Ornamentation in ML

Thomas Williams, Didier Rémy

Inria

Edinburgh, June 20, 2017

nat & list

Similar types

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

Similar values

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

Ornament relation

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

The relation α natlist between nat and α list defines an ornament.

add & append

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

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

Coherence

```
add (length m) (length n) = length (append m n)
```

add & append

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

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

Projection
total
(function)

Coherence

```
add (length m) (length n) = length (append m n)
```

add & append

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

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

Projection
total
(function)

Lifting
partial
(relation)

Coherence

add (length m) (length n) = length (append m n)

Lifting

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

let rec append m n = [ ? ]
```

Lifting

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

let rec append m n = [ ? ]
```

```
add (length m) (length n) = length (append m n)
```

Lifting

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

let rec append m n = ?
```

```
add (length m) (length n) = length (append m n)
```

We restrict to syntactic lifting, following the structure of the original function.

Lifting

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

Lifting

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

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

Lifting

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

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

Ouput

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

Lifting

```
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 m with Cons(x,_) → x)
```

Ouput

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

Lifting

```
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 m with Cons(x,_) → x)
```

Ouput

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

More examples

About nat & list

- Canonical, but trivial example
- Still, small enough to be a good running example,
  to explain the details of lifting.

More examples

About nat & list

- Canonical, but trivial example
- Still, small enough to be a good running example,
  to explain the details of lifting.

Many other use cases of lifting

- Pure refactoring: nothing to guess, no need for patches
- Special case: optimizing data representation
- Dealing with administrative stuff, *e.g.* locations:
  Write the code without locations and lift it to the code with locations
- Lifting a library, *e.g.* sets into maps
- Composing liftings: relifting lifted code
- Decomposing lifting into pure refactoring and a true, but simpler lifting

Pure refactoring

```
type exp  =
  | Const of int
  | Add of exp × exp
  | Mul of exp × exp
```

```
type binop' = Add' | Mul'
type exp' =
  | Const' of int
  | Bin' of binop' × exp' × exp'
```

Pure refactoring

```
type exp  =
  | Const of int
  | Add of exp × exp
  | Mul of exp × exp
```

```
type binop' = Add' | Mul'
type exp' =
  | Const' of int
  | Bin' of binop' × exp' × exp'
```

```
type ornament oexp : exp → exp' with
  | Const i    → Const' i
  | Add(u, v) → Bin'(Add', u, v)
  | Mul(u, v) → Bin'(Mul', u, v)
```

Pure refactoring

```
type exp =
  | Const of int
  | Add of exp × exp
  | Mul of exp × exp
```

```
type binop' = Add' | Mul'
type exp' =
  | Const' of int
  | Bin' of binop' × exp' × exp'
```

```
type ornament oexp : exp → exp' with
  | Const i    → Const' i
  | Add(u, v) → Bin'(Add', u, v)
  | Mul(u, v) → Bin'(Mul', u, v)
```

```
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 : oexp → int
```

Pure refactoring

```
type exp  =
  | Const of int
  | Add of exp × exp
  | Mul of exp × exp
```

```
type binop' = Add' | Mul'
type exp' =
  | Const' of int
  | Bin' of binop' × exp' × exp'
```

```
type ornament oexp : exp → exp' with
  | Const i    → Const' i
  | Add(u, v) → Bin'(Add', u, v)
  | Mul(u, v) → Bin'(Mul', u, v)
```

