Ornamentation put into practice in ML

Didier Rémy

based on joined work with

Thomas Williams  and  Lucas Baudin

MFSP 2018

ML

OCaml

Haskell

ML ⋯ OCaml ⋯ Haskell

All have in common. . .

- Datatypes & Pattern-matching
- Polymorphism
- Type inference
- First-class functions

ML .... OCaml
.... Haskell

All have in common...

- Datatypes & Pattern-matching
- Polymorphism
- Type inference
- First-class functions

Therefore,

- Programs are safer by construction
  (and Haskell ones perhaps even more...)
- Still, they sometimes need to be modified...

ML .... OCaml
.... Haskell

All have in common...

- Datatypes & Pattern-matching
- Polymorphism
- Type inference
- First-class functions

Therefore,

- Programs are safer by construction
  (and Haskell ones perhaps even more...)
- Still, they sometimes need to be modified...

Program refactoring and evolution

- Surprisingly, it has been little explored by our communities
- But there are interesting things we can do, thanks to
  - programs being structured around datatypes
  - polymorphism and type inference.

In this talk

- A restricted form of code refactoring and code refinement based on ornaments can be put into practice in ML.
- This can be seen as code generalization a posteriori
- . . . and formalized using logical relations (in a richer language).
- Ornamentation generalizes to its inverse transformation, disornamentation with interesting applications.

In this talk

- A restricted form of code refactoring and code refinement based on ornaments can be put into practice in ML.
- This can be seen as code generalization a posteriori
- …and formalized using logical relations (in a richer language).
- Ornamentation generalizes to its inverse transformation, disornamentation with interesting applications.

Notes

- *Ornaments have been introduced by Conor McBride and explored widely with Pierre-Évariste Dagan in the context of Agda and also by Jeremy Gibbons and Hsiang-Shang Ko.*
- *Our approach is more syntactic, our goal being to bring ornaments-based program transformations to the ML programmer.*

The poor man's (good) tool

The poor man's (good) tool

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

The poor man's (good) tool

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

The poor man's (good) tool

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

```
let parse x = Add' (x, Con' 42)
```

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

The poor man's (good) tool

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

let parse x = Add' (x, Con' 42)
let rec eval e = match e with
  | Con' i → i
  | Add' (u, v) →
        add (eval' u) (eval v)
  | Mul' (u, v) →
        mul (eval u) (eval v)
```

The poor man's (good) tool

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

let parse x = Bin'(Add', x, Con' 42)
let rec eval e = match e with
  | Con' i → i
  | Add' (u, v) →
        add (eval' u) (eval v)
  | Mul' (u, v) →
        mul (eval u) (eval v)
```

The poor man's (good) tool

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

let parse x = Bin'(Add', x, Con' 42)
let rec eval e = match e with
  | Con' i → i
  | Bin'(Add', u, v) →
        add (eval u) (eval v)
  | Mul' (u, v) →
        mul (eval u) (eval v)
```

The poor man's (good) tool

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

let parse x = Bin'(Add', x, Con' 42)
let rec eval e = match e with
  | Con' i → i
  | Bin'(Add', u, v) →
        add (eval u) (eval v)
  | Bin'(Mul', u, v) →
        mul (eval u) (eval v)
```

The poor man's (good) tool

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

let parse x = Bin'(Add', x, Con' 42)
let rec eval e = match e with
  | Con' i → i
  | Bin'(Add', u, v) →
        add (eval u) (eval v)
  | Bin'(Mul', u, v) →
        mul (eval u) (eval v)
```

However

- We have to do manually what could be done automatically
- This may be long – and error prone !
- We should guarantee that the input and output programs are related
- We may miss places where a change is necessary (when types agree)

Can we do better?

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

Can we do better?

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

```
type relation oexp : exp ⇒ exp' with
  | Con i        ⇒  Con' i
  | Add(u, v)    ⇒  Bin'(Add', u, v)      when u v : oexp
  | Mul(u, v)    ⇒  Bin'(Mul', u, v)      when u v : oexp
```

Can we do better?

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

```
type relation oexp : exp ⇒ exp' with
  | Con i       ⇒ Con' i
  | Add(u, v)   ⇒ Bin'(Add', u, v)      when u v : oexp
  | Mul(u, v)   ⇒ Bin'(Mul', u, v)      when u v : oexp
```

Can we do better?

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

```
type relation oexp : exp ⇒ exp' with
  | Con i       ⇒  Con' i
  | Add(u, v)   ⇒  Bin'(Add', u, v)     / when oexp : u : exp ⇒ u : exp'
  | Mul(u, v)   ⇒  Bin'(Mul', u, v)     \  and oexp : v : exp ⇒ v : exp'
```

Can we do better?

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

```
type relation oexp : exp ⇒ exp' with
  | Con i        ⇒  Con' i
  | Add(u, v)    ⇒  Bin'(Add', u, v)      when u v : oexp
  | Mul(u, v)    ⇒  Bin'(Mul', u, v)      when u v : oexp

lifting * with oexp
```

blue + *red* $\Longrightarrow$ *green*

Can we do better?

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

```
let parse x = Bin'(Add', x, Con' 42)
let rec eval e = match e with
  | Con' i → i
  | Bin'(Add', u, v) →
        add (eval u) (eval v)
  | Bin'(Mul', u, v) →
        mul (eval u) (eval v)
```

```
type relation oexp : exp ⇒ exp' with
  | Con i       ⇒  Con' i
  | Add(u, v)   ⇒  Bin'(Add', u, v)      when u v : oexp
  | Mul(u, v)   ⇒  Bin'(Mul', u, v)      when u v : oexp

lifting * with oexp
```

blue + *red*
$\Longrightarrow$ *green*

Can we do better? (reversed)

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

let parse x = Add (x, Con 42)
let rec eval e = match e with
  | Con i → i
  | Add (u, v) → add (eval u) (eval v)
  | Mul (u, v) → mul (eval u) (eval v)
```

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

let parse x = Bin'(Add', x, Con' 42)
let rec eval e = match e with
  | Con' i → i
  | Bin'(Add', u, v) →
        add (eval u) (eval v)
  | Bin'(Mul', u, v) →
        mul (eval u) (eval v)
```

```
type relation oexp : exp' ⇒ exp with
  | Con' i            ⇒ Con i
  | Bin'(Add', u, v)  ⇒ Add(u, v)      when u v : oexp
  | Bin'(Mul', u, v)  ⇒ Mul(u, v)      when u v : oexp

lifting * with oexp
```

blue + *red* ⟹ *green*

Enforcing more invariants

```
type exp =
  | Con of int
  | Abs of (exp → exp)
  | App of exp × exp
```

```
let rec eval e = match e with
  | Con i → Some (Con i)
  | Abs f → Some (Abs f)
  | App (u, v) →
    (match eval u with
      | Some (Con i) → None
      | Some (Abs f) →
        (match eval v with
          Some x → eval (f x) | ..))
      | Some (App (u, v)) → None
      | None → None
```

Enforcing more invariants

