Ornaments in ML

Thomas Williams, Didier Rémy

Inria - Gallium

April 18, 2017

Motivation

Two very similar 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)
```

Motivation

Two very similar 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)
```

Coherent

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

Naturals and lists

Similar types

```
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)
```

The relation between nat and α list defines an ornament.

Lifting a function

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

Lifting a function

```
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)
```

Lifting a function

```
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)
```

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

Lifting a function

```
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)
```

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

Syntactic lifting: we follow the structure of the original function.

Outline

Some examples

Encoding lifting

A meta-language for ornamentation

Encoding ornaments in mML

add/append again

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

add/append again

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

type ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)
```

add/append again

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

type ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)

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

add/append again

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

type ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)

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

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

add/append again

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

type ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)

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

let append = lifting add : _ natlist → _ natlist → _ natlist


let rec append ml nl = match ml with
  | Nil → nl
  | Cons(x,ml') → Cons(#1, append ml' nl)
```

add/append again

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

type ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)

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

let append = lifting add : _ natlist → _ natlist → _ natlist


let rec append ml nl = match ml with
  | Nil → nl
  | Cons(x,ml') → Cons(#1, append ml' nl)
```

add/append again

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

type ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)

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

let append = lifting add : _ natlist → _ natlist → _ natlist with
  | #1 <- (match ml with Cons(x,_) → x)

let rec append ml nl = match ml with
  | Nil → nl
  | Cons(x,ml') → Cons(#1, append ml' nl)
```

add/append again

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

type ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)

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

let append = lifting add : _ natlist → _ natlist → _ natlist with
  | #1 <- (match ml with Cons(x,_) → x)

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

Refactoring

```
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'
```

Refactoring

```
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'
```

```
type ornament oexpr : expr → expr' with
  | Const i → Const' i
  | Add(u, v) → Binop'(Add', u, v)
  | Mul(u, v) → Binop'(Mul', u, v)
```

Refactoring

```
let rec eval e = match e with
| Const i → i
| Add ( u , v ) → add (eval u)  (eval v)
| Mul ( u , v ) → mul (eval u)  (eval v)
```

Refactoring

```
let rec eval e = match e with
| Const i → i
| Add ( u , v ) → add (eval u)  (eval v)
| Mul ( u , v ) → mul (eval u)  (eval v)

let eval' = lifting eval : oexpr → int
```

Refactoring

```
let rec eval e = match e with
| Const i → i
| Add ( u , v ) → add (eval u)  (eval v)
| Mul ( u , v ) → mul (eval u)  (eval v)

let eval' = lifting eval : oexpr → int

let rec eval' e = match e with
  | Const' x → x
  | Binop'(Add', x, x') → add (eval' x) (eval' x')
  | Binop'(Mul', x, x') → mul (eval' x) (eval' x')
```

Why not use the typechecker for refactoring?

- We do automatically what the programmer must do manually.
- We can guarantee that the program obtained is related to the original program.
- The typechecker misses some places where a change is necessary.

Why not use the typechecker for refactoring?

- We do automatically what the programmer must do manually.
- We can guarantee that the program obtained is related to the original program.
- The typechecker misses some places where a change is necessary.

Permuting values

```
type bool = False | True
```

We can safely exchange True and False in some places:

Why not use the typechecker for refactoring?

- We do automatically what the programmer must do manually.
- We can guarantee that the program obtained is related to the original program.
- The typechecker misses some places where a change is necessary.

Permuting values

```
type bool = False | True
```

We can safely exchange True and False in some places:

```
type ornament not : bool → bool with
  | True → False
  | False → True
```

The relations between bare and ornamented values are tracked through the program (by *ornament* inference).

Ornament inference

The ornamentation constraints are propagated as with type inference.

```
let rec add_gen m n = match (orn-match #3) m with
    | Z_skel → n
    | S_skel x → (orn-cons #1) (S_skel (add_gen x n)) #2
val add_gen : ∀(α < nat)(β < nat). α → β → β
```

Specialization

```
type α map =
  | Node of α map × key × α × α map
  | Leaf
```

Specialization

```
type α map =
  | Node of α map × key × α × α map
  | Leaf
```

Instead of `unit map`, we could use a more compact representation:

```
type set =
  | SNode of set × key × set
  | SLeaf
```

Specialization

```
type α map =
  | Node of α map × key × α × α map
  | Leaf
```

