Two very similar functions

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

\[
\text{let rec } \text{ append } ml \ nl = \text{ match } ml \ \text{ with} \\
\ | \ \text{Nil} \rightarrow nl \\
\ | \ \text{Cons}(x,ml') \rightarrow \text{Cons}(x,\text{append } ml' \ nl)
\]
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

\[
\begin{align*}
\text{type} & \quad \text{nat} = Z \mid S \text{ of } \text{nat} \\
\text{type} & \quad \alpha \text{ list} = \text{Nil} \mid \text{Cons of } \alpha \times \alpha \text{ list}
\end{align*}
\]

\[
S ( S ( S ( Z )))
\]

Cons(1, Cons(2, Cons(3, Nil)))

Projection function

\[
\begin{align*}
\text{let rec} & \quad \text{length} = \text{function} \\
& \quad | \quad \text{Nil} \rightarrow Z \\
& \quad | \quad \text{Cons}(x, xs) \rightarrow S(\text{length} \ xs)
\end{align*}
\]

The relation between \(\text{nat}\) and \(\alpha\) \(\text{list}\) 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

```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)
```

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

```ml
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

\[
\begin{align*}
\textbf{type} & \quad \text{nat} = Z \mid S \, \text{of} \, \text{nat} \\
\textbf{type} & \quad \alpha \, \text{list} = \text{Nil} \mid \text{Cons \, of} \, \alpha \times \alpha \, \text{list}
\end{align*}
\]
add/append again

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

\[
\text{type ornament } \alpha \text{ natlist : nat } \rightarrow \alpha \text{ list with} \\
\mid Z \rightarrow \text{Nil} \\
\mid S \ x_\text{s} \rightarrow \text{Cons(\_., } x_\text{s})
\]
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**

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

**Type ornament**

\[
\text{type ornament} \; \alpha \; \text{natlist} : \text{nat} \to \alpha \; \text{list} \; \text{with} \\
\quad | \; Z \to \text{Nil} \\
\quad | \; S \; \text{x} \to \text{Cons}(\_, \text{x}) \\
\]

**Let rec**

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

\[
\begin{align*}
\text{let} \; \text{append} &= \text{lifting add} : \_ \; \text{natlist} \to \_ \; \text{natlist} \to \_ \; \text{natlist} \\
\end{align*}
\]

**Let rec**

\[
\begin{align*}
\text{let rec} \; \text{append} \; \text{ml} \; \text{nl} &= \text{match ml with} \\
\quad | \; \text{Nil} \to \text{nl} \\
\quad | \; \text{Cons}(\text{x},\text{ml'}) \to \text{Cons}(\#1, \text{append ml' nl}) \\
\end{align*}
\]
add/append again

```ocaml
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

```plaintext
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

```haskell
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)
```
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 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

```ocaml
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

```plaintext
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

```plaintext
type bool = False \mid True
```

We can safely exchange True and False in some places:

```plaintext
type ornament not : bool \rightarrow bool with
  \mid True \rightarrow False
  \mid False \rightarrow True
```

The relations between bare and ornamented values are tracked through the program (by *ornament* inference).
The ornamentation constraints are propagated as with type inference.

```ocaml
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

\[
\text{type } \alpha \text{ map } = \\
| \text{Node of } \alpha \text{ map } \times \text{key } \times \alpha \times \alpha \text{ map } \\
| \text{Leaf}
\]
Specialization

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

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

definition type set =
    | SNode of set × key × set
    | SLeaf
Specialization

\[
\text{type } \alpha \text{ map } = \\
| \text{Node of } \alpha \text{ map } \times \text{key } \times \alpha \times \alpha \text{ map} \\
| \text{Leaf}
\]

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

\[
\text{type } \text{set } = \\
| \text{SNode of } \text{set } \times \text{key } \times \text{set} \\
| \text{SLeaf}
\]

\[
\text{type } \text{ornament mapset } : \text{unit map } \rightarrow \text{set with} \\
| \text{Node(l,k,(),r) } \rightarrow \text{SNode(l,k,r)} \\
| \text{Leaf } \rightarrow \text{SLeaf}
\]
Specialization: unboxing

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

type booloption =
  | NoneBool
  | SomeTrue
  | SomeFalse
```
Specialization: unboxing

```haskell
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:

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

```ocaml
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:

```ocaml
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:

```ocaml
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.