```
type exp =
  | Con of int
  | Abs of (exp → exp)
  | App of exp × exp
```

```
let rec eval e = match e with
  | Con i → Some (Con i)
  | Abs f → Some (Abs f)
  | App (u, v) →
    (match eval u with
      | Some (Con i) → None
      | Some (Abs f) →
        (match eval v with
          Some x → eval (f x) | ..))
      | Some (App (u, v)) → None
      | None → None
```

```
type exp' =
  | Val of value'
  | App' of exp' × exp'
and value' =
  | Con' of int
  | Abs' of (value' → exp')
```

Enforcing more invariants

```
type exp =
  | Con of int
  | Abs of (exp → exp)
  | App of exp × exp
```

```
let rec eval e = match e with
  | Con i → Some (Con i)
  | Abs f → Some (Abs f)
  | App (u, v) →
    (match eval u with
      | Some (Con i) → None
      | Some (Abs f) →
        (match eval v with
          Some x → eval (f x) | ..))
      | Some (App (u, v)) → None
      | None → None
```

```
type exp' =
  | Val of value'
  | App' of exp' × exp'
and value' =
  | Con' of int
  | Abs' of (value' → exp')
```

```
let rec eval' e = match e with
  | Con' i → Some (Int i)
  | Abs' f → Some (Fun f)
  | App'(u, v) →
    (match eval' u with
      | Some (Con' i) → None
      | Some (Abs' f) →
        (match eval' v with
          Some x → eval' (f x) | ..))
      | None → None
```

Enforcing more invariants

```
type exp =
  | Con of int
  | Abs of (exp → exp)
  | App of exp × exp
```

```
type exp' =
  | Val of value'
  | App' of exp' × exp'
and value' =
  | Con' of int
  | Abs' of (value' → exp')
```

```
type relation oexp : exp ⇒ exp' with
  | Con i      ⇒ Val (Con' i)
  | Abs f      ⇒ Val (Abs' f)   when f : ovalue → oexp
  | App (u,v)  ⇒ App' (u, v)    when u v : oexp
and ovalue : exp ⇒ value' with
  | Con i      ⇒ Con' i
  | Abs f      ⇒ Abs' f         when f : ovalue → oexp
  | App (u,v)  ⇒ ~
```

indicates an impossible case

Porting operations on lists to tuples

Scenario

- Operations on lists are already implemented in a library
- Need for large homogeneous tuples
- Use lists for convenience.
- For efficiency (and safety) reasons, rewrite the code to use tuples

This can be automated

Porting operations on lists to tuples

```
type α list = Nil | Cons of (α × α list )
let rec map f z = match z with
  | Nil → Nil
  | Cons(x, t) → Cons(f x, map f t)
```

```
type unit = U
type α triple =
  T of α × (α × (α × unit))
```

Porting operations on lists to tuples

```
type α list = Nil | Cons of (α × α list )
let rec map f z = match z with
  | Nil → Nil
  | Cons(x, t) → Cons(f x, map f t)
```

```
type unit = U
type α triple =
  T of α × (α × (α × unit))
```

```
type relation α list_triple : α list ⇒ α triple with
  | Cons (x₁ , Cons (x₂ , Cons (x₃ , Nil ))) ⇒ T (x₁ , (x₂ , (x₃ , U)))
```

Simplified

Porting operations on lists to tuples

```
type α list = Nil | Cons of (α × α list)
let rec map f z = match z with
  | Nil → Nil
  | Cons(x, t) → Cons(f x, map f t)
```

```
type unit = U
type α triple =
  T of α × (α × (α × unit))
```

Simplified

```
type relation α list_triple : α list ⇒ α triple with
  | Cons (x₁, Cons (x₂, Cons (x₃, Nil))) ⇒ T (x₁, (x₂, (x₃, U)))
let map_tuple = lifting map : (α → β) → α list_triple → β list_triple
```

Porting operations on lists to tuples

```
type α list = Nil | Cons of (α × α list )
let rec map f z = match z with
  | Nil → Nil
  | Cons(x, t) → Cons(f x, map f t)
```

```
type unit = U
type α triple =
  T of α × (α × (α × unit))
```

Simplified

```
type relation α list_triple : α list ⇒ α triple with
  | Cons (x1 , Cons (x2 , Cons (x3 , Nil ))) ⇒ T (x1 , (x2 , (x3 , U)))
let map_tuple = lifting map : (α → β) → α list_triple → β list_triple
```

Automatically unfolding the recursion. . .

With manual inlining

```
let rec map_tuple f z =
  match z with T(x1 , x2 , x3 , U) → T(f x1 , f x2 , f x3 , U)
```

```
type α gen =
  | Pair  of (α gen × α gen)
  | Value of α
  | Unit
```

```
let rec map f z = match z with
  | Pair (u, v) → Pair (map f u, map f v)
  | Value x     → Value (f x)
  | Unit        → Unit
```

Generic programming *Lists*

```
type α gen =
  | Pair  of (α gen × α gen)
  | Value of α
  | Unit
```

```
let rec map f z = match z with
  | Pair (u, v) → Pair (map f u, map f v)
  | Value x     → Value (f x)
  | Unit        → Unit
```

```
type α list = Nil | Cons of (α × α list )

type relation α gen_list : α gen ⇒ α list with
  | Unit ⇒ Nil
  | Pair (Value x, t) ⇒ Cons (x, t) when t : α gen_list

let map_list = lifting map : (α → β) → α gen_list → β gen_list
```

(Simplified)

```
let rec map_list f z = match z with
  | Nil → Nil
  | Cons (u, v) → Cons (f u, map_list f v)
```

(Inlined)

```
type α gen =
  | Pair  of (α gen × α gen)
  | Value of α
  | Unit
```

```
let rec map f z = match z with
  | Pair (u, v) → Pair (map f u, map f v)
  | Value x     → Value (f x)
  | Unit        → Unit
```

```
type α tree = Leaf | Node of (α × (α tree × α tree))

type relation α gen_tree : α gen ⇒ α tree with
  | Unit ⇒ Leaf
  | Pair (Value x, Pair (t₁, t₂)) ⇒ Node (x, t₁, t₂) when t₁, t₂ : α gen_tree

let map_tree = lifting map : (α → β) → α gen_tree → β gen_tree
```

(Simplified)

```
let rec map_tree f z = match z with
  | Leaf → Leaf
  | Node (x, t₁, t₂) → Node (f x, map_tree f t₁, map_tree t₂)
```

(Inlined)

More examples

- Code specialization $\qquad$ *sets as unit maps*
- Code generalization $\qquad$ *from sets to maps*

More examples

- Code specialization *sets as unit maps*
- Code generalization *from sets to maps*

from nats to lists

(will be our running example)

A simpler example *nat & list*

(used as a running example to explain the details of lifting.)

Similar types

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

With similar values

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

Ornament relation

length

The ornament relation

```
type relation α natlist : nat ⇒ α list with
  | Z   ⇒ Nil
  | S m ⇒ Cons (_, m )  when α natlist : m ⇒ m
```

- _ stands for any value; may only appear on the right-hand side

add & append

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

Lifting

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

Lifting add into append

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

Lifting add into append

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

let append = lifting add : _ natlist → _ natlist → _ natlist
  patch #1 ← match m with Cons (x, _) → x
```

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

Lifting add into append

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

let append = lifting add : _ natlist → _ natlist → _ natlist
  patch #1 ← match m with Cons (x, _) → x
```

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

Lifting add into append

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

let append = lifting add : _ natlist → _ natlist → _ natlist
  patch .. | Cons (x, _) → Cons (#, _) ← x
```

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

Lifting add into append

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

let append = lifting add : _ natlist → _ natlist → _ natlist
  patch .. | Cons (x, _) → Cons (#, _) ← x
```

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

Lifting add into append

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

let append = lifting add : _ natlist → _ natlist → _ natlist
  patch .. | Cons (x, _) → Cons (#, _) ← x
```

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

How to proceed?