Instead of `unit map`, we could use a more compact representation:

```
type set =
  | SNode of set × key × set
  | SLeaf
```

```
type ornament mapset : unit map → set with
  | Node(l,k,(),r) → SNode(l,k,r)
  | Leaf → SLeaf
```

Specialization: unboxing

```
type α option =
  | None
  | Some of α
```

```
type booloption =
  | NoneBool
  | SomeTrue
  | SomeFalse
```

Specialization: unboxing

```
type α option =
  | None
  | Some of α
```

```
type booloption =
  | NoneBool
  | SomeTrue
  | SomeFalse
```

```
type ornament boolopt : bool option → booloption with
  | None → NoneBool
  | Some(true) → SomeTrue
  | Some(false) → SomeFalse
```

And also...

- Specialization: removing cases
- Future work: adding invariants using GADTs

Outline

Some examples

Encoding lifting

A meta-language for ornamentation

Encoding ornaments in mML

From add to append

Idea: insert some conversion code in add to obtain append.

First attempt:

```
let rec append m n =
  match list2nat m with
    | Z → n
    | S m' → nat2list (S (append m' n))
```

From add to append

Idea: insert some conversion code in add to obtain append.

First attempt:

```
let rec append m n =
  match list2nat m with
    | Z → n
    | S m' → nat2list (S (append m' n))


let rec list2nat a = match a with
  | Nil → Z
  | Cons(_,xs) → S (list2nat xs)
```

From add to append

Idea: insert some conversion code in add to obtain append.

First attempt:

```
let rec append m n =
  match list2nat m with
    | Z → n
    | S m' → nat2list (S (append m' n))
```

```
let rec list2nat a = match a with
  | Nil → Z
  | Cons(_,xs) → S (list2nat xs)
let rec nat2list n = match n with
  | Z → Nil
  | S xs → Cons( ? , nat2list xs)
```

From add to append

Idea: insert some conversion code in add to obtain append.

First attempt:

```
let rec append m n =
  match list2nat m with
    | Z → n
    | S m' → nat2list (S (list2nat
                 (append (nat2list m') n))

let rec list2nat a = match a with
  | Nil → Z
  | Cons(_,xs) → S (list2nat xs)
let rec nat2list n = match n with
  | Z → Nil
  | S xs → Cons( ? , nat2list xs)
```

Opening the recursion

Ignoring for the moment the missing argument to Cons, we can solve our problems by incrementalizing.

```
type α nat_skel = Z' | S' of α
let list2nat' a = match a with
  | Nil → Z'
  | Cons(_,xs) → S' xs
let nat'2list n = match n with
  | Z' → Nil
  | S' xs → Cons( ? , xs)
```

Opening the recursion

Ignoring for the moment the missing argument to Cons, we can solve our problems by incrementalizing.

```
type α nat_skel = Z' | S' of α
let list2nat' a = match a with
  | Nil → Z'
  | Cons(_,xs) → S' xs
let nat'2list n = match n with
  | Z' → Nil
  | S' xs → Cons( ? , xs)

let rec append m n =
  match list2nat' m with
    | Z' → n
    | S' m' → nat'2list (S' (append m' n))
```

Opening the recursion

Now the extra information can simply be passed as argument to nat'2 list .

```
type α nat_skel = Z' | S' of α
let list2nat ' a = match a with
  | Nil → Z'
  | Cons(_,xs) → S' xs
let nat'2 list n x = match n with
  | Z' → Nil
  | S' xs → Cons(x, xs)

let rec append m n =
  match list2nat ' m with
    | Z' → n
    | S' m' → nat'2list (S' (append m' n)) (List.hd m)
```

Marking the encoding

The encoding introduces a lot of function calls that we would like to eliminate. We separate normal function calls from calls to ornamentation functions: meta-abstraction and application are noted **#**.

```
type α nat_skel = Z' | S' of α
let list2nat' = fun l #> match l with
  | Nil → Z'
  | Cons(_,xs) → S' xs
let nat'2list = fun n x #> match n with
  | Z' → Nil
  | S' xs → Cons(x, xs)

let rec append m n =
  match list2nat' # m with
    | Z' → n
    | S' m' → nat'2list # (S' (append m' n)) # (List.hd m)
```

Marking the encoding

The encoding introduces a lot of function calls that we would like to eliminate. We separate normal function calls from calls to ornamentation functions: meta-abstraction and application are noted **#**.

```
type α nat_skel = Z' | S' of α
let list2nat ' = fun l #> match l with
  | Nil → Z'
  | Cons(_,xs) → S' xs
let nat'2 list = fun n x #> match n with
  | Z' → Nil
  | S' xs → Cons(x, xs)

let rec append m n =
  match list2nat ' # m with
    | Z' → n
    | S' m' → nat'2list # (S' (append m' n)) # (List.hd m)
```

They can then be reduced without affecting the rest of the code.

