Ornamentation put into practice in ML

Didier Rémy

based on joined work with

Thomas Williams and Lucas Baudin

MFSP 2018
All have in common...

- Datatypes & Pattern-matching
- Polymorphism
- Type inference
- First-class functions
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...
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 \times exp
  | Mul of exp \times exp

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

type binop' = Add' | Mul'
type exp' =
  | Con' of int
  | Bin' of binop' \times exp' \times exp'
The poor man’s (good) tool

```ocaml
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)
```

```ocaml
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

```ocaml
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)
```

```ocaml
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

```ocaml
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)
```

```ocaml
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

```ocaml
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)
```

```ocaml
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

\[
\text{type } \text{exp} \ = \n
\begin{align*}
| \ & \text{Con} \ \text{of} \ \text{int} \\
| \ & \text{Add} \ \text{of} \ \exp \ \times \ \exp \\
| \ & \text{Mul} \ \text{of} \ \exp \ \times \ \exp
\end{align*}
\]

\[
\text{let } \text{parse } x = \text{Add} (x, \ \text{Con} \ 42)
\]

\[
\text{let rec } \text{eval } e = \text{match } e \ \text{with} \\
| \ & \text{Con} i \rightarrow i \\
| \ & \text{Add} (u, \ v) \rightarrow \text{add} (\text{eval} u) (\text{eval} v) \\
| \ & \text{Mul} (u, \ v) \rightarrow \text{mul} (\text{eval} u) (\text{eval} v)
\]

\[
\text{type } \text{binop} = \text{Add}’ | \ \text{Mul}’
\]

\[
\text{type } \text{exp’} = \\
| \ & \text{Con’} \ \text{of} \ \text{int} \\
| \ & \text{Bin’} \ \text{of} \ \text{binop} \ \times \ \exp’ \ \times \ \exp’
\]

\[
\text{let } \text{parse } x = \text{Bin’}(\text{Add’}, \ x, \ \text{Con’} \ 42)
\]

\[
\text{let rec } \text{eval } e = \text{match } e \ \text{with} \\
| \ & \text{Con’} i \rightarrow i \\
| \ & \text{Bin’}(\text{Add’}, \ u, \ v) \rightarrow \\
& \quad \text{add} (\text{eval} u) (\text{eval} v) \\
| \ & \text{Bin’}(\text{Mul’}, \ u, \ v) \rightarrow \\
& \quad \text{mul} (\text{eval} u) (\text{eval} v)
\]
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?

```ocaml
type exp =
| Con of int
| Add of exp \times exp
| Mul of exp \times exp

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

type binop’ = Add’ | Mul’
type exp’ =
| Con’ of int
| Bin’ of binop’ \times exp’ \times exp’

type relation oexp : exp \Rightarrow exp’ with
| Con i \Rightarrow Con’ i
| Add(u, v) \Rightarrow Bin’(Add’, u, v) \quad \text{when } u v : oexp
| Mul(u, v) \Rightarrow Bin’(Mul’, u, v) \quad \text{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**

\[
\text{type } \text{exp} = \\
| \text{Con } \text{of } \text{int} \\
| \text{Add } \text{of } \text{exp } \times \text{exp} \\
| \text{Mul } \text{of } \text{exp } \times \text{exp}
\]

**Let**

let parse \( x = \text{Add } (x, \text{Con } 42) \)

let rec eval \( e = \text{match } e \text{ with} \\
| \text{Con } i \rightarrow i \\
| \text{Add } (u, v) \rightarrow \text{add } (\text{eval } u) (\text{eval } v) \\
| \text{Mul } (u, v) \rightarrow \text{mul } (\text{eval } u) (\text{eval } v)
\]

**Type**

\[
\text{type } \text{binop}' = \text{Add}' | \text{Mul}'
\]

\[
\text{type } \text{exp}' = \\
| \text{Con}' \text{of } \text{int} \\
| \text{Bin}' \text{of } \text{binop}' \times \text{exp}' \times \text{exp}'
\]

**Type**

\[
\text{type } \text{relation } oexp : \text{exp} \Rightarrow \text{exp}' \text{ with} \\
| \text{Con } i \Rightarrow \text{Con}' i \\
| \text{Add}(u, v) \Rightarrow \text{Bin}'(\text{Add}', u, v) \quad \text{// when } oexp : u : \text{exp} \Rightarrow u : \text{exp}' \\
| \text{Mul}(u, v) \Rightarrow \text{Bin}'(\text{Mul}', u, v) \quad \text{// and } oexp : v : \text{exp} \Rightarrow v : \text{exp}'
\]
Can we do better?