```plaintext
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 = 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.

```plaintext
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`.

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

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)
```

```ocaml
type α nat_skel = Z’ | S’ of α

let list2nat’ a
```

```ocaml
let nat’2 list n x
```

```ocaml
let rec append m n =
```

```ocaml
match list2nat’ m with
```

```ocaml
| Z’ → n
```

```ocaml
| 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 #.

\[
\text{type } \alpha \text{ nat_skel } = Z' \mid S' \text{ of } \alpha
\]

\[
\text{let list2nat’ } = \text{fun } l \mapsto \text{match } l \text{ with}
\]
\[
| \text{Nil } \rightarrow Z'
\]
\[
| \text{Cons}(_,xs) \rightarrow S' \text{ xs}
\]

\[
\text{let nat’2 list } = \text{fun } n x \mapsto \text{match } n \text{ with}
\]
\[
| Z' \rightarrow \text{Nil}
\]
\[
| S' \text{ xs } \rightarrow \text{Cons(x, xs)}
\]

\[
\text{let rec append m n } =
\]
\[
\text{match list2nat’ # m with}
\]
\[
| Z' \rightarrow n
\]
\[
| S' \text{ m’ } \rightarrow \text{nat’2list # (S’ (append m’ n)) # (List.hd m)}
\]
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 #.

```ocaml
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)
```

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:

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

There remains two redundant pattern matchings, decoding lists to `nat_skel` and encoding `nat_skel` to lists.
Eliminating ornamentation calls

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

```ml
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:

```ocaml
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 \texttt{nat\_skel} and encoding \texttt{nat\_skel} to lists. We can eliminate them by reduction:

\[
\text{let rec append } m \ n = \\
\text{match (match } m \text{ with}
\begin{align*}
\mid \text{Nil} & \rightarrow Z' \\
\mid \text{Cons(\_\_, } xs) & \rightarrow S' \ xs \\
\mid Z' & \rightarrow n \\
\mid S' \ m' & \rightarrow \text{Cons(List.hd } m, \ \text{append } m' \ n)
\end{align*}
\]
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:

```ml
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:

```ml
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

```ocaml
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

```ocaml
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_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?

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

What is the type of \( x \)?
The case for dependent types

What if we add data to the Z constructor too?

```latex
\text{type } \alpha \text{ stream } = \text{End} \mid \text{Continued} \mid \text{More of } \alpha \times \alpha \text{ stream}
```

```latex
\text{ornament } \alpha \text{ natstream } : \text{nat } \rightarrow \alpha \text{ stream with}
```

```latex
\mid Z \rightarrow (\text{End} \mid \text{Continued})
\mid S \ n \rightarrow \text{More}(\_, \ n)
```

```latex
\text{let } \text{nat}^{\prime}2\text{stream } n \times =
```

```latex
\text{match } n \text{ with }
```

```latex
\mid Z' \rightarrow (\text{match } x \text{ with }
```

```latex
\mid \text{true } \rightarrow \text{Continued}
\mid \text{false } \rightarrow \text{End})
```

```latex
\mid S' \ n' \rightarrow \text{More}(x, n')
```

What is the type of $x$?
The case for dependent types

What if we add data to the Z constructor too?

\[
\text{type } \alpha \text{ stream } = \text{End} \mid \text{Continued} \mid \text{More of } \alpha \times \alpha \text{ stream}
\]

\[
\text{ornament } \alpha \text{ natstream } : \text{nat } \to \alpha \text{ stream with}
\]

\[
| \text{Z } \to (\text{End} \mid \text{Continued})
| \text{S } n \to \text{More}(\_, n)
\]

\[
\text{let } \text{nat'}2\text{stream } n \times =
\]

\[
\text{match } n \text{ with}
| \text{Z'} \to (\text{match } x \text{ with}
  | \text{true } \to \text{Continued}
  | \text{false } \to \text{End})
| \text{S'} n' \to \text{More}(x, n')
\]

What is the type of \(x\)?
The case for dependent types

What if we add data to the Z constructor too?