- in a principled manner—without arbitrary choices!
- so that the lifted program behaves similarly to the base one

Lifting

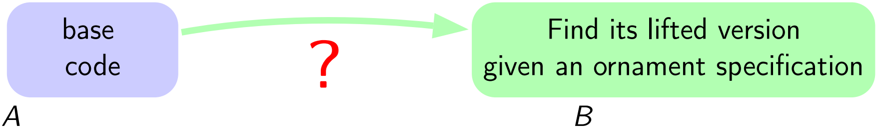
No reasonable place for abstraction a priori

base
code

A

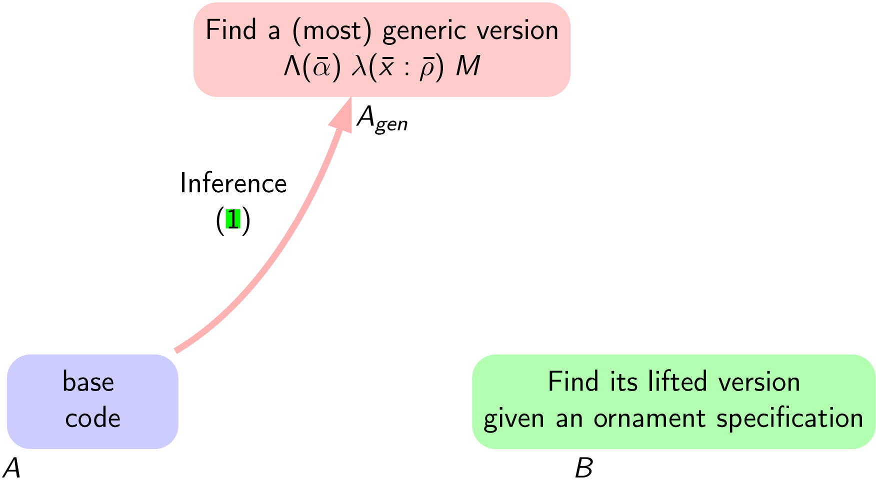
Lifting

Need to ornament some of the datatypes

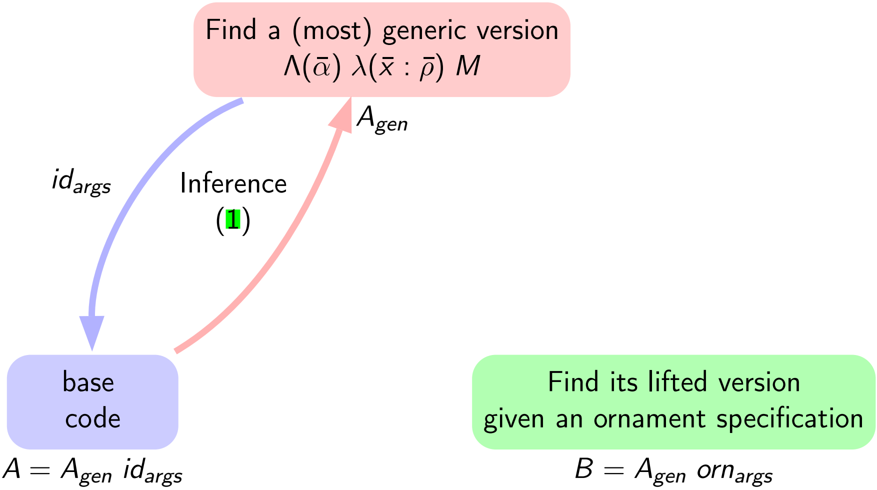


Lifting *by abstraction a posteriori*

(1) Abstract over (depends only on) what is ornamented.

