Ornaments in Practice

Thomas Williams
Encadré par: Pierre-Évariste Dagand, Didier Rémy

INRIA - Gallium

September 8, 2014
Motivation

Two very similar functions

```ocaml
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)
```
Motivation

Two very similar functions

```ocaml
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)
```

Coherent

```
add (length ml) (length nl) = length (append ml nl)
```
Naturals and lists

Similar types

\[
\text{type } \text{n}at = Z \mid S \text{ of } \text{n}at \\
\text{type } \alpha \text{ list} = \text{Nil} \mid \text{Cons of } \alpha \times \alpha \text{ list}
\]

\[
S \ ( \ S \ ( \ S \ ( \ Z ))) \\
\text{Cons}(1, \text{Cons}(2, \text{Cons}(3, \text{Nil})))
\]

Projection function

\[
\text{let rec } \text{length} = \text{function} \\
| \text{Nil} \to Z \\
| \text{Cons}(x, \text{xs}) \to S(\text{length} \ \text{xs})
\]
Motivation

Trees

\[
\text{type } \text{tree} = \begin{cases} 
\text{LLeaf} \\
\text{LNode of ltree } \times \text{lltree}
\end{cases}
\]

\[
\text{type } \alpha \text{ ntree} = \begin{cases} 
\text{NLeaf} \\
\text{NNode of } \alpha \text{ ntree } \times \alpha \\
\times \alpha \text{ ntree}
\end{cases}
\]

GADTs

\[
\text{type } \alpha \text{ list} = \begin{cases} 
\text{Nil} \\
\text{Cons of } \alpha \times \alpha \text{ list}
\end{cases}
\]

\[
\text{type } (\_, \alpha) \text{ vec} = \begin{cases} 
\text{VNil : (zero, } \alpha \text{ vec} \\
\text{VCons : } \alpha \times (n, \alpha) \text{ vec} \\
\rightarrow (n \text{ succ, } \alpha) \text{ vec}
\end{cases}
\]

\[
\text{type } \text{zero} = \text{Zero} \quad \text{type } \_ \text{ succ} = \text{Succ}
\]

Ornaments (McBride, 2010; Dagand, 2012)
Ornaments in ML

Ornaments were developed in type theory. Can they be adapted to ML?

Toy implementation: a preprocessor for a small System F-like language with GADTs. It adapts easily to ML: we will assume this in the examples.
Contents

1. Ornaments in ML
2. Applications
3. Theory
A syntax for ornaments

An ornament is defined by a *projection function* from the ornamented type to the bare type.

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

The function is subject to some syntactic restrictions to ensure it preserves the recursive structure. They are checked by the system when declaring an ornament:

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

Coherence

\[ \text{length} \ (\text{append} \ ml \ nl) = \text{add} \ (\text{length} \ ml) \ (\text{length} \ nl) \]
Lifting functions

Coherence

\[ \text{length} \ (\text{append} \ ml \ nl) = \text{add} \ (\text{length} \ ml) \ (\text{length} \ nl) \]

\[ \text{project} \ (f_{\text{lifted}} \ x \ y) = f \ (\text{project} \ x) \ (\text{project} \ y) \]

The function \( f_{\text{lifted}} \) is a lifting of \( f \).
Syntactic lifting

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

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

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

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}
Syntactic lifting

```
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}
```
Syntactic lifting

```ocaml
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 =
```

```
m' ← ml'

n ← nl

add m' n ← append ml' nl
```

```
m' ← ml'

n ← nl

add m' n ← append ml' nl
```
Syntactic lifting

\[
\text{let rec}\ \text{add}\ m\ n = \text{match}\ m\ \text{with}
\]
\[
\begin{align*}
| Z & \rightarrow n \\
| S\ m' & \rightarrow S\ (\text{add}\ m'\ n)
\end{align*}
\]

\[
\text{let lifting}\ \text{append}\ \text{from}\ \text{add}\ \text{with}
\]
\[
\begin{align*}
\{\text{length}\} & \rightarrow \{\text{length}\} \rightarrow \{\text{length}\}
\end{align*}
\]

\[
\text{let rec}\ \text{append}\ ml\ nl =
\]

\[
\begin{align*}
m' & \leftarrow ml' \\
n & \leftarrow nl \\
\text{add}\ m'\ n & \leftarrow \text{append}\ ml'\ nl
\end{align*}
\]
Syntactic lifting

```ml
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
  | Nil -> nl
  | Cons(x,ml') -> Cons(x, append ml' nl)
```

```
m' ← length ml'

n ← length nl

dd m' n ← length append ml' nl
```

```
Z ← length Nil

n ← length nl

S(n) ← length Cons(x, nl)
```

```
m ← length ml

n ← length nl
```
Syntactic lifting