```plaintext
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?

```plaintext
match n with Z' → bool | S' _ → α
```
The case for dependent types

The type may depend on more than the constructor.

```plaintext
type \( \alpha \) list01 =
| Nil01
| Cons0 of \( \alpha \) list01
| Cons1 of \( \alpha \times \alpha \) list01

ornament \( \alpha \) olist01 : bool list \( \rightarrow \) \( \alpha \) list01 with
| Nil \( \rightarrow \) Nil01
| Cons(False, xs) \( \rightarrow \) Cons0(xs)
| Cons(True, xs) \( \rightarrow \) Cons1(\_, xs)
```
The case for dependent types

The type may depend on more than the constructor.

\[
\text{type } \alpha \text{ list01 } = \\
| \text{Nil01} \\
| \text{Cons0 of } \alpha \text{ list01} \\
| \text{Cons1 of } \alpha \times \alpha \text{ list01}
\]

\[
\text{ornament } \alpha \text{ olist01 : bool list } \rightarrow \alpha \text{ list01 with} \\
| \text{Nil } \rightarrow \text{Nil01} \\
| \text{Cons(False, xs)} \rightarrow \text{Cons0(xs)} \\
| \text{Cons(True, xs)} \rightarrow \text{Cons1(\_, xs)}
\]

\[
\text{match } m \text{ with} \\
| \text{Nil' } \rightarrow \text{unit} \\
| \text{Cons'(False, \_)} \rightarrow \text{unit} \\
| \text{Cons'(True, \_)} \rightarrow \alpha
\]
Outline

Some examples

Encoding lifting

A meta-language for ornamentation

Encoding ornaments in mML
Starting from ML

\[\begin{align*}
\tau, \sigma &::= \alpha \mid \tau \rightarrow \tau \mid \zeta \bar{\tau} \mid \forall (\alpha : \text{Typ}) \tau \\
a, b &::= x \mid \text{let } x = a \text{ in } a \mid \text{fix } (x : \tau) x. a \mid a a \\
&\quad \mid \Lambda (\alpha : \text{Typ}). u \mid a \tau \mid d \bar{\tau} \bar{a} \mid \text{match } a \text{ with } \underline{P \rightarrow a} \\
P &::= d \bar{\tau} \bar{x}
\end{align*}\]
Starting from ML

\[
E ::= [] | E\ a | v\ E | d(v,\ldots,v,E,a,\ldots,a) | \Lambda(\alpha : \text{Typ}).\ E | E\ \tau \\
| \text{match}\ E\ \text{with}\ P \rightarrow a | \text{let}\ x = E\ \text{in}\ a
\]

\[
(fix (x : \tau) y.\ a)\ v \rightarrow^h_{\beta} a[x \leftarrow \text{fix} (x : \tau) y.\ a, y \leftarrow v]
\]

\[
(\Lambda(\alpha : \text{Typ}).\ v)\ \tau \rightarrow^h_{\beta} v[\alpha \leftarrow \tau]
\]

\[
\text{let}\ x = v\ \text{in}\ a \rightarrow^h_{\beta} a[x \leftarrow v]
\]

\[
\text{match}\ d_j \overrightarrow{\tau_j} (v_i)^i\ \text{with}\ (d_j \overrightarrow{\tau_j} (x_{ji})^i \rightarrow a_j)^j \rightarrow^h_{\beta} a_j[x_{ij} \leftarrow v_i]^i
\]

Context-Beta

\[
a \rightarrow^h_{\beta} b
\]

\[
E[a] \rightarrow^\beta E[b]
\]
From ML to $m$ML

- eML: 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 eML
- From eML, eliminate type-level pattern matching and get a term in ML
eML

eML is obtained by extending the type system of ML.
eML

eML is obtained by extending the type system of ML.

\[ \Gamma = \alpha : \text{Typ}, \, m : \text{nat'} (\text{list } \alpha), \, x : \text{match } m \text{ with } Z' \rightarrow \text{unit} \mid S' \_ \rightarrow \alpha \]

\[
\begin{align*}
\text{match } m \text{ with} \\
| \ Z' \rightarrow \text{Nil} \\
| S' \ m' \rightarrow \text{Cons} (x, m')
\end{align*}
\]
eML

eML is obtained by extending the type system of ML.