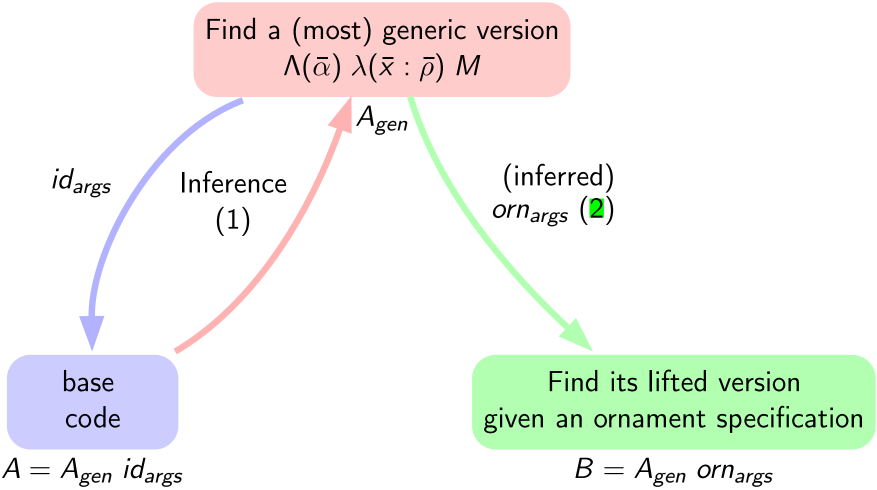


Lifting *by abstraction a posteriori*



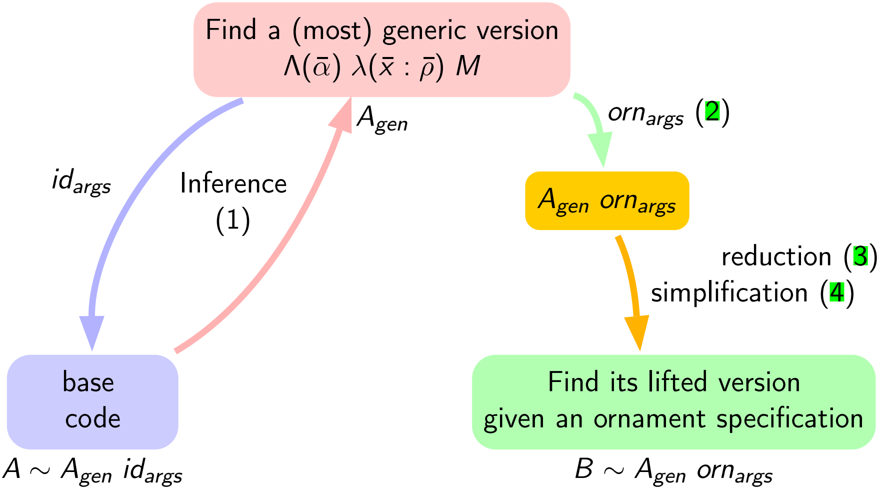
Lifting *by abstraction a posteriori*

(2) Specialize according to the liftting specification

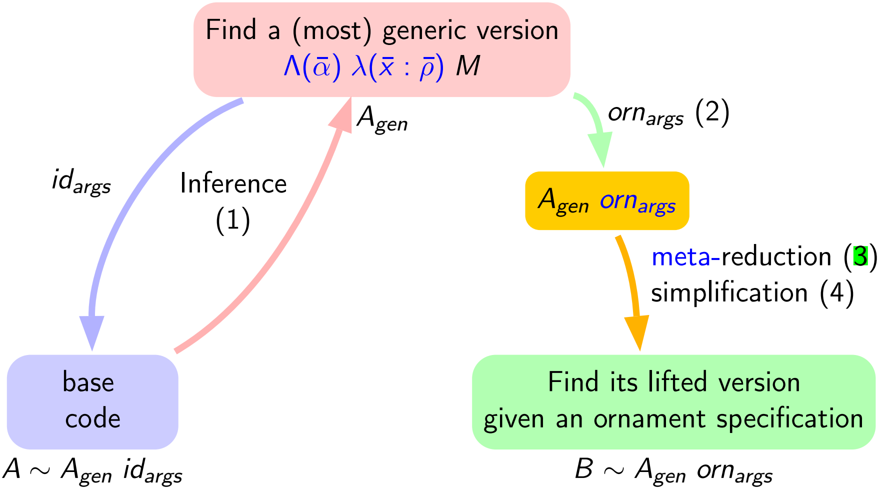


Lifting *by abstraction a posteriori*

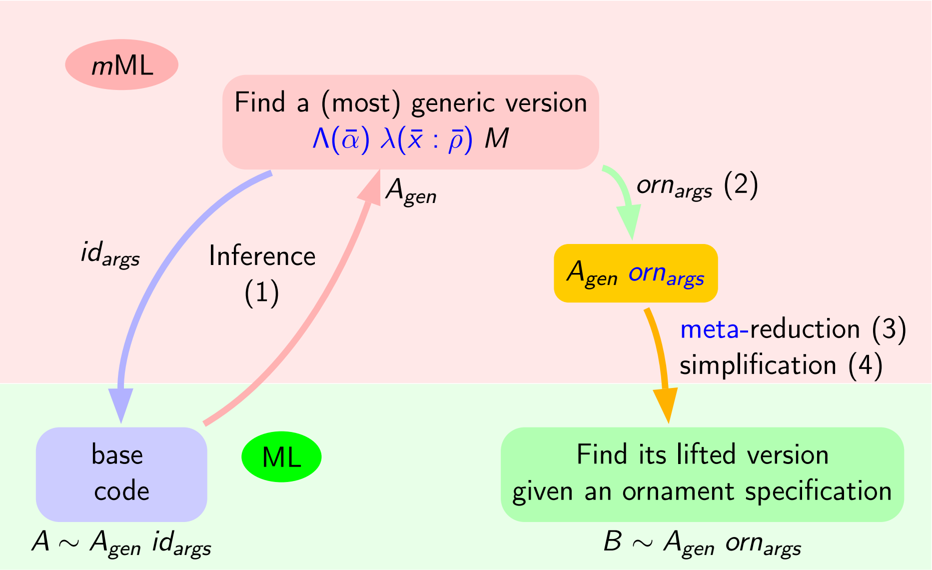
(3) Reduce and (4) Simplify



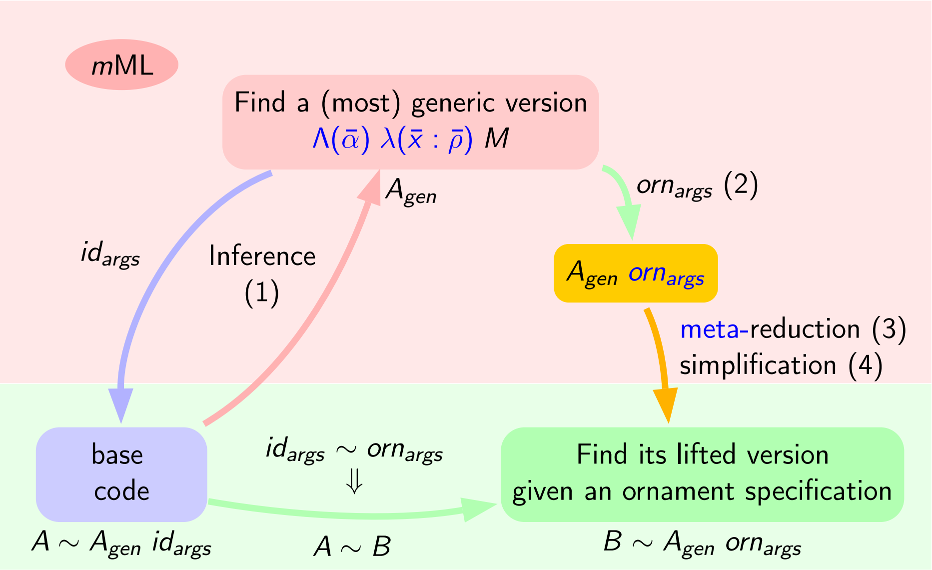
Lifting *by abstraction a posteriori*



Lifting *by abstraction a posteriori*



Lifting *by abstraction a posteriori*



Representing ornaments of nat

```
type α natS = Z' | S' of α
```

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

Representing ornaments of nat

```
type α natS = Z' | S' of α
```

- The ornamented datatype piggy bags on this skeleton:

$$\begin{tikzcd} \texttt{'a list} \arrow[r, bend left, "\texttt{list\_proj}"] & \texttt{('a list) natS} \arrow[l, bend left, "\texttt{list\_inj}"] \end{tikzcd}$$

```
let list_proj n =
  match n with
  | Nil          → Z'
  | Cons(_, t)   → S' t
```

```
let list_inj n x =
  match n with
  | Z'    → Nil
  | S' t  → Cons(x, t)
```

Representing ornaments of nat

```
type α natS = Z' | S' of α
```

- The ornamented datatype piggy bags on this skeleton:

$$\begin{tikzcd} \texttt{'a list} \arrow[r, bend left, "\texttt{list\_proj}"] & \texttt{('a list) natS} \arrow[l, bend left, "\texttt{list\_inj}"] \end{tikzcd}$$

- For convenience, we pack them in a datatype

```
type (α,β,γ) orn =
   { inj : α → β → γ;     proj : γ → α  }

let natlist =
  ({ inj = list_inj;        proj = list_proj }
   : ((α list) natS, α, α list) orn)
```

From add to append

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

From add to append

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

From add to append

```
let append =
  let rec add m n =
    match natlist.proj m with
      | Z' → n
      | S' m' →                    (S' (add m' n))
  in add
```

From add to append

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

From… add to a generic lifting…

```
let add_gen orn          patch =
  let rec add m n =
    match    orn.proj m with
      | Z' → n
      | S' m' →      orn.inj (S' (add m' n)) (patch m n)
  in add
```