Eliminating ornamentation calls

They can then be reduced without affecting the rest of the code:

```
let rec append m n =
  match list2nat ' # m  with


     | Z' → n
     | S' m' → nat'2list # (S' (append m' n)) # (List.hd m)
```

Eliminating ornamentation calls

They can then be reduced without affecting the rest of the code:

```
let rec append m n =
  match (match m with
           | Nil → Z'
           | Cons(_, xs) → S' xs) with
     | Z' → n
     | S' m' →
       (match S' (append m' n) with
          | Z' → Nil
          | S' zs → Cons(List.hd m, zs))
```

Eliminating ornamentation calls

They can then be reduced without affecting the rest of the code:

```
let rec append m n =
  match (match m with
         | Nil → Z'
         | Cons(_, xs) → S' xs) with
    | Z' → n
    | S' m' →
      (match S' (append m' n) with
         | Z' → Nil
         | S' zs → Cons(List.hd m, zs))
```

There remains two redundant pattern matchings, decoding lists to nat_skel and encoding nat_skel to lists.

Eliminating the encoding

There remains two redundant pattern matchings, decoding lists to nat_skel and encoding nat_skel to lists. We can eliminate them by reduction:

```
let rec append m n =
  match (match m with
          | Nil → Z'
          | Cons(_, xs) → S' xs) with
     | Z' → n
     | S' m' → Cons(List.hd m, append m' n)
```

There remains two redundant pattern matchings, decoding lists to nat_skel and encoding nat_skel to lists. We can eliminate them by reduction, and by extruding the nested pattern matching:

```
let rec append m n =
  match m with
    | Nil →
      (match Z' with
          | Z' → n
          | S' m' → Cons(List.hd m, append m' n))
    | Cons(_, xs) →
      (match S' xs with
          | Z' → n
          | S' m' → Cons(List.hd m, append m' n))
```

There remains two redundant pattern matchings, decoding lists to nat_skel and encoding nat_skel to lists. We can eliminate them by reduction, and by extruding the nested pattern matching, and reducing again:

```
let rec append m n =
  match m with
    | Nil → n
    | Cons(_, xs) → Cons(List.hd m, append m' n)
```

... and we obtain the code for append.

add, generalized

```
let add_gen = fun m2nat' nat'2m
                  n2nat' nat'2n patch ⇏
  let rec add m n =
    match m2nat' # m with
      | Z' → n
      | S' m' → nat'2n # S' (add m' n) # patch m n
  in
  add
```

add, generalized

```
let add_gen = fun m2nat' nat'2m
                  n2nat' nat'2n patch ⇏
  let rec add m n =
    match m2nat' # m with
      | Z' → n
      | S' m' → nat'2n # S' (add m' n) # patch m n
  in
  add

let append = add_gen # list2nat' # nat'2list
                     # list2nat' # nat'2list
              # (fun m _ → match m with Cons(x,_) → x)
```

add, generalized

```
let add_gen = fun m2nat' nat'2m
                  n2nat' nat'2n patch ⇏
  let rec add m n =
    match m2nat' # m with
      | Z' → n
      | S' m' → nat'2n # S' (add m' n) # patch m n
  in
  add

let append = add_gen # list2nat' # nat'2list
                     # list2nat' # nat'2list
               # (fun m _ → match m with Cons(x,_) → x)
let add = add_gen # nat2nat' # nat'2nat
                  # nat2nat' # nat'2nat
                  # (fun _ _ → ())
```

Why generalize?

- We need to be able to represent partially-instantiated terms to display it to the user.
- A completely uninstantiated is a natural output for ornament inference in the absence of any annotation.
- The completely uninstantiated term can be instantiated by the identity to give back the original term. This will be useful for proving correctness.