```ocaml
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 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

type binop’ = Add’ | Mul’

type exp’ =
  | Con’ of int
  | Bin’ of binop’ × exp’ × exp’
```

lifting ∗ with oexp
Can we do better?

```ocaml
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)
```

```ocaml
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
⇒ green
```
Can we do better?

```ocaml
type exp =
  | Con of int
  | Add of exp \times exp
  | Mul of exp \times exp

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

type relation oexp : exp' \Rightarrow exp with
  | Con' i \Rightarrow Con i
  | Bin'(Add', u, v) \Rightarrow Add(u, v)
  | Bin'(Mul', u, v) \Rightarrow Mul(u, v)

lifting \ast with oexp
```

```ocaml
type binop' = Add' | Mul'
type exp' =
  | Con' of int
  | Bin' of binop' \times exp' \times exp'

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

lifting \ast with oexp

```text
blue + red \implies green
```
Enforcing more invariants

```ocaml
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

```plaintext
type exp =
  | Con of int
  | Abs of (exp \rightarrow exp)
  | App of exp \times exp

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

```plaintext
type exp' =
  | Val of value'
  | App' of exp' \times exp'

and value' =
  | Con' of int
  | Abs' of (value' \rightarrow 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

```plaintext
type exp =
| Con of int
| Abs of (exp \rightarrow exp)
| App of exp \times exp

type exp' =
| Val of value'
| App' of exp' \times exp'
and value' =
| Con' of int
| Abs' of (value' \rightarrow exp')

type relation oexp : exp \Rightarrow exp' with
| Con i \Rightarrow Val (Con' i)
| Abs f \Rightarrow Val (Abs' f) \text{ when } f : ovalue \rightarrow oexp
| App (u,v) \Rightarrow App' (u,v) \text{ when } u,v : oexp
and ovalue : exp \Rightarrow value' with
| Con i \Rightarrow Con' i
| Abs f \Rightarrow Abs' f \text{ when } f : ovalue \rightarrow oexp
| App (u,v) \Rightarrow \sim
```

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

```
let rec map f z = match z with
  | Nil → Nil
  | Cons(x, t) → Cons(f x, map f t)
```

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

type α triple = T of α × (α × (α × unit))
```
Porting operations on lists to tuples

type $\alpha$ list = Nil | Cons of ($\alpha \times \alpha$ list)

let rec map f z = match z with
| Nil $\rightarrow$ Nil
| Cons($x$, $t$) $\rightarrow$ Cons($f$ $x$, map $f$ $t$)

type unit = U

type $\alpha$ triple = T of $\alpha \times (\alpha \times (\alpha \times$ unit))

type relation $\alpha$ list_triple : $\alpha$ list $\Rightarrow$ $\alpha$ triple with
| Cons ($x_1$, Cons ($x_2$, Cons ($x_3$, Nil)))) $\Rightarrow$ T ($x_1$, ($x_2$, ($x_3$, U))))

Simplified
Porting operations on lists to tuples

\[
\text{type } \alpha\ \text{list } = \text{Nil} | \ \text{Cons of } (\alpha \times \alpha\ \text{list})
\]

let rec map f z = match z with
  | Nil → Nil
  | Cons(x, t) → Cons(f x, map f t)

\[
\text{type } \text{unit } = U
\]

\[
\text{type } \alpha\ \text{triple } = \\text{T of } \alpha \times (\alpha \times (\alpha \times \text{unit}))
\]

\[
\text{Simplified type relation } \alpha\ \text{list}_\text{triple} : \alpha\ \text{list} \Rightarrow \alpha\ \text{triple with}
\]

| Cons (x₁, Cons (x₂, Cons (x₃, Nil))) \(\Rightarrow\) T (x₁, (x₂, (x₃, U)))

let map_tuple = lifting map : (\alpha \rightarrow \beta) \rightarrow \alpha\ \text{list}_\text{triple} \rightarrow \beta\ \text{list}_\text{triple}
Porting operations on lists to tuples

```ocaml
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

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