```
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 : oexp → int
```

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

Why not just rely on the typechecker?

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

Why not just rely on the typechecker?

- We do automatically what the programmer must do manually.
- We guarantee that the program obtained is related to the original one
- 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 just rely on the typechecker?

- We do automatically what the programmer must do manually.
- We guarantee that the program obtained is related to the original one
- 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).

(Semi automated) code specialization

- Remove a field that is instanciated with `unit` ▷
- Represent several boolean fields on a single integer
- Switch to a representation that can be unboxed (`bool option`) ▷

Our goal

- Show that ornaments are a convenient tool for the ML programmer
- Design (the building blocks of) a language for meta-programming with ornamentation in ML
- Follow a composable approach, where ornamentation can be combined with other transformations, *e.g.* other forms of code inference, mixed with user interaction, *etc.*
- Lift ML programs to other ML programs
- Ensure that ornamentation is well-behaved
- Also an experiment in typed-based, user-driven code transformations.

our inspiration

Abstraction is our inspiration

Code reuse by abstraction *a priori* as a design principle, an easy case:

Polymorphic code or a function
$\Lambda(\alpha, \beta) \ \ldots \lambda(x : \tau, y : \sigma)\ M$

F

A

Specialize the code with
type and value arguments

$F\ A$

Abstraction is our inspiration

Code reuse by abstraction *a priori* as a design principle, an easy case:

Polymorphic code or a function
$\Lambda(\alpha, \beta)\ \ldots \lambda(x : \tau, y : \sigma)\ M$

F

A

B

Specialize the code with type and value arguments

$F\ A$

Reuse the code with other type and value arguments

$F\ B$

Abstraction is our inspiration

Code reuse by abstraction *a priori* as a design principle, an easy case:

Polymorphic code or a function
$\Lambda(\alpha, \beta)\ \ldots \lambda(x : \tau, y : \sigma)\ M$

F

A

B

Specialize the code with
type and value arguments

$F\ A$

Reuse the code with other
type and value arguments

$F\ B$

Theorems for free

Parametricity ensures that the code $F\ A$ and $F\ B$ behaves the same up to the differences between A and B.

Refactoring

base code

A

Refactoring

base code $\xrightarrow{?}$ Find its lifted version given an ornament specification

A $\qquad$ B

Refactoring by *abstraction a posteriori*

Find a generic version
$\Lambda(\alpha, \beta)\ \lambda(x : \tau)(y : \sigma)\ M$

A_{gen}

Inference
1

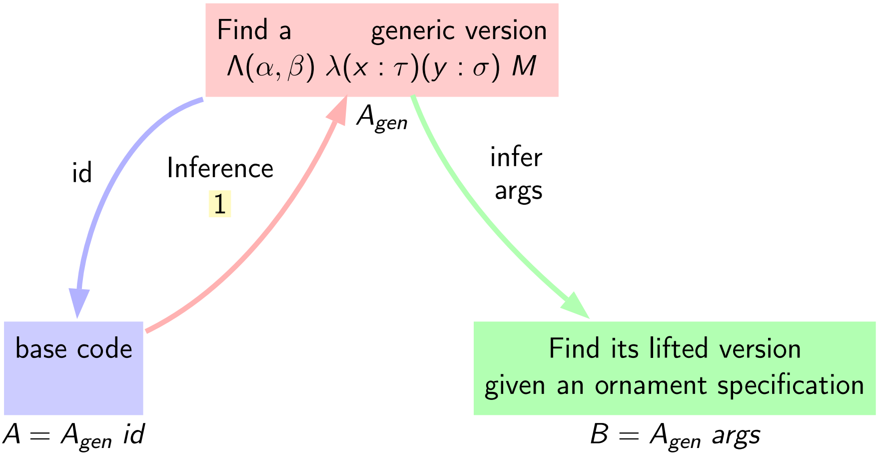
