Ornaments in Practice

Thomas Williams, Pierre-Évariste Dagand, Didier Rémy

Inria

August 29, 2014

Motivation

Very similar data structures expressed as algebraic data types:

- trees with values at the leaves, at the nodes, etc
- GADTs encoding different invariants

Very similar functions on these structures

Motivation

Very similar data structures expressed as algebraic data types:

- trees with values at the leaves, at the nodes, etc
- GADTs encoding different invariants

Very similar functions on these structures

Ornaments (McBride,2010)

- express the link between similar datatypes
- between operations on these types

Naturals and lists

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

Naturals and lists

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

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

Naturals and lists

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

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

Projection function:

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

ornament from length : α list → nat
```

Valid ornaments

Intuitively, an ornament match values from an *ornamented* datatype to values of a *bare* type.

- Project the constructors from the *ornamented* to the *bare* type
- maybe forget some information
- while keeping the recursive structure of the value

An ornament is defined by a projection function, subject to some syntactic conditions described in our paper.

Relating functions

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

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

Coherence:

```
length (append ml nl) = add (length ml) (length nl)
project (f_lifted x y) = f (project x) (project y)
```

Lifting functions

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

Lifting functions

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

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

Lifting functions

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

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

Lifting functions

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

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

let rec append ml nl =
```

Lifting functions

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

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

let rec append ml nl =




length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl =




length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl = match  with




length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl = match  with
```

```
length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl = match ml with




length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl = match ml  with




length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl = match ml  with
  | Nil →



length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl = match ml  with
  | Nil →



length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl = match ml  with
   | Nil → nl



length ml = m
length nl = n
```

Lifting functions

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

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

let rec append ml nl = match ml  with
  | Nil → nl



length ml = m
length nl = n
```

Lifting functions

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

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

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


length ml = m
length nl = n
length ml' = m'
```

Lifting functions

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

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

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


length ml = m
length nl = n
length ml' = m'
```

Lifting functions

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

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

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


length ml = m
length nl = n
length ml' = m'
```

Lifting functions

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

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

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


length ml = m
length nl = n
length ml' = m'
```

Lifting functions

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

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

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


length ml = m
length nl = n
length ml' = m'
```

Lifting functions

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

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

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


length ml = m
length nl = n
length ml' = m'
```

Lifting functions

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

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

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


length ml = m
length nl = n
length ml' = m'
```

Filling the missing part

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

- Manually, by intervention of the programmer
- With a *patch* specifying what should be added where
- Code inference: `x` makes the most sense here

The other liftings

```
length (add_lifted ml nl) = add (length ml) (length nl)

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

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

Ornaments for refactoring

```
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
```

Ornaments for refactoring

```
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

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

Ornaments for refactoring (2)

```
let rec convert : expr' → expr = 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

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

The projection `convert` is bijective: the lifting is uniquely defined.

Ornaments for refactoring (2)

```
let rec convert : expr' → expr = 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

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

The projection convert is bijective: the lifting is uniquely defined.

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

Lifting data structures

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

type α map =
  | MEmpty
  | MNode of key × α × α map × α map

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

Lifting an higher-order function

```
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
```

Lifting an higher-order function

```
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
```

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

Lifting an higher-order function

```
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
```

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

```
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
```

GADTs

Several data structures with the same contents but different invariants, *i.e.* a constraint on the shape of the type.
Lists and vectors:

```
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

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
```

The lifting should be unambiguous.

Lifting for GADTs

Automatic for some invariants, we only need to give the expected type of the function:

```
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"
```

Lifting for GADTs

Automatic for some invariants, we only need to give the expected type of the function:

```
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"

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

Lifting for GADTs

Automatic for some invariants, we only need to give the expected type of the function:

```
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"

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

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"
```

Conclusion

1. Describing ornaments by projection is a good fit for ML
2. There are ornaments in the wild
3. The automatic lifting is incommplete, but gives good and predictable results

Questions ?