```
let lift = lifting map : (α → β) → α list_triple → β list_triple

let let rec map_tuple f z = match z with
  | Nil → Nil
  | Cons(x, t) → Cons(f x, map f t)
```

Simplified type relation

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

With manual inlining

```ocaml
let rec map_tuple f z = match z with
  | Nil → Nil
  | Cons(x, t) → Cons(f x, map f t)
```
**Generic programming**

\[
\text{type } \alpha \text{ gen } = \\
| \text{Pair of } (\alpha \text{ gen } \times \alpha \text{ gen}) \\
| \text{Value of } \alpha \\
| \text{Unit}
\]

**Generic code**

\[
\text{let rec map } f \ z = \text{match } z \text{ with} \\
| \text{Pair } (u, v) \rightarrow \text{Pair } (\text{map } f \ u, \text{map } f \ v) \\
| \text{Value } x \rightarrow \text{Value } (f \ x) \\
| \text{Unit} \rightarrow \text{Unit}
\]
Generic programming

Lists

type \( \alpha \) gen =
  | Pair of \((\alpha \) gen \times \(\alpha \) gen)
  | Value of \(\alpha\)
  | Unit

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


type \(\alpha\) list = Nil | Cons of \((\alpha \times \alpha\) list\)

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

Simplified

type relation \(\alpha\) gen_list : \(\alpha\) gen ⇒ \(\alpha\) list with
  | Unit ⇒ Nil
  | Pair \((\text{Value} \(x\), t)\) ⇒ Cons \((x, t)\) when \(t : \alpha\) gen_list

let map_list = lifting \(map : (\alpha \rightarrow \beta) \rightarrow \alpha\) gen_list \(\rightarrow\) \(\beta\) gen_list

(Inlined)

let rec map_list \(f\) \(z\) = match \(z\) with
  | Nil \(\rightarrow\) Nil
  | Cons \((u, v)\) \(\rightarrow\) Cons \((f \(u\), \text{map_list} \(f\) \(v\))\)
Generic programming

**Trees**

```plaintext
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

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)
```

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

(Simplified)

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)
```

```plaintext
(3)9 / 41
```
More examples

- Code specialization  
  sets as unit maps
- Code generalization  
  from sets to maps
More examples

- Code specialization
  sets as unit maps
  from sets to maps
  from nats to lists
(will be our running example)

- Code generalization
A simpler example \textit{nat & list}

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

Similar types

\texttt{type nat = Z | S of nat}

\texttt{type \alpha list = Nil | Cons of \alpha \times \alpha list}

With similar values

\texttt{S (S (S (Z )))}

\texttt{Cons (1, Cons (2, Cons (3, Nil )))}

The ornament relation

\texttt{type relation \alpha natlist : nat \Rightarrow \alpha list with}

\begin{align*}
| \texttt{Z} & \Rightarrow \texttt{Nil} \\
| \texttt{S \: m} & \Rightarrow \texttt{Cons (\_, \: m)} \quad \text{when} \quad \texttt{\alpha natlist : \: m \Rightarrow \: m}
\end{align*}

\(\_\) 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)

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

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

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

```ocaml
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

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

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

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

```reason
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 (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

Find its lifted version given an ornament specification

A

base code

? 

B

Find its lifted version given an ornament specification
Lifting by abstraction a posteriori

Find a (most) generic version
\[ \Lambda(\bar{\alpha}) \lambda(\bar{x} : \bar{\rho}) M \]

Abstract over (depends only on) what is ornamented.

Inference

(1) Find its lifted version given an ornament specification

A

base code

A_{\text{gen}}

B

Find its lifted version given an ornament specification

(1)
Lifting 

by abstraction a posteriori

Find a (most) generic version
\[ \Lambda(\bar{\alpha}) \lambda(\bar{x} : \bar{\rho}) M \]

Inference

Find its lifted version given an ornament specification
\[ B = A_{gen} \text{orn}_{args} \]
Lifting by abstraction a posteriori

(2) Specialize according to the lifting specification

Find a (most) generic version
\[ \Lambda(\vec{\alpha}) \lambda(\vec{x} : \vec{\rho}) M \]

\( A_{\text{gen}} \)