From… a generic lifting back to append

```
let add_gen orn0 orn1 patch =
  let rec add m n =
    match orn0.proj m with
      | Z' → n
      | S' m' → orn1.inj (S' (add m' n)) (patch m n)
  in add
```

From… or back to add

```
let add_gen orn0 orn1 patch =
  let rec add m n =
    match orn0.proj m with
      | Z' → n
      | S' m' → orn1.inj (S' (add m' n)) (patch m n)
  in add
```

From add_gen back to append

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

From… or back to add

```
let add_gen orn0 orn1 patch =
  let rec add m n =
    match orn0.proj m with
      | Z' → n
      | S' m' → orn1.inj (S' (add m' n)) (patch m n)
  in add
```

From add_gen back to append

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

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

```
let add = add_gen natnat natnat (fun _ _ → ())

let natnat : (nat natSkel, α, nat) orn =
  { proj = (fun n → match n with Z → Z' | S m → S' m )
    inj = (fun n x → match n with Z' → Z | S' m → S m )}
```

Type Inference

```
let add_gen (orn0: (_,_,γ0) orn) (orn1: (_,β1,γ1) orn) p1 =

  let rec add m n =
    match orn0.proj m with
      | Z' → n
      | S' m' → orn1.inj (S' (add m' n)) (p1 m n : β1)

  in add
```

Coherence

- the same base type may be ornamented differently in different places
- except if their values (may) communicate

ML-style type inference

- For the ornament natlist

Type Inference

```
let add_gen (orn₀: (_,_,γ₀) orn) (orn₁: (_,β₁,γ₁) orn) p₁
                                 (orn₂: (_,β₂,γ₁) orn) p₂ =
  let rec add m n =
    match orn₀.proj m with
      | Z' → n
      | S₁' m' → orn₁.inj (S₁' (add m' n)) (p₁ m n : β₁)
      | S₂' m' → orn₂.inj (S₂' (add m' n)) (p₂ m n : β₂)
  in add
```

Coherence

- the same base type may be ornamented differently in different places
- except if their values (may) communicate

ML-style type inference

- If nat had 2 successor nodes, we would get . . .

Type Inference

```
let add_gen (orn₀: (_,_,γ₀) orn) (orn₁: (_,β₁,γ₁) orn) p₁
                                                        p₂ =
  let rec add m n =
    match orn₀.proj m with
      | Z' → n
      | S₁' m' → orn₁.inj (S₁' (add m' n)) (p₁ m n : β₁)
      | S₂' m' → orn₁.inj (S₂' (add m' n)) (p₂ m n : β₁)
  in add
```

Coherence

- the same base type may be ornamented differently in different places
- except if their values (may) communicate

ML-style type inference

- . . . and orn_1 and orn_2 should be identified

Type Inference

```
let add_gen (orn₀: (_,_,γ₀) orn) (orn₁: (_,β₁,γ₁) orn) p₁
                                                          p₂ =
  let rec add m n =
    match orn₀.proj m with
      | Z' → n
      | S₁' m' → orn₁.inj (S₁' (add m' n)) (p₁ m n : β₁)
      | S₂' m' → orn₁.inj (S₂' (add m' n)) (p₂ m n : β₂)
  in add
```

Coherence

- the same base type may be ornamented differently in different places
- except if their values (may) communicate

ML-style type inference

- Suffices here, but the injection need a dependent type in fine

Staging

```
let add_gen = fun orn₀  orn₁ patch →

  let rec add m n =
    match orn₀.proj    m with
      | Z' → n
      | S' m' → orn₁.inj    S' (add m' n)     (patch m n)
  in add

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

Meta-reduction

- We use meta abstractions and applications for the encoding
- To only reduce those redexes at compile time

Staging

```
let add_gen = fun orn₀  orn₁ patch ⇛

  let rec add m n =
    match orn₀.proj # m with
      | Z' → n
      | S' m' → orn₁.inj # S' (add m' n) # (patch m n)
  in add

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

Meta-reduction

- We use meta abstractions and applications for the encoding
- To only reduce those redexes at compile time

Staging

```
let add_gen = fun orn₀  orn₁ patch #⇒

  let rec add m n =
    match orn₀.proj # m with
      | Z' → n
      | S' m' → orn₁.inj # S' (add m' n) # (patch m n)
  in add

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

Meta-reduction

- We use meta abstractions and applications for the encoding
- To only reduce those redexes at compile time
- All #-abstractions and #-applications can actually be reduced.
- This is ensured just by a typing argument!

After meta-reduction

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

    | Z' → n
    | S' m' →
      (match S' (append m' n) with
        | Z' → Nil
        | S' t → Cons((match m with Cons(x,_) → x), t  ))
```

- There remains some redundant pattern matchings...
- Decoding `list` to `natS` and encoding `natS` to `list`.

Elimination of the encoding

```
let rec append m n =
  match (match m with
          | Nil → Z'
          | Cons(_, m') → S' m') with
    | Z' → n
    | S' m' →
      (match S' (append m' n) with
        | Z' → Nil
        | S' t → Cons((match m with Cons(x,_) → x), t ))
```

- 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(_, m') → S' m') with
    | Z' → n
    | S' m' →
      Cons((match m with Cons(x,_) → x), append m' n)
```

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

Elimination of the encoding

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

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

Elimination of the encoding

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

      (match Z' with
         | Z' → n
         | S' m' →
            Cons((match m with Cons(x,_) → x), append m' n))

   | Cons(_, m') →

      (match S' m' with
         | Z' → n
         | S' m' →
            Cons((match m with Cons(x,_) → x), append m' n))
```

- and reducing again

Elimination of the encoding

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

      (match Z' with
        | Z' → n
        | S' m' →
          Cons((match m with Cons(x,_) → x), append m' n))

