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*, 2014. To appear.

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

F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Principles of Programming Languages*, pages 232–244, 2006. doi: 10.1145/1111037.1111058.

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

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *International Conference on Functional Programming*, pages 341–352, 2009. doi: 10.1145/1596550.1596599.