Find its lifted version given an ornament specification
\( B = A_{\text{gen}} \ orn_{\text{args}} \)

\( A = A_{\text{gen}} \ id_{\text{args}} \)

Inference (1)

Inference (2)
Lifting \textit{by abstraction a posteriori}

(3) Reduce and (4) Simplify

Find a (most) generic version
\[ \Lambda(\bar{\alpha}) \lambda(\bar{x} : \bar{\rho}) M \]

\( A \sim A_{\text{gen}} \ id_{\text{args}} \)

Inference (1)

\( A_{\text{gen}} \)

\( \text{orn}_{\text{args}} \) (2)

\( A_{\text{gen}} \ orn_{\text{args}} \)

Find its lifted version given an ornament specification

\( B \sim A_{\text{gen}} \ orn_{\text{args}} \)
Lifting by abstraction a posteriori

Find a (most) generic version
\[ \Lambda(\bar{\alpha}) \lambda(\bar{x} : \bar{\rho}) \ M \]

Find its lifted version given an ornament specification
\[ B \sim A_{gen} \text{orn}_{args} \]

Inference
\[ A \sim A_{gen} \text{id}_{args} \]

\[ \text{Inference (1)} \]

\[ \text{orn}_{args} \text{(2)} \]

\[ \text{meta-reduction (3)} \]

\[ \text{simplification (4)} \]
Lifting by abstraction a posteriori

Find a (most) generic version
$$\Lambda(\bar{\alpha}) \lambda(\bar{x} : \bar{\rho}) M$$

Inference (1)

Find its lifted version given an ornament specification
$$B \sim A_{gen} orn_{args}$$
Lifting by abstraction a posteriori

Find a (most) generic version
\( \Lambda(\bar{\alpha}) \lambda(\bar{x} : \bar{\rho}) M \)

\[ A \sim A_{\text{gen}} id_{\text{args}} \]

\[ id_{\text{args}} \sim orn_{\text{args}} \]

Find its lifted version given an ornament specification
\[ B \sim A_{\text{gen}} orn_{\text{args}} \]

Inference

meta-reduction

simplification

mML
Representing ornaments of nat

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

- 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** \( \alpha \) natS = \( Z' \mid S' \) of \( \alpha \)

The ornamented datatype piggy bags on this skeleton:

\[ \text{list}_\text{proj} \]

\[ 'a \text{ list} \]

\[ \text{list}_\text{inj} \]

\[ (\ 'a \text{ list} ) \text{ natS} \]

\[ \text{let} \text{ list}_\text{proj} \text{ } n = \begin{align*} &\text{match } n \text{ with} \nonumber \ \text{match } n \text{ with} \nonumber \ \text{match } n \text{ with} \nonumber \ \text{match } n \text{ with} \nonumber \ | \ N\text{i}l & \rightarrow Z' \nonumber \ | \ Z' & \rightarrow \text{Nil} \nonumber \ | \ \text{C}o\text{ns}(\_, t) & \rightarrow S' \ t 
\end{align*} \]

\[ \text{let} \text{ list}_\text{inj} \text{ } n \ x = \begin{align*} &\text{match } n \text{ with} \nonumber \ | \ Z' & \rightarrow \text{Nil} \nonumber \ | \ S' \ t & \rightarrow \text{Cons}(x, \ t) 
\end{align*} \]
Representing ornaments of nat

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

- The ornamented datatype piggy bags on this skeleton:
  \[
  \begin{array}{c}
  \text{list\_proj} \\
  \text{'a list} \leftrightarrow \text{('a list) natS} \\
  \text{list\_inj}
  \end{array}
  \]

- For convenience, we pack them in a datatype

\[
\text{type } (\alpha, \beta, \gamma) \text{ orn } =
\begin{cases}
\text{inj : } \alpha \to \beta \to \gamma; & \text{proj : } \gamma \to \alpha
\end{cases}
\]

\[
\text{let } \text{nat\_list } =
\begin{cases}
\text{inj } = \text{list\_inj}; & \text{proj } = \text{list\_proj }
\end{cases}
\begin{cases}
: ((\alpha \text{ list) natS}, \alpha, \alpha \text{ list) orn})
\end{cases}
\]
From add to append

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

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
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
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)
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

```ml
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

