

Putting Ornaments to work (master's intership proposal)

Pierre Dagand and Didier Rémy

December 13, 2013

Introduction to ornaments

Inductive datatypes and parametric polymorphism were two key new features introduced in the ML family of languages in the 80's. Datatypes stress the algebraic structure of data while parametric polymorphism allows to exploit universal properties of algorithms working on algebraic structures.

Datatype definitions are inductively defined as sum and product combinations of primitive types. It is thus not surprising that datatypes definitions often look similar to one another. The user has sometimes the feeling of repeatedly programming the same operations with only minor variations and of stealing the code from other, similar definitions. One is left wondering whether this cut and paste could not be automated.

The recent theory of ornaments aims at answering this question. 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. Once a close correspondence between a datatype and its ornament has been established, the functions that operate only on the structure on the original datatype can be semi-automatically lifted to its ornaments.

For example, lists can be described as the ornament of the Church encoding of natural numbers as follows:

```
type nat = Zero | Succ of nat

type 'a list extends nat =
  | Nil extends Zero
  | Cons of 'a * 'a list extends Succ
```

Indeed, lists are nothing but a copy of natural numbers with, after renaming of data constructors, some additional data of type 'a attached to the Cons constructors. Similarly, the option type can be defined as an ornament of booleans:

```
type 'a option extends bool =
| None extends false
| Some of 'a extends true
```

Now, from the comparison function `gt` (greater than) on natural numbers

```
let rec gt (m : nat) (n : nat) : bool =
  match m with
  | Zero -> false
  | Succ m' ->
    match n with
    | Zero -> true
    | Succ n -> gt m' n
```

we ought to be able to *automatically* obtain the implementation of the function `nth` that accesses the n -th element of a list:

```
let rec nth (m : 'a list) (n : nat) : 'a option =
  match m with
  | Nil -> None
  | Cons (x, m') ->
    match n with
    | Zero -> Some x
    | Succ n -> nth m' n
```

Indeed, `nth` behaves on the ornamented types as `gt` behaves on numbers.

Description of the internship

While the theory of ornaments is relatively well-understood, turning the theory into practice in a real language remains to be done. This is the goal of the internship.

This requires extending the core language with a notation for specifying ornaments and with a mechanism by which functions can be lifted onto their ornamented version, giving the programmer the ability to specify some extra-information if required by the ornament.

The code of `nth` above is completely determined by the ornament and the comparison function. However, in general, the code of lifted operations might only be partially determined. The user must then explicitly specify the behavior of the lifted function on the data introduced by the ornament. This raises several design issues that haven't been explored yet in the theory of ornaments. Recovering inductive definitions—on which the theory of ornament is built—from the recursive definitions that are more common when programming is another research topic.

Although the core of the work is on language design, it also touches several related areas. The theory of ornaments is done in the setting of dependently-typed languages, using tools from category theory. In the internship, we will however restrict ourselves to the simpler setting of non-dependent types—at

least to start with—using F-omega as our core language. Knowledge of category theory is thus not necessary—but it may help to understand the related literature. On the more practical side, the student should also prototype his design. In fact, the implementation of ornaments can be seen as a high-level compilation from the host language extended with ornaments into the lower-level host language without ornaments. This will very likely raise optimization issues.

Parts of the work may also be more experimental. For instance, searching in the existing code base for potential and useful applications of ornaments. Or developing a small killer-application to demonstrate the benefit and power of ornaments.

Ornaments may also be useful for code refactoring, when the new data-structure can be described as an ornament of the older one. When the representations are isomorphic, one may expect the transformation of operations between the two representations to be fully automated.

Hopefully, this internship should confirm the conjecture that ornaments are a quite useful construct in programming languages with datatypes and polymorphism, which will then open the door to many future research directions.

Contact

The work will be conducted at INRIA Rocquencourt in the Gallium team under the supervision of Pierre Dagand and Didier Rémy. There may also be a possibility for working part time in the Paris sub-office of INRIA located at Place d'Italie.

For more information, please contact `Pierre-Evariste.Dagand@inria.fr` or `Didier.Remy@inria.fr`.

References

- Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*, 2013. URL <http://personal.cis.strath.ac.uk/~conor/pub/OAA0/LitOrn.pdf>. To appear.
- Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of Functional Programming*, In press. URL <http://gallium.inria.fr/~pdagand/stuffs/journal-2013-patch-jfp/paper.pdf>.
- Pierre-Evariste Dagand. *Reusability and Dependent Types*. PhD thesis, University of Strathclyde, 2013. URL <http://gallium.inria.fr/~pdagand/stuffs/thesis-2011-phd/thesis.pdf>.