Disornamentation

Lucas Baudin[1] and Didier Rémy[2]

[1]École Normale Supérieure, [2]Inria

Abstract

Ornaments, which have recently been put in the spotlight, are a way to describe changes in datatype definitions, reorganizing, adding, or dropping some pieces of data. Ornamentation is the process of translating code operating on the original datatype to code operating on the new one. A formalization and an implementation of ornaments has been proposed for the ML family of languages. Our work focuses on the opposite transformation, called *disornamentation*. We generalize the ornamentation framework developed for ML and based on a posteriori abstraction so that both ornamentation and disornamentation become instances of this framework, allowing more expressive relational transformations of datatypes. We adapt the ornamentation prototype to support such bidirectional transformations and use it to present several typical examples using disornamentation or a combination of ornamentation and disornamentation.

1 Contributions

Disornamentation. We propose a formalization and an implementation of a mechanism to automatically adapt code along disornaments, which are relations opposite to ornaments.

Mixed transformations. We actually generalize the ornamentation framework for ML, so that both ornamentation and disornamentation become instances of it, allowing more expressive relational transformations of datatypes, also including addition and removal of constructors.

A robust patch language. The previous prototype that dealt with ornaments only allowed to fill holes using a number, which could change when the code was modified. We propose a more robust way to specify patches, using term patterns to select holes. This also benefits to ornaments and helps go back and forth between ornamented and disornamented code.

Examples. Several examples that demonstrate these contributions are available online[1], including: disornamentation of red-black trees to remove the balancing information (shown below); an example of synchronization between code operating on bare expression and on expressions ornamented with location information; an illustration of how a new, unrelated constructor can be added to a datatype. A JavaScript interface[2] is also available to experiment with the prototype and code synchronization in particular.

[1]http://gallium.inria.fr/~remy/disornamentation/
[2]http://www.eleves.ens.fr/home/lbaudin/demo/

2 Ornaments

In the ML family of languages, datatypes are inductively defined as labeled sums and products over other datatypes. In this context, ornaments are relations that describe changes in datatype definitions reorganizing, adding, or dropping some pieces of data [Dagand and McBride, 2014, Ko and Gibbons, 2016]. They can be used to partially and sometimes totally lift ML values operating on the bare definition into values operating on the ornamented structure. This process is called ornamentation.

These transformations are useful to do various refactoring operations: adding new arguments to constructors, automatically translating code via an isomorphism or (more surprisingly) removing constructors.

Williams and Rémy [2018] have proposed a formalization and a prototype to perform ornamentation. Ornaments can be described by writing relations between values of the source and target datatypes. For instance, we can write a relation between the datatype which represents peano numbers and the list datatype

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

type relation α natlist: nat ⇒ α list with
  | Z ⇒ Nil
  | S n ⇒ Cons(_, n) when n : α natlist
```

We use the keyword `relation` instead of `ornament` to include both ornamentation and disornamentation. Values are related using patterns, for instance `Z` and `Nil` are related. Values of the form `S` n_- are related with values of the form `Cons(_,` n_+`)` when n_- and n_+ are themselves related by α `natlist`.

Then, values can be lifted along ornaments. For instance, an `add` function can be lifted to a `append` function between lists.

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

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

which results in the following incomplete term:

```
let rec append n m = match n with
  | Nil → m
  | Cons(_, n) → Cons(#3, append n m)
```

Indeed, it contains a hole `#3` whose content must be specified by the user, using a patch in the lifting declaration:

```
let append = lifting add
  : _ natlist ⇒ _ natlist ⇒ _ natlist
  patch #3[match n with Cons(a, _) → a]
```

This will fill in the hole with the corresponding code, then simplify the code, and return the append function as expected:

```
  | Nil → m
  | Cons(a, n) → Cons(a, append n m)
```

Williams and Rémy proved the lifting process correct using a step-indexed logical relation to establish a strong correspondence between the original and lifted codes. Other examples of ornaments can be found online[2].

3 Disornaments

Disornaments describe relations between a source and a target datatype that contains less information. They are the reverse of ornaments. Like ornamentation, disornamentation is the process of adapting terms along disornaments. Disornamentation is therefore the opposite transformation to ornamentation. As ornamentation, it is a way to perform some code refactoring, adapting existing code to new datatypes.

Code synchronization A typical use of ornaments is the addition of locations to abstract syntax trees. For instance, we may have written an evaluator for some language of expressions. In order to provide meaningful error messages, we may later add location information, by changing the expression type to a tuple that also contains locations (and this recursively), i.e., changing the type definition from

```
type expr = App of expr * expr | ...
```

to

```
type expr' = App of expr * expr | ...
and expr = expr' * location
```

Writing an ornament from the first type to the second one is straightforward. However, if we wish to modify the lifted code (e.g. add `let` constructs), we may prefer to do this first on the original code (where editing is easier) if the modification does not involve locations (and only on the lifted code otherwise). Disornamentation makes it possible to go back and forth between those two views of the code. The whole example is available online.

More generally, having two views of the same code, one operating on the base datatype and the other operating on the ornamented datatype, would be useful in many cases. For instance, it could be used to keep a version of the code that is easier to read and modify as long as the ornamented part is not involved, while using it is still possible in the ornamented code.

When performing ornamentation, every construct of the source term can be mapped to a construct in the resulting term. Indeed, values are extended and more code is needed to create and manipulate the new pieces of data. On the opposite, disornamentation typically removes parts of the source data structure, and therefore the code used to compute pieces of data that have been removed becomes useless and may be garbage-collected.

For disornamentation to have good properties (such as being the identity transformation with the identity ornament), useless code elimination must only be performed on code that was already useless before the transformation. Notice that such an elimination has no effect after ornamentation, which does not introduce useless code.

Conversely, the new version of the code may miss pieces of data that were obtained by pattern-matching on the richer data structure. When this information is still needed to build values of the disornamented structure, the lifted code will contain holes to be filled with user-provided patches.

A robust patch language

Code synchronization makes it possible to automatically disornament code that was previously obtained by lifting through the inverse ornamentation relation. For this purpose, it is a key to have a robust patch language that uses term patterns rather than numbers to select holes (and capture variables that can be used in the patch content). For instance, the patch of the append function can now be written:

```
patch match _ with Cons(a, _) → Cons(#[a], _)
```

The content of the patch, a, which appears between `#[` and `]` uses a variable captured in the term pattern `match _ with Cons(_, _) → Cons(#, _)`. The patch applies wherever the term pattern appears in the code. The term pattern is often a small suffix of the path from the root to the hole: it should be short enough to also apply in similar situations, but long enough to avoid applying at other undesired occurrences.

This new language is robust in the sense that it allows to write patches that usually need not be changed when parts of the program not directly related to the code being patched are (moderately) modified, and similar patches may easily factorize, including new cases that may appear when the code is extended. An example of a patch that fills two holes is given in the next section.

When a term is disornamented, it may be useful to reornament it afterwards, to recover the original term: this is the basis of code synchronization. In order to do so, we automatically generate patches during disornamentation that will be used for reornamentation to fill the holes corresponding to the code removed by disornamentation. Exact patches are first generated by comparing the incomplete, reornamented code with a normalized version of the ornamented code. Patches are then *minimized* by sharing them whenever possible and reducing the total size of term-patterns without changing the matching occurrences and keeping the size of each term-pattern above a threshold to reduce the risk of accidental capture in future changes.

Disornamentation needing patches Up to now, we have only considered the disornamentation of previously ornamented code. In this setting, it is expected that disornamentation can be performed without further user input because disornamented code does not depend on any ornamented part. However, disornamentation can also be performed directly on hand-written code in which a part removed from the datatype may be used to compute another part, still present in the disornamented datatype. We illustrate several features of disornamentation on red-black trees:

```
type redblack =
  | Black of redblack ∗ elt ∗ redblack
  | Red of redblack ∗ elt ∗ redblack
  | Empty

let add x s =
  let rec add_aux t = match t with
    | Empty → Red(Empty, x, Empty)
    | Red(l, y, r) →
        begin match compare x y with
        | Lt → Red(add_aux l, y, r)
        | Gt → Red(l, y, add_aux r)
        | Eq → t end
    | Black(l, y, r)  →
        begin match compare x y with
        | Lt → balance_l (add_aux l) y r
        | Gt → balance_r l y (add_aux r)
        | Eq → t end in
  match add_aux s with
    | Red(a, b, c) → Black(a, b, c)
    | a → a
```

Balancing information (i.e. whether a node is red or black) can be removed, for instance to have a simplified view of the algorithm.

```
type tree =
  | SEmpty
  | SNode of tree ∗ elt ∗ tree

type relation simplify: redblack ⇒ tree with
  | Empty ⇒ SEmpty
  | (Red(a, e, b) | Black(a, e, b))
      ⇒ SNode(a, e, b) when a b: simplify
```

However, the removed balancing information, which was used to decide whether a new element had to be added on the left or on the right of the currently selected node, is now missing in the disornamented code, and replaced by holes (overlayed, and not to be confounded with code ellipses "`begin ... end`"):

```
let simple_add =
  lifting add: elt → simplify → simplify

let simple_add x s =
  let rec add_aux t = match t with
    | SEmpty → SNode(SEmpty, x, SEmpty)
    | SNode(l, y, r) →
      begin match #38 with
        | Left _ → begin ... end
        | Right _ → begin ... end end in
  let a = add_aux s in
  match a with
   | SEmpty → a
   | SNode(a, e, b) →
     begin match #8 with
       | Left _ → begin ... end
       | Right _ → begin ... end end
```

More precisely, these holes are caused by the encoding of the disjunctive pattern in the definition of the `simplify` relation: the user has to specify which of the two alternatives must be taken when a `SNode` is pattern-matched. While in the original code there is a branch for `Red` and a branch for `Black`, a value of the α `sum` type that contains two constructors, `Left` and `Right`, must be specified to choose the branch. We may fill in the holes using the robust patch language described above, for example by always inserting as if it were a `Red` node:

```
let simple_add =
  lifting add : elt → simplify → simplify with
  patch match #[Left ()] with Left _ → _
```

Here, the patch is generic enough to be used for both holes `#38` and `#8`. It selects the first branch every time there is a choice to make. More details about this patch language can be found online. After the patch is applied, the code is *automatically* simplified, returning:

```
let simple_add x s =
  let rec add_aux t = match t with
    | SEmpty → SNode(SEmpty, x, SEmpty)
    | SNode(l, y, r) →
      begin match compare x y with
        | Lt → SNode(add_aux l, y, r)
        | Gt → SNode(l, y, add_aux r)
        | Eq → t
      end in
  let a = add_aux s in
  match a with
    | SEmpty → a
    | SNode(a, e, b) → SNode(a, e, b)
```

Notice that the last three lines could easily be further simplified to a, but this η-contraction is not currently automatically performed. In fact, performing the η-contraction would be a problem for reornamentation because ornamentation does not automatically perform η-expansion, for good reasons. If disornamentation were to perform η-contraction, then one would need to add a mechanism for manually requesting some η-expansions before ornamentation.

Interestingly, the framework previously introduced to perform ornamentation alone in ML [Williams and Rémy, 2018] is also well suited for disornamentation: both theory (including the proof using the step-indexed logical relation) and implementation can be largely reused for disornamentation, and only require some small adaptation.

4 Formalization

We now sketch the formalization of ornaments and stress out what needs to be generalized to handle disornamentation.

Ornamentation Ornaments are encoded using a skeleton type and two functions, a *projection* and an *injection*. The skeleton type is an "open version" of the original datatype, i.e., whose constructors are abstracted over the types of their arguments (see [Williams and Rémy, 2018] for details). The *projection* function maps the target datatype to the skeleton type while the *injection* function takes an element of the skeleton type and an extra argument (whose type depends on its first argument) to generate an element of the target datatype.

Ornamenting an ML definition is performed in two steps. First, a generic term is elaborated independently of any ornament. This term is parametrized by one or several ornaments (each one described by a tuple composed of the target type, the skeleton type, and the projection and injection functions). Basically, every pattern-matching construct is replaced by a call to the projection function and a pattern matching on the corresponding skeleton while every constructor is replaced by a call to the injection function (leaving a hole for the extra argument). In a second step, this generic term can be instantiated with specific ornaments. The term is then reduced and simplified so that the skeleton type disappears and only the target type remains. At every call site of the injection function, if the extra argument does not belong to the unit type, the resulting term has holes which have to be filled using user-provided information given as *patches*.

Generalization for disornaments We now describe extension of this framework to cover disornamentation. To avoid any confusion, disornamentation is considered as a transformation from an ornamented type to a disornamented type. As with ornamentation, the skeleton type is an open version of the source (*i.e.* the ornamented) type. We are going to describe functions analogous to the projection and injection function of ornaments. For the sake of clarity, we keep those names even if they are neither projection nor injection anymore. That is, the projection operates from the disornamented datatype towards the skeleton type while the injection operates from the skeleton type to the ornamented datatype. The projection now needs additional data, provided as an extra argument, similar to the one given to the injection function in the ornamentation case. On the opposite, the injection function does not need this extra argument anymore. Therefore, to cover both ornamentation and disornamentation simultaneously, projection and injection functions each take an extra argument which contains data that is needed to construct an element of the skeleton type (for disornamentation) or an element of the target type (for ornamentation), and becomes more symmetrical.

The elaboration to a generic term needs to be updated accordingly. The rest of the transformation proceeds as with ornamentation.

Implementation We implemented disornamentation in the existing prototype that generates ML code from lifting declarations and which was initially written to handle ornamentation alone. The relations describing ornaments restricted the left patterns to have no wildcards and no disjunctive patterns. To allow disornamentation, we dropped those restrictions and allowed the same patterns on both side. Therefore, for every ornament, the associated disornament can be written in the same language, roughly swapping patterns to get the inverse relation. We also extended projection and injection functions as previously described, and modified the elaboration of the generic term accordingly. Then, the existing mechanisms to perform simplifications (which are crucial to get code that is as close as possible to the code that the user would have manually written) could be reused. We only had to add a step to remove useless code, as describe above, which was not needed for ornamentation alone.

Interestingly, this generalization allows describing transformations that are neither ornamentation nor disornamentation but mixtures thereof. We also get for free the addition of a new unrelated constructor to an existing datatype (as a dual of constructor removal, which we previously had as a pathological case of ornamentation)—a feature already recognized as desirable by Najd and Peyton-Jones [2016].

5 Conclusions

We have shown that disornamentation, the dual of ornamentation, can be easily added to, and smoothly mixed with, ornamentation, allowing both transformations to be composed in sequence or performed simultaneously; they complement one another, allowing a production version of the code to be kept in sync with a simplified version where non essential details have been removed.

We have only tried small examples in our current prototype, but hopefully, disornamentation could also be added to an ongoing implementation of a more ambitious prototype based on a subset of OCaml and tested on real world programs.

While ornamentation and disornamentation are based on transforming datatype definitions, we should also be able to remove some arguments of functions with disornamentation by (virtually) boxing all arguments, in a similar way we can add some with ornamentation.

Disornamentation is another example of code transformation based on “a posteriori code generalization” a concept introduced for ornamentation by Williams and Rémy which allows code reuse without the heavy boilerplate of abstracting over all possible futures.

References

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

H.-S. Ko and J. Gibbons. Programming with ornaments. *Journal of Functional Programming*, 27, 2016. doi: 10.1017/S0956796816000307.

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

T. Williams and D. Rémy. A Principled Approach to Ornamentation in ML. *Proceedings of the ACM on Programming Languages*, pages 1–30, Jan. 2018. doi: 10.1145/3158109. URL https://hal.inria.fr/hal-01666104.