```ml
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 orn₁ and orn₂ should be identified
**Type Inference**

```ml
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
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
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
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

```plaintext
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

\[
\text{let rec append } m n = \\
\quad \text{match } (\text{match } m \text{ with } \\
\quad \quad | \text{Nil } \rightarrow Z' \\
\quad \quad | \text{Cons}(\_ , m') \rightarrow S' m') \text{ with } \\
\quad \quad \quad | Z' \rightarrow n \\
\quad \quad \quad | S' m' \rightarrow \\
\quad \quad \quad \quad (\text{match } S' (\text{append } m' n) \text{ with } \\
\quad \quad \quad \quad \quad | Z' \rightarrow \text{Nil} \\
\quad \quad \quad \quad \quad | S' t \rightarrow \text{Cons}((\text{match } m \text{ with } \text{Cons}(x,\_ ) \rightarrow x), t))
\]

- 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)
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

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

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

```plaintext
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)
let rec append \( m \ n = \)

\[
\begin{align*}
\text{match } m \text{ with } \\
| \text{Nil} & \rightarrow n \\
| \text{Cons } (x, m') & \rightarrow \text{Cons ((match } m \text{ with Cons } x \rightarrow x), \text{ append } m' \ n)} \\
\end{align*}
\]

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)
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

\[ \text{ML} \subseteq \text{eML} \subseteq \text{mML} \]
The source language is (explicitly typed) ML
Stratification

ML ⊆ eML ⊆ mML

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

Needed for typing the injection functions:

\[
\text{list}_\text{inj} : \Lambda^\# \alpha. \Pi(m : \text{natS(list } \alpha)) . \Pi((x : \text{match } m \text{ with } Z' \rightarrow \text{unit} | S' _ \_ \rightarrow \alpha).)
\]

\[
\text{list } \alpha
\]
Stratification

\[ \text{ML} \subseteq \text{eML} \subseteq \text{mML} \]

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

Needed for typing the injection functions:

\[
\text{list\_inj} : \Lambda^\#_{\alpha}. \ \Pi(m : \text{natS}(\text{list}\ \alpha)).
\Pi((x : \text{match } m \text{ with } Z' \rightarrow \text{unit} | S' \_ \rightarrow \alpha).
\text{list}\ \alpha
\]
The source language is (explicitly typed) ML

eML adds dependent types over term equalities to ML

mML adds (meta) abstractions and applications over all language constructs, including type equalities.
Stratification

\[ \text{ML} \subseteq \text{eML} \subseteq \text{mML} \]

generic lifting

instantiation

#-reduction
Stratification

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

Type equivalences in derivations of ML judgments can always be eliminated.
eML

Type equivalence

Types depend on expressions & typing contexts contain term equalities

\[ \text{match } a \text{ with } (P \rightarrow \tau \mid .. P \rightarrow \tau) \]

\[ \Gamma, a =_{\tau} b \]

Equations introduced on pattern matching are used in equalities, implicitly

\[ \Gamma \vdash a : \zeta \bar{\tau} \quad (d_i : \forall \alpha . (\tau_{ij})^j \rightarrow \zeta \bar{\alpha})^i \]

\[ (\Gamma, (x_{ij} : \tau_{ij}[\bar{\alpha} \leftarrow \bar{\tau}])^j, a =_{\zeta \bar{\tau}} d_i \bar{\tau}(x_{ij})^j \vdash b_i : \tau)^i \]

\[ \Gamma \vdash \text{match } a \text{ with } (d_i \bar{\tau}(x_{ij})^j \rightarrow b_i)^i : \tau \]

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 $E[\tau]_\gamma$ and $V[\tau]_\gamma$ on $\text{mML}$. (A bit involved, because of dependent types, but standard.)

Ornament types $\omega$

- same as types, but extended with datatype ornaments $\chi$ (e.g. natlist)