Summarizing the process

1. Generalize the base code
2. Instanciate with specific ornaments and patches
3. Reduce to eliminate the meta code
4. Simplify the pattern matching

The case for dependent types

What if we add data to the Z constructor too ?

```
type α stream = End | Continued | More of α × α stream
ornament α natstream : nat → α stream with
  | Z → (End | Continued)
  | S n → More(_, n)
```

The case for dependent types

What if we add data to the Z constructor too ?

```
type α stream = End | Continued | More of α × α stream
ornament α natstream : nat → α stream with
  | Z → (End | Continued)
  | S n → More(_, n)

let nat'2stream n x =
  match n with
    | Z' → (match x with
              | true → Continued
              | false → End)
    | S' n' → More(x,n')
```

The case for dependent types

What if we add data to the Z constructor too ?

```
type α stream = End | Continued | More of α × α stream
ornament α natstream : nat → α stream with
  | Z → (End | Continued)
  | S n → More(_, n)

let nat'2stream n x =
  match n with
    | Z' → (match x with
              | true → Continued
              | false → End)
    | S' n' → More(x,n')
```

What is the type of x?

The case for dependent types

What if we add data to the Z constructor too ?

```
type α stream = End | Continued | More of α × α stream
ornament α natstream : nat → α stream with
  | Z → (End | Continued)
  | S n → More(_, n)

let nat'2stream n x =
  match n with
    | Z' → (match x with
              | true → Continued
              | false → End)
    | S' n' → More(x,n')
```

What is the type of x?

$$\text{match } n \text{ with } Z' \to \text{bool} \mid S' \; \_ \to \alpha$$

The case for dependent types

The type may depend on more than the constructor.

```
type α list01 =
  | Nil01
  | Cons0 of α list01
  | Cons1 of α × α list01

ornament α olist01 : bool list → α list01 with
  | Nil → Nil01
  | Cons(False, xs) → Cons0(xs)
  | Cons(True, xs) → Cons1(_, xs)
```

The case for dependent types

The type may depend on more than the constructor.

```
type α list01 =
  | Nil01
  | Cons0 of α list01
  | Cons1 of α × α list01

ornament α olist01 : bool list → α list01 with
  | Nil → Nil01
  | Cons(False, xs) → Cons0(xs)
  | Cons(True, xs) → Cons1(_, xs)
```

$$\begin{array}{l} \text{match } m \text{ with} \\ \quad \mid \text{Nil}' \to \text{unit} \\ \quad \mid \text{Cons}'\ (\text{False}, \_) \to \text{unit} \\ \quad \mid \text{Cons}'\ (\text{True}, \_) \to \alpha \end{array}$$

Outline

Starting from ML

$$
\begin{aligned}
\tau, \sigma &::= \alpha \mid \tau \to \tau \mid \zeta\ \overline{\tau} \mid \forall(\alpha : \mathsf{Typ})\, \tau \\
a, b &::= x \mid \mathsf{let}\ x = a\ \mathsf{in}\ a \mid \mathsf{fix}\,(x : \tau)\, x.\ a \mid a\ a \\
&\mid\ \Lambda(\alpha : \mathsf{Typ}).\ u \mid a\ \tau \mid d\ \overline{\tau}\ \overline{a} \mid \mathsf{match}\ a\ \mathsf{with}\ \overline{P \to a} \\
P &::= d\ \overline{\tau}\ \overline{x}
\end{aligned}
$$

Starting from ML

$$E ::= [] \mid E\ a \mid v\ E \mid d(v, ..\ v, E, a, ..\ a) \mid \Lambda(\alpha : \mathsf{Typ}).\ E \mid E\ \tau \\ \mid\ \text{match } E \text{ with } \overline{P \to a} \mid \text{let } x = E \text{ in } a$$

$$\begin{aligned}
(\text{fix}\,(x:\tau)\,y.\,a)\ v &\longrightarrow^h_\beta a[x \leftarrow \text{fix}\,(x:\tau)\,y.\,a, y \leftarrow v] \\
(\Lambda(\alpha : \mathsf{Typ}).\ v)\ \tau &\longrightarrow^h_\beta v[\alpha \leftarrow \tau] \\
\text{let } x = v \text{ in } a &\longrightarrow^h_\beta a[x \leftarrow v] \\
\begin{array}{r}\text{match } d_j\,\overline{\tau_j}\,(v_i)^i \text{ with} \\ (d_j\,\overline{\tau_j}\,(x_{ji})^i \to a_j)^j\end{array} &\longrightarrow^h_\beta a_j[x_{ij} \leftarrow v_i]^i
\end{aligned}$$

Context-Beta

$$\frac{a \longrightarrow^h_\beta b}{E[a] \longrightarrow_\beta E[b]}$$

From ML to *m*ML

- *e*ML: add type-level pattern matching and equalities.
- *m*ML: add dependent, meta-abstraction and application.