  | Cons(_, m') →

      (match S' m' with
        | Z' → n
        | S' m' →
          Cons((match m with Cons(x,_) → x), append m' n))
```

- and reducing again

Elimination of the encoding

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

Elimination of the encoding *back to ML*

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

Elimination of the encoding *back to ML*

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

Elimination of the encoding *back to ML*

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

Elimination of the encoding *back to ML*

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

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

Some technical points

Stratification

$$\mathsf{ML} \quad \subseteq \quad e\mathsf{ML} \quad \subseteq \quad m\mathsf{ML}$$

Stratification

$$\mathsf{ML} \quad \subseteq \quad e\mathsf{ML} \quad \subseteq \quad m\mathsf{ML}$$

- The source language is (explicitly typed) ML

Stratification

$$\mathsf{ML} \quad \subseteq \quad e\mathsf{ML} \quad \subseteq \quad m\mathsf{ML}$$

- The source language is (explicitly typed) ML
- *e*ML adds dependent types over term equalities to ML

  Needed for typing the injection functions:

$$\begin{aligned} \mathit{list\_inj} \quad : \quad & \Lambda^\sharp\alpha.\, \Pi(m : \mathit{natS}(\mathsf{list}\ \alpha)). \\ & \Pi((x : \mathsf{match}\ m\ \mathsf{with}\ \mathsf{Z}' \to \mathsf{unit} \mid \mathsf{S}'\ \_ \to \alpha). \\ & \mathsf{list}\ \alpha \end{aligned}$$

Stratification

$$\mathrm{ML} \quad \subseteq \quad e\mathrm{ML} \quad \subseteq \quad m\mathrm{ML}$$

- The source language is (explicitly typed) ML
- *e*ML adds dependent types over term equalities to ML

  Needed for typing the injection functions:

$$\begin{aligned} \mathit{list\_inj} \quad : \quad & \Lambda^{\sharp}\alpha.\, \Pi(m : \mathit{natS}(\mathrm{list}\ \alpha)). \\ & \Pi((x : \mathrm{match}\ m\ \mathrm{with}\ \mathrm{Z}' \rightarrow \mathrm{unit} \mid \mathrm{S}'\ \_ \rightarrow \alpha)). \\ & \mathrm{list}\ \alpha \end{aligned}$$

Stratification

$$\mathrm{ML} \xrightarrow{\text{generic lifting}} \cdots \quad \mathrm{ML} \subseteq e\mathrm{ML} \subseteq m\mathrm{ML}$$

- The source language is (explicitly typed) ML
- *e*ML adds dependent types over term equalities to ML
- *m*ML adds (meta) abstractions and applications over all language constructs, including type equalities.

Stratification

$$\begin{tikzcd}[column sep=large] \mathsf{ML} \arrow[rrrr, bend left, "\text{generic lifting}"] & \subseteq & e\mathsf{ML} & \subseteq & m\mathsf{ML} \arrow[ll, bend left, "\substack{\text{instantiation}\\ \#\text{-reduction}}"] \end{tikzcd}$$

Stratification

$$\begin{tikzcd}
\mathsf{ML} \arrow[rrrr, bend left, "\text{generic lifting}"] & \subseteq & e\mathsf{ML} \arrow[ll, bend left, "\text{simplifications}"] & \subseteq & m\mathsf{ML} \arrow[ll, bend left, "\substack{\text{instantiation} \\ \#\text{-reduction}}"]
\end{tikzcd}$$

- By stratification, preservation of typing, and termination of meta-reduction

Stratification

$$\begin{tikzcd}[column sep=large]
\mathsf{ML} \arrow[rrrr, bend left, "\text{generic lifting}"] & \subseteq & e\mathsf{ML} \arrow[ll, bend left, "\text{simplifications}"] & \subseteq & m\mathsf{ML} \arrow[ll, bend left, "\substack{\text{instantiation} \\ \#\text{-reduction}}"]
\end{tikzcd}$$

- Type equivalences in derivations of ML judgments can always be eliminated

Type equivalence

Types depend on expressions & typing contexts contain term equalities

$$\text{match } \boxed{a} \text{ with } (P \to \tau \mid .. P \to \tau) \qquad\qquad \Gamma, \boxed{a =_\tau b}$$

Equations introduced on pattern matching are used in equalities, implicitly

$$\frac{\Gamma \vdash a : \zeta\,\overline{\tau} \qquad (d_i : \forall\,\overline{\alpha}\,.\,(\tau_{ij})^j \to \zeta\,\overline{\alpha}\,)^i \qquad (\Gamma, (x_{ij} : \tau_{ij}[\overline{\alpha} \leftarrow \overline{\tau}])^j, \boxed{a =_{\zeta\,\overline{\tau}} d_i\,\overline{\tau}\,(x_{ij})^j} \vdash b_i : \tau)^i}{\Gamma \vdash \text{match } a \text{ with } (d_i\,\overline{\tau}\,(x_{ij})^j \to b_i)^i : \tau}
\qquad
\frac{\Gamma \vdash a : \tau_1 \qquad \Gamma \vdash \tau_1 \simeq \tau_2}{\Gamma \vdash a : \tau_2}$$

Type equality, closed by equality on terms which includes

- term equalities assumptions, case splitting on pure terms
- reduction of type applications, pattern matchings, pure let-bindings
- closure by arbitrary context

Type equalities are necessary to check types defined by pattern matching and detect dead branches.

Logical relation

We first define a step-indexed logical relation $\mathcal{E}[\tau]_\gamma$ and $\mathcal{V}[\tau]_\gamma$ on *m*ML.
(A bit involved, because of dependent types, but standard.)

Ornament types ω

- same as types, but extended with datatype ornaments χ (*e.g.* natlist)
- can be projected to types ω^- and ω^+ *e.g.*

$$\begin{tikzcd}
& \alpha\ \mathsf{natlist} \to \alpha\ \mathsf{natlist} \to \alpha\ \mathsf{natlist} \arrow[dl, bend right, "-"'] \arrow[dr, bend left, "+"] & \\
\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} & & \alpha\ \mathsf{list} \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}
\end{tikzcd}$$

Logical relation naturally extends to ornament types:

- At ornament datatypes ω, we use the corresponding user defined ornament relation, *i.e.* pairs of values of types ω^- and ω^+ (much as the interpretation of abstract types)
- Use the standard definition elsewhere.

Ornament relations

An ornament definition

```
type relation α natlist : nat → α list with
  | Z   → Nil
  | S t → Cons (_, t)  when t : α natlist
```

defines a relation $\mathcal{V}[\text{natlist}\;\omega]_\gamma$ between values of type nat and list ω^+

$$(\mathsf{Z}, \mathsf{Nil}) \in \mathcal{V}[\text{natlist}\;\omega]_\gamma \qquad \frac{(u^-, u^+) \in \mathcal{V}[\text{natlist}\;\omega]_\gamma \qquad (v^-, v^+) \in \mathcal{V}[\omega]_\gamma}{(\mathsf{S}\;u^-, \mathsf{Cons}\;(v^+, u^+)) \in \mathcal{V}[\text{natlist}\;\tau]_\gamma}$$

Ornament relations

An ornament definition