base code

A

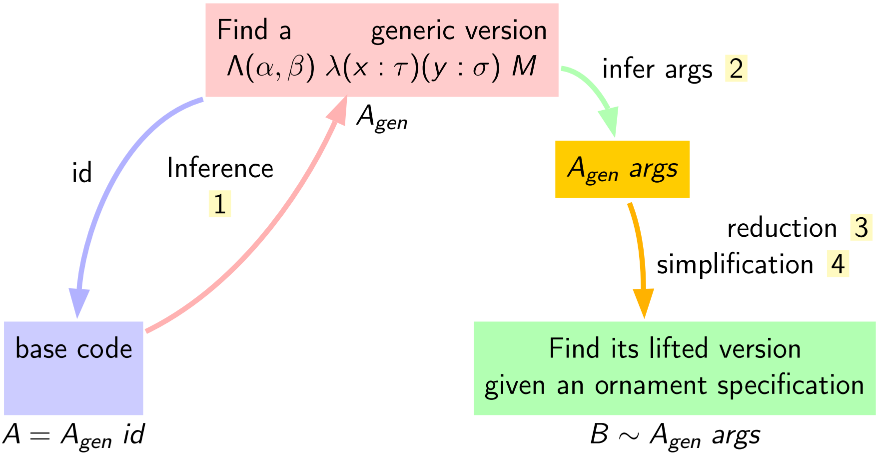
Find its lifted version
given an ornament specification

B

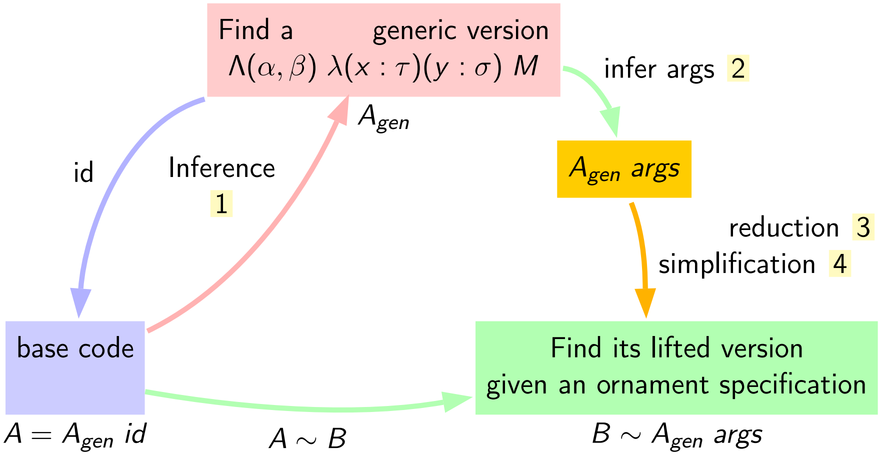
Refactoring by *abstraction a posteriori*



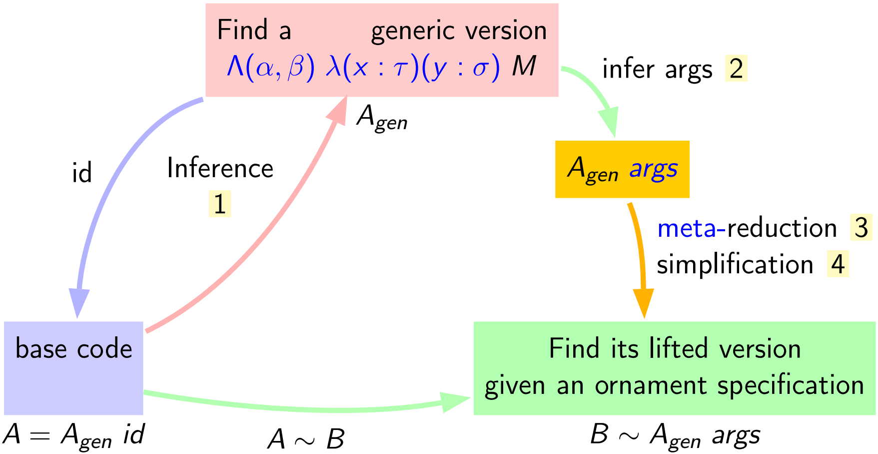
Refactoring by *abstraction a posteriori*



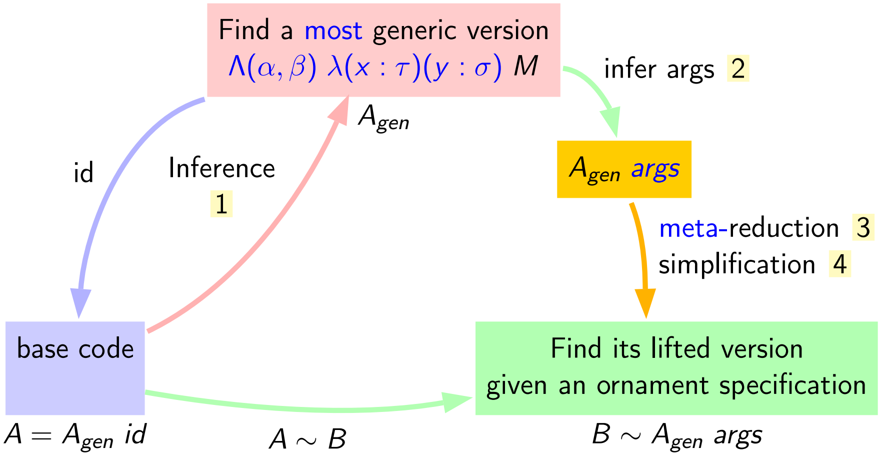
Refactoring by *abstraction a posteriori*



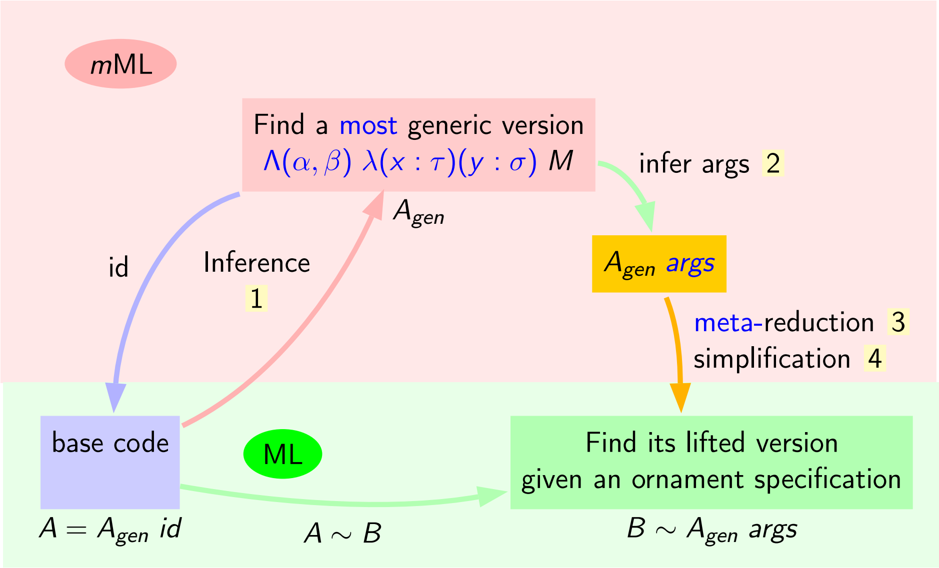
Refactoring by *abstraction a posteriori*



Refactoring by *abstraction a posteriori*



Refactoring by *abstraction a posteriori*



Questions & difficulties

The meta-language *m*ML must

- trace meta-reductions (easy by stratification)
- keep fine-grain invariants to ensure that it can be simplified to ML
- trace equalities between expressions for dead branches elimination
- have dependent types: type depends on pattern matching branches

The generic version

- depends solely on the source, not on the ornament (split of concerns)
- we restrict to the syntactic variants: we can only abstract over data-types that are explicit in the program (constructed or destructed)
- we abstract over all possible ornamentations of these data-types, *respecting their recursive structure*

Key Ideas from add to append. . .

- Introduce a skeleton (open definition) of nat, to allows for hybrid nats where the head looks like a nat but the tail need not be a nat.