Reduction (under some typing conditions):

- From *m*ML, reduce meta-application and get a term in *e*ML
- From *e*ML, eliminate type-level pattern matching and get a term in ML

*e*ML

*e*ML is obtained by extending the type system of ML.

*e*ML

*e*ML is obtained by extending the type system of ML.

$$\Gamma = \alpha : \mathsf{Typ}, m : \mathsf{nat}' \ (\mathsf{list}\ \alpha), x : \mathsf{match}\ m\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha$$

$$\begin{aligned} &\mathsf{match}\ m\ \mathsf{with} \\ &\quad \mid \mathsf{Z}' \to \mathsf{Nil} \\ &\quad \mid \mathsf{S}'\ \ m' \to \mathsf{Cons}\ (x, m') \end{aligned}$$

eML

eML is obtained by extending the type system of ML.

$$\Gamma = \alpha : \mathsf{Typ}, m : \mathsf{nat}' \ (\mathsf{list}\ \alpha), x : \mathsf{match}\ m\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha$$

$$\begin{aligned}&\mathsf{match}\ m\ \mathsf{with}\\ &\qquad \mid \mathsf{Z}' \to \mathsf{Nil}\\ &\qquad \mid \mathsf{S}'\ \ m' \to \mathsf{Cons}\ (x, m')\end{aligned}$$

In the S' branch: we know $m = \mathsf{S}'\ \ m'$. Thus:

$$\begin{aligned} x \quad &: \quad \mathsf{match}\ m\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha\\ &= \quad \mathsf{match}\ \mathsf{S}'\ \ m'\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha\\ &= \quad \alpha\end{aligned}$$

Introducing equalities

We extend the typing environment with equalities:

$$\Gamma \ ::= \ \ldots \mid \Gamma, a =_\tau b$$

Introducing equalities

We extend the typing environment with equalities:

$$\Gamma \ ::= \ \dots \mid \Gamma, a =_\tau b$$

Equalities are introduced on pattern matching:

$$\frac{\Gamma \vdash \tau : \mathsf{Sch} \qquad (d_i : \forall (\alpha_k : \mathsf{Typ})^k \, (\tau_{ij})^j \to \zeta \, (\alpha_k)^k)^i \quad \Gamma \vdash a : \zeta \, (\tau_k)^k \quad (\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, a =_{\zeta \, (\tau_k)^k} d_i(\tau_{ij})^k (x_{ij})^j \vdash b_i : \tau)^i}{\Gamma \vdash \text{match } a \text{ with } (d_i(\tau_{ik})^k (x_{ij})^j \to b_i)^i : \tau}$$

Eliminating equalities

The equalities in the context are used to prove *type equalities*: $\Gamma \vdash \tau_1 \simeq \tau_2$.

This type equalities can be used to convert (implicitly) in typing derivations:

$$\frac{\Gamma \vdash \tau_1 \simeq \tau_2 \qquad \Gamma \vdash a : \tau_1}{\Gamma \vdash a : \tau_2}$$

Elimination of equalities

We restrict reduction in equalities so that they stay decidable.

Suppose we have a term a in eML such that $\Gamma \vdash a : \tau$, where Γ and τ are in ML. Then, we can transform a into a well-typed ML term by:

- Using an equality to substitute in a term
- Extruding a nested pattern matching
- Reduding pattern matching

Thus it makes sense to use eML as an intermediate language for ornamentation.

Meta-programming in *m*ML

We want to be able to eliminate all abstractions and applications marked with **#**. We introduce a separate type for meta-functions, so that they can only be applied using meta-application.

$$(\lambda^\sharp(x:\tau).\,a)\,\sharp\,u \longrightarrow^h_\sharp a[x \leftarrow u]$$

We restrict the types so meta-constructions can not be manipulated by the ML fragment.

Meta-reduction

If there are no meta-typed variables in the context, the meta-reduction $\longrightarrow_\sharp$ will eliminate all meta constructions and give an *e*ML term.

But the meta-reduction also commutes with the ML reduction.

We thus have two dynamic semantics for the same term:

- For reasoning, we can consider that meta and ML reduction are interleaved.
- We can use the meta reduction in the first stage to compile an *m*ML term down to an *e*ML term.

Dependent functions

Since meta-abstraction and meta-application will be eliminated, we enrich them with some features that could not exist in ML or *e*ML and that we need to encode ornaments.