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

m' ← ml'

n ← nl

add m' n ← append ml' nl

m ← ml

n ← nl
Syntactic lifting

```ocaml
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)
```

```
m'  \text{length} \rightarrow ml'
n  \text{length} \rightarrow nl
\hline
\text{add} \hspace{1cm} m' \hspace{1cm} n \hspace{1cm} \text{length} \rightarrow \text{append} \hspace{1cm} ml' \hspace{1cm} nl
```

```
m \hspace{1cm} \text{length} \rightarrow ml
n \hspace{1cm} \text{length} \rightarrow nl
```

**Syntactic lifting**

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

m’ ← ml’
m ← ml
n ← nl

add m’ n ← append ml’ nl

n ← nl

m ← ml
Syntactic lifting

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

let lifting append from add with
{\text{length}} \rightarrow {\text{length}} \rightarrow {\text{length}}

let rec append ml nl = match ml with

\[
\begin{align*}
\text{m' } &\xleftarrow{\text{length}} \text{ ml'} \\
\text{n } &\xleftarrow{\text{length}} \text{ nl} \\
\text{add } m' \ n &\xrightarrow{\text{length}} \text{ append ml' nl} \\
\text{Z } &\xrightarrow{\text{length}} \text{ Nil} \\
\text{n } &\xleftarrow{\text{length}} \text{ nl} \\
\text{S(n) } &\xrightarrow{\text{length}} \text{ Cons(x, nl)} \\
\text{m } &\xleftarrow{\text{length}} \text{ ml} \\
\text{n } &\xleftarrow{\text{length}} \text{ nl}
\end{align*}
\]
Syntactic lifting

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)
Syntactic lifting

let rec add m n = match m 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 ml with
  | Nil →
  | Cons(x,ml') →

\[
\begin{align*}
\text{add } m' n & \leftarrow \text{ append } ml' nl \\
Z & \leftarrow \text{ Nil} \\
S(n) & \leftarrow \text{ Cons}(x, nl) \\
\end{align*}
\]
Syntactic lifting

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') →
Syntactic lifting

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') →

m' \xrightarrow{\text{length}} ml'

\[
\begin{align*}
  m' & \xrightarrow{\text{length}} ml' \\
  n & \xrightarrow{\text{length}} nl \\
  \text{add m' n} & \xrightarrow{\text{length}} \text{append ml' nl} \\
  Z & \xrightarrow{\text{length}} \text{Nil} \\
  \text{n} & \xrightarrow{\text{length}} \text{nl} \\
  \text{S(n)} & \xrightarrow{\text{length}} \text{Cons(x, nl)} \\
  m & \xrightarrow{\text{length}} ml \\
  \text{n} & \xrightarrow{\text{length}} nl \\
  m' & \xrightarrow{\text{length}} ml'
\end{align*}
\]
Syntactic lifting

```
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)
```

```
m' ← ml'

n ← nl

add m' n ← append ml' nl

Z ← Nil

| length

n ← nl

S(n) ← Cons(x, nl)

| length

m ← ml

| length

n ← nl

m' ← ml'
```
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)
Syntactic lifting

\[
\text{let rec add m n = match m with}
\begin{align*}
| Z & \rightarrow n \\
| S \ m' & \rightarrow S (\text{add } m' \ n)
\end{align*}
\]

\[
\text{let lifting append from add with}
\begin{align*}
\{ \text{length} \} & \rightarrow \{ \text{length} \} \rightarrow \{ \text{length} \}
\end{align*}
\]

\[
\text{let rec append ml nl = match ml with}
\begin{align*}
| \text{Nil} & \rightarrow nl \\
| \text{Cons}(x,ml') & \rightarrow
\begin{align*}
\text{Cons}(\quad, \text{append } ml' \ nl)
\end{align*}
\end{align*}
\]
Syntactic lifting

let rec add m n = match m 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 ml with
  | Nil → nl
  | Cons(x,ml') → Cons(?, append ml' nl)
Two phases