- can be projected to types $\omega^-$ and $\omega^+$ e.g.

\[
\begin{align*}
\alpha \text{natlist} & \to \alpha \text{natlist} \\
\alpha \text{natlist} & \to \alpha \text{natlist} \\
\alpha \text{list} & \to \alpha \text{list} \\
\alpha \text{list} & \to \alpha \text{list}
\end{align*}
\]

Logical relation naturally extends to ornament types:

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

\[
\text{type relation } \alpha \text{ natlist : nat } \rightarrow \alpha \text{ list with } \\
\quad | \ Z \rightarrow \text{Nil} \\
\quad | \ S \ t \rightarrow \text{Cons} \ (\_ \ , \ t) \ \text{ when } t : \alpha \text{ natlist}
\]

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

\[
(Z, \text{Nil}) \in \mathcal{V}[\text{natlist } \omega]_\gamma \quad (u^-, u^+) \in \mathcal{V}[\text{natlist } \omega]_\gamma \quad (v^-, v^+) \in \mathcal{V}[\omega]_\gamma
\]

\[
(S \ u^-, \text{Cons} \ (v^+, u^+)) \in \mathcal{V}[\text{natlist } \tau]_\gamma
\]
Ornament relations

An ornament definition

\[
\text{type relation } \alpha \text{ natlist : nat } \rightarrow \alpha \text{ list with } \\
| Z \rightarrow \text{Nil} \\
| S \ t \rightarrow \text{Cons (_, t)} \quad \text{when } t : \alpha \text{ natlist}
\]
defines a relation \( \mathcal{V}[\text{natlist } \omega]_\gamma \) between values of type nat and list \( \omega^+ \)

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

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

\[
(u^-, u^+) \in \mathcal{V}[\text{natlist } \omega]_\gamma \iff u^- = \text{length } u^+ \\
\quad \land \quad \begin{cases} 
  u^- : \text{nat} \\
  u^+ : \text{list } \omega^+
\end{cases}
\]
Correctness of ornamentation

(Sketch)

- \( \text{add\_gen} \sim \text{add\_gen} \)
  at a complicated ornament type
- \( \text{natnat} \sim \text{natlist} \)
- \( \text{p\_nat} \sim \text{p\_list} \)
  at ornament type natlist \( \alpha \rightarrow \text{natlist} \alpha \rightarrow \top \)
  (we will never look into values returned by patches)
- \( \text{add\_gen} \, \text{natnat} \, \text{natnat} \, \text{p\_nat} \sim \text{add\_gen} \, \text{natlist} \, \text{natlist} \, \text{p\_list} \)

Hence,

- \( \text{add} \sim \text{append} \)
  at ornament type natlist \( \alpha \rightarrow \text{natlist} \alpha \rightarrow \text{natlist} \alpha \)

Then:

\[
\begin{align*}
\text{length} & \quad \text{add} \quad (\text{length } n) \quad (\text{length } m) \\
\text{append} & \quad n \quad m \\
\end{align*}
\]

: \( \alpha \) list

\( \alpha \) natlist

\( \alpha \) nat

\( \alpha \) list

\( \alpha \) natlist
Disornamentation
Disornamentation

Ornamentation

S

S

Z

1

C

2

N
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
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. $\alpha$ list into nat**

\[
\text{type (}\alpha, \beta\text{) list}S = \text{Nil'} | \text{Cons'} \text{ of } \alpha \times \beta
\]

Coercions

\[
\begin{array}{c}
\text{nat proj} \\
\text{nat} \\
\text{nat inj}
\end{array}
\]

\[
\begin{array}{c}
\text{nat proj } n \ x = \text{match } n \text{ with} \\
| \text{Z } \rightarrow \text{Nil'} \\
| \text{S } t \rightarrow \text{Cons'} (\ x, \ t)
\end{array}
\]

\[
\begin{array}{c}
\text{let } \text{nat proj } n \ x = \text{match } n \text{ with} \\
| \text{Z } \rightarrow \text{Nil'} \\
| \text{S } t \rightarrow \text{Cons'} (\ x, \ t)
\end{array}
\]

\[
\begin{array}{c}
\text{let } \text{nat inj } n = \text{match } n \text{ with} \\
| \text{Nil'} \rightarrow \text{Z} \\
| \text{Cons'}(\ _, \ t) \rightarrow \text{S } t
\end{array}
\]

\[
\text{type (}\alpha, \beta, \gamma\text{) disorn} = \{ \text{ inj : } \alpha \rightarrow \gamma; \text{ proj : } \gamma \rightarrow \beta \rightarrow \alpha \}
\]

\[
\text{let list nat} = \{ \text{ inj = nat inj; proj = nat proj } \}
\]