  ```
  type α natS = Z' | S' of α
  ```

- Insert conversions between lists and natS in add to obtain append.

  ```
  let list2natS a = match a with
    | Nil → Z'
    | Cons(_,xs) → S' xs

  let natS2list n x = match n with
    | Z' → Nil
    | S' xs → Cons(x, xs)
  ```

Key Ideas from add to append...

- Introduce a skeleton (open definition) of nat, to allows for hybrid nats where the head looks like a nat but the tail need not be a nat.

```
type α natS = Z' | S' of α
```

- Insert conversions between lists and natS in add to obtain append.

```
let list2natS a = match a with
  | Nil → Z'
  | Cons(_,xs) → S' xs

let natS2list n x = match n with
  | Z' → Nil
  | S' xs → Cons(x, xs)

let rec append m n =
  match list2natS m with
    | Z' → n
    | S' m' → natS2list (S' (append m' n)) (List.hd m)
```

Key ideas ...and to a generic lifting

From add to add_gen: *abstract append over the ornament*

```
let append =
  let rec add m n =
    match list2natS m with
      | Z' → n
      | S' m' → natS2list (S' (add m' n)) (List.hd m)
  in add
```

Key ideas ...and to a generic lifting

From add to add_gen: *abstract append over the ornament*

```
let add_gen                                                    =
  let rec add m n =
    match m2natS    m with
      | Z' → n
      | S' m' → natS2n    (S' (add m' n)) (patch m n)
  in add
```

Key ideas ...and to a generic lifting

From add to add_gen: *abstract append over the ornament*

```
let add_gen m2natS                   natS2n patch =
  let rec add m n =
    match m2natS    m with
      | Z' → n
      | S' m' → natS2n    (S' (add m' n)) (patch m n)
  in add
```

Key ideas ... and to a generic lifting

From add to add_gen: *abstract append over the ornament*

```
let add_gen m2natS natS2m n2natS natS2n patch =
  let rec add m n =
    match m2natS    m with
      | Z' → n
      | S' m' → natS2n    (S' (add m' n)) (patch m n)
  in add
```

Key ideas ...and to a generic lifting

From add to add_gen: *abstract append over the ornament*

```
let add_gen m2natS natS2m n2natS natS2n patch =
  let rec add m n =
    match m2natS    m with
      | Z' → n
      | S' m' → natS2n    (S' (add m' n)) (patch m n)
  in add
```

From add_gen back to append

```
let append = add_gen list2natS natS2list list2natS natS2list
                     (fun m _ → match m with Cons(x,_) → x)
```

Key ideas … and to a generic lifting

From add to add_gen: *abstract append over the ornament*

```
let add_gen m2natS natS2m n2natS natS2n patch =
  let rec add m n =
    match m2natS     m with
      | Z' → n
      | S' m' → natS2n     (S' (add m' n)) (patch m n)
  in add
```

From add_gen back to append

```
let append = add_gen list2natS natS2list list2natS natS2list
                    (fun m _ → match m with Cons(x,_) → x)
```

From add_gen back to add: by passing the "identity" ornament

```
let nat2natS = function Z → Z' | S m → S' m
let natS2nat n x = match n with Z' → Z | S' m' → S m'
let add = add_gen nat2natS natS2nat nat2natS natS2nat
                 (fun _ _ → ())
```

Staging

We need to

- to generate readable code (the one the user would have written)
- preserve the computational behavior/complexity, not just the meaning
- bring the lifted code back to ML

Staging

We need to

- to generate readable code (the one the user would have written)
- preserve the computational behavior/complexity, not just the meaning
- bring the lifted code back to ML

Mark meta-abstractions and meta-applications that have been introduced:

```
let add_gen = fun m2natS natS2m n2natS natS2n patch →
  let rec add m n =
    match m2natS  m with
      | Z' -> n
      | S' m' -> natS2n  S' (add m' n)  patch m n
  in add

let append = add_gen  list2natS  natS2list  list2natS  natS2list
                      (fun m _ -> match m with Cons(x,_) -> x)
```

Staging

We need to

- to generate readable code (the one the user would have written)
- preserve the computational behavior/complexity, not just the meaning
- bring the lifted code back to ML

Mark meta-abstractions and meta-applications that have been introduced:

```
let add_gen = fun m2natS natS2m n2natS natS2n patch #>
  let rec add m n =
    match m2natS # m with
      | Z' -> n
      | S' m' -> natS2n # S' (add m' n) # patch m n
  in add

let append = add_gen # list2natS # natS2list # list2natS # natS2list
                     # (fun m _ -> match m with Cons(x,_) -> x)
```

Meta-reduction of the lifted code

```
let add_gen = fun m2natS natS2m n2natS natS2n patch #⇒
  let rec add m n =
    match m2natS # m with
      | Z' → n
      | S' m' → natS2n # S' (add m' n) # patch m n
  in add

let append = add_gen #list2natS# natS2list # list2natS #natS2list
                     # (fun m _ → match m with Cons(x,_) → x)
```

- Reduce #-redexes at compile time.
- All #-abstractions and #-applications can actually be reduced.
- This ensured by typing!

Meta-reduction of the lifted code

```
let add_gen = fun m2natS natS2m n2natS natS2n patch ⇏
  let rec add m n =
    match m2natS # m with
      | Z' → n
      | S' m' → natS2n # S' (add m' n) # patch m n
  in add

let append = add_gen # list2natS # natS2list # list2natS # natS2list
                     # (fun m _ → match m with Cons(x,_) → x)
```

- Reduce #-redexes at compile time.
- All #-abstractions and #-applications can actually be reduced.
- This ensured by typing!

Meta-reduction

```
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 some redundant pattern matchings...
- Decoding list to natS and encoding natS to list .
- We can eliminate the last one by reduction

Elimination of the encoding

```
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)
```

Eliminating the encoding

```
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)
```

Eliminating the encoding

```
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)
```

- And the other by extrusion... (commuting matches)

Eliminating the encoding

```
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))
```

Eliminating the encoding

```
let rec append m n =
  match m with
    | Nil →

              n

    | Cons(_, xs) →

                      Cons(List.hd m, append m’ n))
```

and reducing again

Back to ML

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

Back to ML

```
let rec append m n =
  match m with
    | Nil → n
    | Cons (x, xs)
          → Cons ((match m with Cons x → x), append m' n)
```

Back to ML

```
let rec append m n =
  match m with
    | Nil → n
    | Cons (x, xs)
          → Cons (x, append m' n)
```

- We obtain the code for append.
- This transformation also eliminates all our uses of dependent types.
- This is always the case

In practice

We have a prototype implementation

- It follows the process outlined here.
- User interface issues: for specifying the instantiation, we take labelled patches and ornaments.
- To build the generic lifting, we transform deep pattern matching into shallow pattern matching.
- We try to recover the shape of the original program in a post-processing phase, keeping sharing annotations during dupplication
- We also expand local polymorphic lets (only a user interface problem)

See `http://gallium.inria.fr/~remy/ornaments/`

Goal: next version of the prototype for OCaml to run larger examples.

Discussion

Effects

- We use call-by-value, carefully preserving the evaluation order
- Should work without surpise in the presence of effects.
- A formal result about effects?

Recursion

- Modifying the recursive structure. Allowing mutual recursion.
- Non-regular types. GADTs.

Patches

- Can we write robust patches (that resist to code transformations)?
- Combine with some form of code inference (for patches)

Questions

- Should we give the user access to the intermediate language *m*ML?
- Can we use *m*ML for other purposes?

Take away

About ornaments

- Ornaments are useful in ML, both for software reuse and *evolution*
- Going from the source program to the target program via a generic lifting that is later instantiated seems the right approach:
  - correctness by parametricity.
  - also allows to represent partially instantiated terms (user interface)
- We can even generate user-readable code!

Software evolution

- Ornaments are one way of doing software evolution.
- Software evolution via abstraction *a posteriori* seems a good principle, with other potential applications.
- Typed languages are a good setting for software evolution/refactoring that we should also explore further.