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

\[
\text{match } m \text{ with } \\
\mid Z' \to \text{Nil} \\
\mid S' \ m' \to \text{Cons (} x, m' \text{)}
\]

In the \( S' \) branch: we know \( m = S' \ m' \). Thus:

\[
\begin{align*}
x & : \text{match } m \text{ with } Z' \to \text{unit } \mid S' \_ \to \alpha \\
& = \text{match } S' \ m' \text{ with } Z' \to \text{unit } \mid S' \_ \to \alpha \\
& = \alpha
\end{align*}
\]
Introducing equalities

We extend the typing environment with equalities:

\[ \Gamma ::= \ldots | \Gamma, a =_\tau b \]
Introducing equalities

We extend the typing environment with equalities:

\[ \Gamma ::= \ldots \mid \Gamma, a =_\tau b \]

Equalities are introduced on pattern matching:

\[
\begin{align*}
\Gamma \vdash \tau : \text{Sch} & \quad (d_i : \forall(\alpha_k : \text{Typ})^k (\tau_{ij})^j \rightarrow \zeta (\alpha_k)^k)^i \\
\Gamma \vdash a : \zeta (\tau_k)^k & \\
(\Gamma, (x_{ij} : \tau_{ij}[\alpha_k \leftarrow \tau_k]^k)^j, a = \zeta (\tau_k)^k d_i (\tau_{ij})^k (x_{ij})^j \vdash b_i : \tau)^i & \\
\Gamma \vdash \text{match } a \text{ with } (d_i (\tau_{ik})^k (x_{ij})^j \rightarrow b_i)^i : \tau
\end{align*}
\]
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:

$$\Gamma \vdash \tau_1 \simeq \tau_2 \quad \Gamma \vdash a : \tau_1 \quad \frac{}{\Gamma \vdash a : \tau_2}$$
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 $\Gamma$ and $\tau$ 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
- Reducing pattern matching

Thus it makes sense to use eML as an intermediate language for ornamentation.
Meta-programming in \( mML \)

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.

\[
(\lambda^\#(x: \tau). \, a)^\# \, u \rightarrow^h_{\#} \, a[x \leftarrow u]
\]

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 $\rightarrow^\#$ will eliminate all meta constructions and give an eML 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 eML and that we need to encode ornaments.

We need dependent types for the encoding function:

\[
\text{nat'}\_\to\_\text{list} : \prod(x : \text{nat'} (\text{list } \alpha)). \\
\quad \prod(y : \text{match } x \text{ with } Z' \to \text{unit} \mid S' \_ \to \alpha). \\
\quad \text{list } \alpha
\]
Dependent functions

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

We need dependent types for the encoding function:

\[
\text{nat'}_{\text{to_list}} : \Pi(x : \text{nat'} (\text{list } \alpha)). \Pi(y : \text{match } x \text{ with } Z' \rightarrow \text{unit} \mid S' _\_ \rightarrow \alpha). \text{list } \alpha
\]

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

Some examples

Encoding lifting

A meta-language for ornamentation

Encoding ornaments in mML
Semantics of ornament specifications

\[ \text{let append} = \text{lifting add} : \alpha \text{ natlist} \rightarrow \alpha \text{ natlist} \rightarrow \alpha \text{ natlist} \]

What we mean:

- If \( ml \) is a lifting of \( m \) (for \( \text{natlist} \) )
- and \( nl \) is a lifting of \( n \)
- then \( \text{append} \ ml \ nl \) is a lifting of \( \text{add} m n \)
Semantics of ornament specifications

```plaintext
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 mML, 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:

\[
\text{ornament } \alpha \text{ natlist } : \text{ nat } \to \alpha \text{ list } \text{ with }
\]

| Z  → Nil  \\
| S xs → Cons(_, xs)  

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.
Datatype ornaments

A datatype ornament naturally gives a relation:

\[\text{ornament } \alpha \text{ natlist : nat } \rightarrow \alpha \text{ list with} \]
\[| \ Z \rightarrow \text{Nil} \]
\[| S \ xs \rightarrow \text{Cons}(_{,} , xs)\]

\[(Z, \text{Nil}) \in \mathcal{V}_k[\text{natlist } \tau]\]

\[\frac{(u, \nu) \in \mathcal{V}_k[\text{natlist } \tau]}{(S \ u, \text{Cons} (a, \nu)) \in \mathcal{V}_k[\text{natlist } \tau]}\]
Datatype ornaments

A datatype ornament naturally gives a relation:

\[ \text{ornament } \alpha \text{ natlist : nat } \rightarrow \alpha \text{ list with } \]
\[ | \ Z \rightarrow \text{Nil} \]
\[ | \ S \ xs \rightarrow \text{Cons}(_{\_}, \ xs) \]

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

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 $\gamma_+$ and the identity instanciation $\gamma_-$. 
- $\gamma_-(a)$ 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 interface 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 mML?
- Can we use mML for something else?