\[
: ((\alpha, \text{nat) list}S, \alpha, \text{nat) disorn})
\]
Disornamentation

Ex. \(\alpha\) list into \(\text{nat}\)

\[
\text{type } (\alpha, \beta) \text{ list}_S = \text{Nil'} | \text{Cons'} \text{ of } \alpha \times \beta
\]

Coercions

\[
\begin{align*}
\text{n} \quad & \quad \text{n} \quad \text{n} \text{nat}\_\text{proj} \quad \text{n} \text{nat}\_\text{inj}
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{n} \text{at}\_\text{proj } n \ x &= \text{match } n \text{ with } \\
& | \text{Z} & \rightarrow \text{Nil'} \\
& | \text{S } t & \rightarrow \text{Cons'} (x, t)
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{n} \text{at}\_\text{inj } n &= \text{match } n \text{ with } \\
& | \text{Nil'} & \rightarrow \text{Z} \\
& | \text{Cons'}(_, t) & \rightarrow \text{S } t
\end{align*}
\]

Append generic version

\[
\begin{align*}
\text{let } \text{append\_gen } \text{orn}_0 \ \text{orn}_1 \ \text{patch} &= \\
\quad \text{let rec } \text{append } m \ n = \\
\quad \quad \text{match } \text{orn}_0.\text{proj } m \ (\text{patch } m \ n) \text{ with } \\
\quad \quad \quad | \text{Nil'} & \rightarrow n \\
\quad \quad \quad | \text{Cons'}(x, m') & \rightarrow \text{orn}_1.\text{inj} (\text{Cons'} (x, \text{append } m' \ n))
\end{align*}
\]
Disornamentation

Ex. \( \alpha \) list into \( \text{nat} \)

\[
\begin{align*}
\text{type } (\alpha, \beta) \text{ list } S &= \text{Nil}' \mid \text{Cons}' \text{ of } \alpha \times \beta \\
\text{Coercions}
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{nat}_\text{proj } n & = \text{match } n \text{ with } \\
| \text{Z} & \rightarrow \text{Nil}' \\
| \text{S } t & \rightarrow \text{Cons}' (x, t) \\
\text{let } \text{nat}_\text{inj } n & = \text{match } n \text{ with } \\
| \text{Nil}' & \rightarrow \text{Z} \\
| \text{Cons}'(\_, t) & \rightarrow \text{S } t \\
\end{align*}
\]

Append generic version, specialized to nats and simplified

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

Ex. \( \alpha \) list into \( \text{nat} \)

type \((\alpha, \beta)\) list\(S = \text{Nil}' \mid \text{Cons'} \) of \(\alpha \times \beta\)

Coercions

\[
\begin{array}{ccc}
\text{nat} & \xrightarrow{\text{nat}_\text{proj}} & (\alpha, \text{nat}) \ \text{list}\(S \\
& \xleftarrow{\text{nat}_\text{inj}} & \text{nat}
\end{array}
\]

let \(\text{nat}_\text{proj} \) \(n \ x = \text{match} \ n \ \text{with}
| \ Z \rightarrow \text{Nil}'
| \ S \ t \rightarrow \text{Cons'} (x, t)

let \(\text{nat}_\text{inj} \) \(n = \text{match} \ n \ \text{with}
| \ \text{Nil}' \rightarrow Z
| \ \text{Cons'}(\_, t) \rightarrow S \ t

Append generic version, specialized to nats and simplified

let add = let rec append \(m \ n =

\[
\begin{align*}
& \text{match} \ m \ \text{with} \\
& | \ Z \rightarrow Z \\
& | \ S \ m' \rightarrow S \ (\text{append} \ m' \ n)
\end{align*}
\]
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!
  ⇒ Generate patches for reornamentation during disornamentation
Adding a new constructor to an existing data-type

- Use an empty type on the left-hand side

```plaintext
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.
✗ $\eta$-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
Combining transformations

\[ P \rightarrow P_1 \rightarrow P_2 \rightarrow Q \]
Combining transformations

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

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

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

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

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/](http://gallium.inria.fr/~remy/ornaments/))

Let’s automate more parts of programming!
1 Examples
2 Nats and lists
3 Abstraction a posteriori
4 Encoding and simplifications
5 Technical points
6 Disornamentation
7 Discussion
8 Conclusion