Outline

Outline

1. Dependent types
2. A meta-language for ornamentation
3. Encoding ornaments in mML
4. More examples

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 natS2stream 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 natS2stream n x = match n with
    | Z' → (match x with
                | true → Continued
                | false → End)
    | S' n' → More (x, n')
```

What is the type of natS2stream ?

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 natS2stream n x = match n with
    | Z' → (match x with
                | true → Continued
                | false → End)
    | S' n' → More (x, n')
```

What is the type of x ?

$$(\text{match } x \text{ with Z}' \to \text{unit} \mid \text{S}' \ \_ \to \alpha)$$

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 natS2stream n x = match n with
    | Z' → (match x with
              | true → Continued
              | false → End)
    | S' n' → More (x, n')
```

What is the type of natS2stream ?

$$\lambda^\sharp \alpha.\ \Pi(x : \mathsf{natS}\ (\mathsf{list}\ \alpha)).\ \Pi(y : (\mathsf{match}\ x\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha)).\ \mathsf{list}\ \alpha$$

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 natS2stream n x = match n with
    | Z' → (match x with
                | true → Continued
                | false → End)
    | S' n' → More (x, n')
```

What is the type of natS2stream ?

$$\begin{aligned}&\lambda^\sharp\alpha.\,\Pi(x : \mathsf{natS}\ (\mathsf{list}\ \alpha)).\\&\qquad\Pi(y : (\mathsf{match}\ x\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha)).\ \mathsf{list}\ \alpha\end{aligned}$$

Dependent types we introduce can always be eliminated.

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)
```

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

Outline

Starting from ML

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

Starting from ML

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

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

$$\text{Context-Beta}\quad \frac{a \;\longrightarrow_\beta\; b}{E[a] \longrightarrow_\beta E[b]}$$

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.

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

Consider:

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

eML

eML is obtained by extending the type system of ML.

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

Consider:

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

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

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

Equalities

We extend the typing environment with equalities:

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

Introduced on pattern matching

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

Used to prove *type* equalities

Since terms appears in types, they generate equalities on types, which allows for *implicit* conversions:

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

Elimination of equalities

We restrict reduction in equalities so that it remains decidable.

Assume a is term an *e*ML a 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 equalities to substitute in terms
- Extruding nested pattern matching
- Reduding pattern matching

This justifies the use of *e*ML as an intermediate language for ornamentation

Meta-programming in mML

We introduce a separate type for meta-functions, so that they can only be applied using meta-application.

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

This enables to eliminate all abstractions and applications marked with **#**.

We restrict types so that 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

We need dependent types for the encoding function:

$$
\begin{aligned}
\mathsf{natS2list} : \quad & \lambda^{\sharp}\alpha.\ \Pi(x : \mathsf{natS}\ (\mathsf{list}\ \alpha)).\\
& \Pi(y : \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha).\\
& \mathsf{list}\ \alpha
\end{aligned}
$$

Dependent functions

We need dependent types for the encoding function:

$$\begin{aligned}\mathsf{natS2list} :\quad & \lambda^\sharp\alpha.\ \Pi(x : \mathsf{natS}\ (\mathsf{list}\ \alpha)).\\ & \Pi(y : \mathsf{match}\ x\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha).\\ & \mathsf{list}\ \alpha\end{aligned}$$

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 $\rightarrow$ α natlist $\rightarrow$ α natlist

We mean:

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

Semantics of ornament specifications

let append = **lifting** add **:** α natlist $\rightarrow$ α natlist $\rightarrow$ α natlist

We mean:

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

We build a (step-indexed) binary logical relation on *m*ML, and add an interpretation for datatype ornaments.

The interpretation of a functional lifting is exactly the interpretation of function types, replacing "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)
```

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

Datatype ornaments

A datatype ornament naturally gives a relation:

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

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

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.

Outline

Specialization: unit map

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

Specialization: unit map

```
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
```