Syntactic lifting

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

The creative part

- Manually, by intervention of the programmer
- With a *patch* specifying what should be added where
  ```ocaml
  let append from add
    with {length} → {length} → {length}
  patch fun _ → match _ with Cons(x, _) → Cons({x}, _)
  ```
- Code inference: `x` makes the most sense here
Coherence is not enough

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

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

The coherence condition is not strong enough to guide the automatic lifting process.
Contents

1. Ornaments in ML
2. Applications
3. Theory
Ornaments for refactoring

```plaintext
type expr =
  | Const of int
  | Add of expr × expr
  | Mul of expr × expr

type binop = Add’ | Mul’

type expr’ =
  | Const’ of int
  | BinOp’ of binop × 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 (2)

let rec conv : expr' → expr = function
  | Const'(i) → Const(i)
  | BinOp(Add', u, v) → Add(conv u, conv v)
  | BinOp(Mul', u, v) → Mul(conv u, conv v)

ornament from conv : expr' → expr

let lifting eval' from eval with {conv} → _

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

The lifting is unique, because conv is bijective.
Other applications

Large-scale lifting of data structures

- Ocaml’s Set library to maps (sets with values)
- Including higher-order functions
- Only the values need to be propagated

GADTs

- A GADT is an ornament: constraints are added
- Lifting unique: the contents are the same
- In practice, works for what the type checker could have proved
Contents

1. What are ornaments?
2. Applications
3. Theory
let rec add m n = match m with
  | Z → n
  | S(m’) → S(add m’ n)

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)

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

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

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

The & in names is only a notation, it has no meaning in the binary language.

Ornaments translate to binary type definitions.

\[
\text{type } \{\alpha\} \text{ nat\&list } = \\
| \text{Z\&Nil} \\
| \text{S\&Cons of } \{\alpha\} \times\{\alpha\} \text{ nat\&list}
\]

The typing enforces ornamentation:

\[
\text{val add\&append} \\
: \{\alpha\}. \{\alpha\} \text{ nat\&list } \rightarrow \{\alpha\} \text{ nat\&list } \rightarrow \{\alpha\} \text{ nat\&list}
\]

The braces guarantee that values don’t escape from the ornamented code to the bare code.
Lifting with binary terms

Lifting: finding a binary term that \textit{projects} to the base term and has the right type.

1. The binary typing relation guarantees that we have a valid lifting.
2. The projections of a well-typed term are well-typed.
3. The complexity is preserved: the additional complexity comes only from the code added between brackets.
4. If the ornamented code terminates, the base code terminates too.
What’s missing?

**Semantic ornaments**
We can recover a semantic definition using contextual equivalence. All syntactic rules remain admissible (it is *compatible*), it is a superset of syntactic equivalence (it is *adequate*), and equivalent to the definition using coherence.

**Higher-order and nested ornaments**
We can use this to understand what is a higher-order ornament and a nested ornament.
Conclusion

What we have learned

▶ Describing ornaments by projection is a good fit for ML
▶ Several classes of useful ornaments
▶ The syntactic lifting gives good, predictable results
▶ And can be well-explained by theory

Future work

▶ Better patches
▶ Integrating into ML: effects? inference?
▶ Combining ornaments: adding the invariants of two GADTs?
Questions ?
Interaction with type inference: inferring the ornament specification of let-bound values?

```ocaml
let rev xs =
  let rec rev_append acc = function
    | x xs -> rev_append (x : acc) xs
    | [] -> acc
  in
  rev_append [] xs
```

Lifting effectful libraries?
Lifting more complex data structures

Sets

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

Maps

```ocaml
type α map =
| MEmpty
| MNode of t × α × α map × α map
```

Ornament

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

```ocaml
let rec exists (p : t → 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 (t → +α → t) → {keys} → bool
```
Lifting a higher-order function

```ocaml
let rec exists (p : t → 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 (t → +α → t) → {keys} → bool

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

**Lists and vectors**

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

The lifting should be unambiguous.
Lifting for GADTs

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

```ocaml
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:

```ocaml
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"
```

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

```ocaml
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"
```
When lifting fails

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

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\}

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"