We need dependent types for the encoding function:

$$
\begin{aligned}
\mathsf{nat'\_to\_list} : \quad & \Pi(x : \mathsf{nat'}\ (\mathsf{list}\ \alpha)). \\
& \Pi(y : \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Z'} \to \mathsf{unit} \mid \mathsf{S'}\ \_ \to \alpha). \\
& \mathsf{list}\ \alpha
\end{aligned}
$$

Dependent functions

Since meta-abstraction and meta-application will be eliminated, we enrich them with some features that could not exist in ML or *e*ML and that we need to encode ornaments.

We need dependent types for the encoding function:

$$\begin{aligned}\mathsf{nat'\_to\_list} : \quad & \Pi(x : \mathsf{nat'}\ (\mathsf{list}\ \alpha)).\\ & \Pi(y : \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Z'} \rightarrow \mathsf{unit} \mid \mathsf{S'}\ \_ \rightarrow \alpha).\\ & \mathsf{list}\ \alpha\end{aligned}$$

For the encoding of ornaments to type correctly, we also add:

- Type-level functions to represent the type of the extra information.
- The ability to abstract on equalities so they can be passed to patches.

Outline

Semantics of ornament specifications

```
let append = lifting add : α natlist → α natlist → α natlist
```

What we mean:

- If ml is a lifting of m (for natlist)
- and nl is a lifting of n
- then append ml nl is a lifting of add m n

Semantics of ornament specifications

```
let append = lifting add : α natlist → α natlist → α natlist
```

What we mean:

- If ml is a lifting of m (for natlist)
- and nl is a lifting of n
- then append ml nl is a lifting of add m n

We build a step-indexed binary logical relation on *m*ML, and add an interpretation for datatype ornaments.
Then, the interpretation of a functional lifting is exactly the interpretation of function types (replace “is a lifting of” by “is related to”).

Datatype ornaments

A datatype ornament naturally gives a relation:

```
ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)
```

Datatype ornaments

A datatype ornament naturally gives a relation:

```
ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)
```

$$(\mathsf{Z}, \mathsf{Nil}) \in \mathcal{V}_k[\mathsf{natlist}\ \tau] \qquad \frac{(u, v) \in \mathcal{V}_k[\mathsf{natlist}\ \tau]}{(\mathsf{S}\ u, \mathsf{Cons}\ (a, v)) \in \mathcal{V}_k[\mathsf{natlist}\ \tau]}$$

Datatype ornaments

A datatype ornament naturally gives a relation:

```
ornament α natlist : nat → α list with
  | Z → Nil
  | S xs → Cons(_, xs)
```

$$(\mathsf{Z}, \mathsf{Nil}) \in \mathcal{V}_k[\mathsf{natlist}\ \tau] \qquad \frac{(u, v) \in \mathcal{V}_k[\mathsf{natlist}\ \tau]}{(\mathsf{S}\ u, \mathsf{Cons}\ (a, v)) \in \mathcal{V}_k[\mathsf{natlist}\ \tau]}$$

We prove that the ornamentation functions are correct relatively to this definition:

- if we construct a natural number and a list from the same skeleton, they are related;
- if we destruct related values, we obtain the same skeleton.

Correctness

- Consider a term a_-.
- Generalize it into a. By the fundamental lemma, a is related to itself.
- Construct an instanciation γ_+ and the identity instanciation γ_-.
- $\gamma_-(a)$ and $\gamma_+(a)$ are related.
- $\gamma_-(a)$ reduces to a_-, preserving the relation.
- Simplify $\gamma_+(a)$ into a_+ (an ML term), preserving the relation
- a_- and a_+ are related.

In practice

- We have a prototype implementation, that follows the process outlined here.
- User interace issues: specifying the instantiation. We take labelled patches and ornaments.
- To build the generic lifting, we have to transform the code: transform deep pattern matching into shallow pattern matching.
- We also expand local polymorphic lets (but this is a user interface problem).
- We try to recover the shape of the original program in a post-processing phase.

Available online:
`http://gallium.inria.fr/~remy/ornaments/`

Conclusion

- Ornaments can be used to lift functions in ML.
- Going through the intermediate language allows a cleaner presentation.
- We proved the lifting is correct.

Future work

- Can we write robust patches?
- A formal result about effects
- Non-regular types, GADTs?
- Should we give the user access to *m*ML?
- Can we use *m*ML for something else?