```
type relation α natlist : nat → α list with
  | Z    → Nil
  | S t → Cons (_, t)  when t : α natlist
```

defines a relation $\mathcal{V}[\text{natlist}\,\omega]_\gamma$ between values of type nat and list ω^+

$$(\mathsf{Z}, \mathsf{Nil}) \in \mathcal{V}[\text{natlist}\,\omega]_\gamma \qquad \frac{(u^-, u^+) \in \mathcal{V}[\text{natlist}\,\omega]_\gamma \qquad (v^-, v^+) \in \mathcal{V}[\omega]_\gamma}{(\mathsf{S}\; u^-, \mathsf{Cons}\;(v^+, u^+)) \in \mathcal{V}[\text{natlist}\,\tau]_\gamma}$$

Here, this relation happens to be the inverse of the length function

$$(u^-, u^+) \in \mathcal{V}[\text{natlist}\,\omega]_\gamma \iff u^- = \mathit{length}\; u^+ \quad \wedge \begin{cases} u^- : \text{nat} \\ u^+ : \text{list}\,\omega^+ \end{cases}$$

Correctness of ornamentation

(Sketch)

- *add_gen* $\sim$ *add_gen* at a complicated ornament type
- *natnat* $\sim$ *natlist*
- $p_{nat} \sim p_{list}$ at ornament type natlist $\alpha \rightarrow$ natlist $\alpha \rightarrow \top$
  (we will never look into values returned by patches)
- *add_gen natnat natnat* p_{nat} $\sim$ *add_gen natlist natlist* p_{list}

Hence,

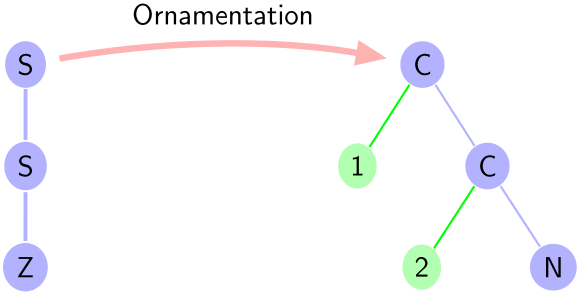
- add $\sim$ append at ornament type natlist $\alpha \rightarrow$ natlist $\alpha \rightarrow$ natlist α

Then:

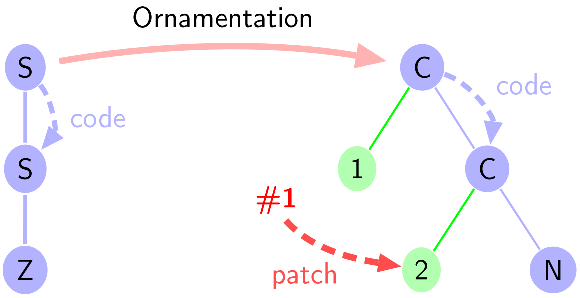
$$\begin{tikzcd}
\text{append} \arrow[d, bend right, "\text{length}"'] & n & m & : & \alpha\ \text{list} \\
\text{add} \arrow[u, dashed] & (\text{length } n) \arrow[u] & (\text{length } m) \arrow[u] & : & \text{nat} \arrow[u, "\alpha\ \text{natlist}"']
\end{tikzcd}$$

Disornamentation

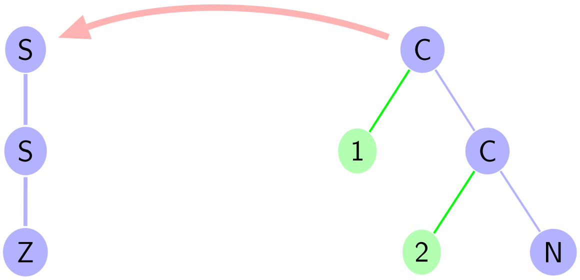
Disornamentation



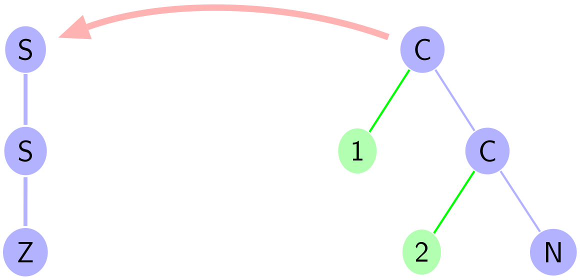
Disornamentation



Disornamentation



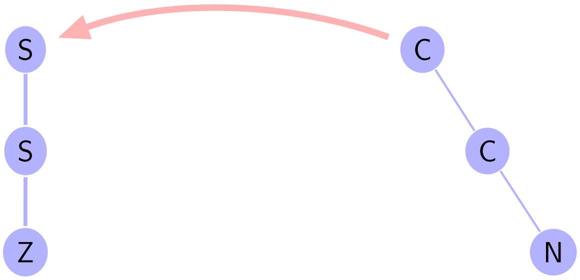
Disornamentation



Why useful?

- undo the ornamentation...
- offer a simplified view: locations, type annotations on ASTs, *etc.*
- ...

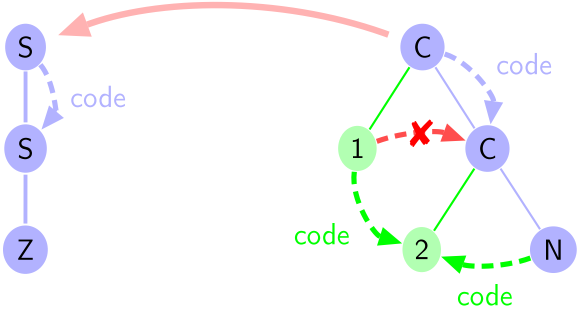
Disornamentation



Trivial case

- (binop example): ornamentation is bijective (no green) disornamentation is an ornamentation.

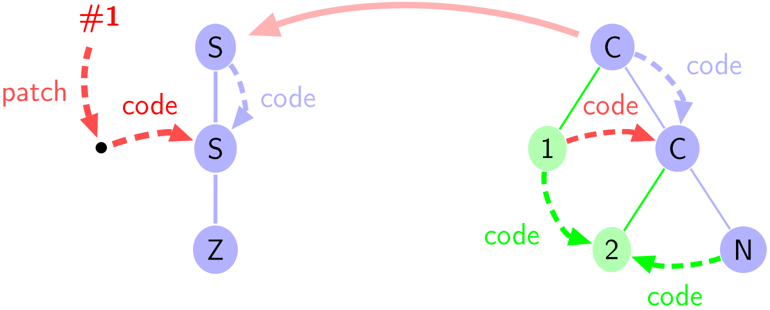
Disornamentation



Easy case

- The source is an ornamentation of the target
- Green nodes may depend on blue nodes but not conversely
- Hence, green code becomes useless code, and green nodes can be eliminated

Disornamentation



General case

- The blue code may be depend on green nodes.
- Then a patch is needed in the target
  to replace missed bindings in pattern matchings on green nodes.
- The green code is garbage collected.

Disornamentation — Ex. α list into nat

```
type (α, β) listS = Nil' | Cons' of α × β
```

Coercions

$$\begin{tikzcd} \texttt{nat} \arrow[r, bend left, "\texttt{nat\_proj}"] & (\alpha, \mathsf{nat})\ \mathsf{listS} \arrow[l, bend left, "\texttt{nat\_inj}"] \end{tikzcd}$$

```
let nat_proj n x = match n with
  | Z    → Nil'
  | S t → Cons' ( x , t)
```

```
let nat_inj n = match n with
  | Nil'            → Z
  | Cons'(_, t) → S t
```

```
type (α,β,γ) disorn = { inj : α → γ;      proj : γ → β → α }
let list_nat =          { inj = nat_inj;  proj = nat_proj }
  : ((α, nat) listS , α, nat) disorn)
```

Disornamentation — Ex. α list into nat

```
type (α, β) listS = Nil' | Cons' of α × β
```

Coercions

$$\begin{tikzcd} \texttt{nat} \arrow[r, bend left, "\texttt{nat\_proj}"] & (\alpha, \mathsf{nat})\ \mathsf{listS} \arrow[l, bend left, "\texttt{nat\_inj}"] \end{tikzcd}$$

```
let nat_proj n x = match n with
  | Z   → Nil'
  | S t → Cons' (x, t)
```

```
let nat_inj n = match n with
  | Nil'          → Z
  | Cons'(_, t)   → S t
```

Append generic version

```
let append_gen orn₀ orn₁ patch =
  let rec append m n =
    match orn₀.proj m (patch m n) with
      | Nil' → n
      | Cons'(x, m') → orn₁.inj (Cons' (x, append m' n))
  in append
```

```
type (α, β) listS = Nil' | Cons' of α × β
```

Coercions

$$\begin{tikzcd} \texttt{nat} \arrow[r, bend left, "\texttt{nat\_proj}"] & (\alpha, \mathsf{nat})\ \mathsf{listS} \arrow[l, bend left, "\texttt{nat\_inj}"] \end{tikzcd}$$

```
let nat_proj n x = match n with
  | Z   → Nil'
  | S t → Cons' (x, t)
```

```
let nat_inj n = match n with
  | Nil'          → Z
  | Cons'(_, t)   → S t
```

Append generic version, specialized to nats and simplified

```
let add patch =
  let rec append m n =
    let x = (patch m n) in                    — Useless binding
    match m with
      | Z → Z
      | S m' → S (append m' n)
  in append
```

Disornamentation — Ex. α list into nat

```
type (α, β) listS = Nil' | Cons' of α × β
```

Coercions

$$\begin{tikzcd} \texttt{nat} \arrow[r, bend left, "\texttt{nat\_proj}"] & (\alpha, \text{nat})\ \text{listS} \arrow[l, bend left, "\texttt{nat\_inj}"] \end{tikzcd}$$

```
let nat_proj n x = match n with
  | Z   → Nil'
  | S t → Cons' ( x , t)
```

```
let nat_inj n = match n with
  | Nil'          → Z
  | Cons'(_, t) → S t
```

Append generic version, specialized to nats and simplified

```
let add        =
  let rec append m n =

    match m with
      | Z → Z
      | S m' → S (append m' n)
  in append
```

Disornamentation *scenarios*

- Dropping balancing information from red-black trees
- Dropping location information from abstract syntax trees
- Better: maintaining two versions of the code in sync!

  $\Rightarrow$ Generate patches for reornamentation during disornamentation

Disornamentation *more*

Adding a new constructor to an existing data-type

- Use an empty type on the left-hand side

```
type relation oexp : exp ⇒ exp'
 ...
and ovalue : value ⇒ exp' with
 | Con i   ⇒  Con' i
 | Abs f   ⇒  Abs' f          when f : ovalue → oexp
 | ∼       ⇒  App' (u, v)
```

- Every pattern-matching on App' will require a patch on the corresponding branch.

Mixing ornamentation and disornamentation in the same transformation

Limitations and extensions

Beyond ML

GADTs ?

- Ornamentation in the presence of GADTs
- Ornamentation of ADTs into GADTs
- Conversely, disornamentation of GADTs into ADTs
  cf. Ghostbuster for Haskell [Trevor, McDonell, Zakian, Cimini, Newton]

Or more general dependent types?

Question

- Will the ornamented terms remain in the same source language ?

Side effects

- Ornamentation has been crafted to preserve the call-by-value evaluation order, so it should be unsurprising in practice.
- But no formalization.

Theoretical limits of (dis)ornamentation

Theorem

The lifted code behaves as the base code up to the relation between values of the base type and values of the lifted type.

Corollary

Ornaments cannot change the behavior of the base code.

- ✘ fix bugs
- ✘ turn an implementation of merge sort into quick sort

Based on datatype transformations

- ✘ modify the control, CPS transform, deforestation, *etc.*
- ✓ add or remove arguments to functions
  - viewing arguments of a given function as a specific tuple of arguments which can then be ornamented or disornamented

Practical limits of ornaments

Lifting is syntactic

- ✘ ornamentation points are derived from the syntax.
- ✘ η-expansion, if necessary, must be performed manually.
  - cannot derive a duplicating function from the identity function
- ✓ Still, unfolding of recursion is possible.

Beyond syntactic lifting

- Semantic preserving transformations may always be applied manually prior to ornamentation.
- Extend the notion of syntactic lifting? (maybe not necessary)

Combining transformations

$$\begin{tikzcd} P \arrow[r, bend right] & Q \end{tikzcd}$$

Combining transformations

$$P \longrightarrow P_1 \longrightarrow P_2 \cdots\cdots\cdots\cdots \dashrightarrow Q$$

Combining transformations

$$P \longrightarrow P_1 \longrightarrow P_2 \cdots\cdots \longrightarrow Q$$

General tooling already needed for pre/post processing

- Generate good names for new variables
- Pattern matching:
  - Transform deep pattern matching into narrow one beforehand
  - Inverse transformation that restores deep pattern matching afterwards
  - Factor identical branches
- Introduce and/or inline let bindings

Combining transformations

$$P \longrightarrow P_1 \longrightarrow P_2 \dashrightarrow Q$$

General tooling already needed for pre/post processing

Code inference

- Could autofill or propose some of the patches
- Inferring code from types, possibly with additional constraints
- Any other forms of code inference could be used.

Combining transformations

$$P \longrightarrow P_1 \longrightarrow P_2 \cdots\cdots\cdots\longrightarrow Q$$

General tooling already needed for pre/post processing

Code inference

Ornamentation-like transformations

- Extensible datatypes ?
  See *Trees that grows* by Shayan Najd & Simon Peyton Jones:
  - Ornamentation can already be used to add or remove constructors
  - They also factor the evolution of datatypes
  - Their solution is by abstraction a priori:
  - Is there an abstraction *a posteriori* alternative?

Combining transformations

$$P \longrightarrow P_1 \longrightarrow P_2 \cdots\cdots\longrightarrow Q$$

General tooling already needed for pre/post processing

Code inference

Ornamentation-like transformations

Other useful semantic preserving transformations?

- CPS transformation, Defunctionalization, Deforestation, *etc.*
- Many compiler optimizations could be made available to the user

Combining transformations

$$P \longrightarrow P_1 \longrightarrow P_2 \cdots\cdots\longrightarrow Q$$

General tooling already needed for pre/post processing

Code inference

Ornamentation-like transformations

Other useful semantic preserving transformations?

Non-semantic preserving transformations

- Necessary, for completeness, and to fix bugs!
- Hopefully, can be reduced to only a few, small transformations inserted between well-behaved ones.

Modes of interaction

- The most appealing usage is probably in an interactive mode, in some IDE with in place changes.
- But, we also need a batch mode
  - to separate the concerns, be independent of any IDE
  - we may wish to maintain two versions in sync (*e.g.* locations)
  - or maintain older versions for archival
- Raises new questions:
  - Design the right syntax for describing transformations
  - Robustness to source changes:
    - Up to which program transformations will a patch remain valid?
    - Can a patch from A to B be adapted when A changes?
  - Merging of two transformations done in parallel ...

Conclusion

We need a toolbox for safer, easier software evolution!

- With simple, composable, well-understood transformations
- Typed languages are a good setting:
  - Focus on type transformations, prior to code transformations.
  - Separate what can be automated, from what must be user provided
  - Abstraction *a posteriori* provides guidance and ensures a semantic preservation property
- Other applications of abstraction a posteriori? replace boiler plate code?

(Mixed) ornamentation is just one of the tools

- fits well within ML (see `http://gallium.inria.fr/~remy/ornaments/`)

Let's automate more parts of programming!

Outline