Ornaments
Exploiting Parametricity for Safer, More Automated Code Transformations

Didier Rémy

based on joined work with
Thomas Williams

informatics mathematics
Inria

Haskell Symposium 2017

Haskell $\triangleleft\cdots\cdots$ ML $\cdots\cdots\triangleright$ OCaml

Hope, Miranda

Haskell $\triangleright$........ ML .... $\triangleleft$ OCaml

Hope, Miranda

Haskell ▷........ ML .... ◁ OCaml

Hope, Miranda

In common, since the origin. . .

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

Haskell ▷........ ML ....◁ OCaml

Hope, Miranda

In common, since the origin...

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

Therefore,

- Programs are safer by construction

  (and Haskell ones perhaps even more...)
- Still, they sometimes need to be modified...

Haskell ▷······· ML ·······◁ OCaml

Hope, Miranda

In common, since the origin...

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

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, because:
  - programs being structured around datatypes
  - polymorphism and type inference.

Plan

In this talk,

- I will show how a small subcase of code refactoring and code refinement based on ornements can be put into practice in languages such as OCaml or (core) Haskell.
  - Examples
  - Look under the hood
- I will also draw conclusions from this experience, and discuss code evolution in more general terms.

This is largely based on joined work with Thomas Williams.

Ornaments have been introduced by Conor McBride and explored with Pierre-Évariste Dagan in the context of Adga.

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
  | 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 (bad) 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 exp 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)
```

Can we do better?

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

let exp 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 ornament 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 exp 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 ornament 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 exp 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 ornament 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 exp 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 ornament 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
```

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

Can we do better?

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

let exp 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 ornament 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 exp 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 ornament 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* $\Longrightarrow$ *green*

Permuting values of a datatype

Input program

```
let process config x = match config with
  gt x (match config with True → 0 | False → 1)
let myconfig = True
let main = process myconfig 42
```

Permuting values of a datatype

Input program

```
let process config x = match config with
  gt x (match config with True → 0 | False → 1)
let myconfig = True
let main = process myconfig 42
```

Safely exchanging the values of boolean, selectively

```
type ornament inverse : bool ⇒ bool with
  | True ⇒ False
  | False ⇒ True
let process = lifting process : inverse → _ → _
let myconfig = lifting myconfig : inverse
let main = lifting main : bool
```

Permuting values of a datatype

Output program

```
let process config x = match config with
  gt x (match config with True → 1 | False → 0)
let myconfig = False
let main = process myconfig 42
```

Safely exchanging the values of boolean, selectively

```
type ornament inverse : bool ⇒ bool with
  | True ⇒ False
  | False ⇒ True
let process = lifting process : inverse → _ → _
let myconfig = lifting myconfig : inverse
let main = lifting main : bool
```

- The inverse transformation is used selectively.
- The ornamentation typechecking / inference tracks the relations between the old and new versions of bool and ensures consistency.

Permuting values of a datatype ◁

Output program incomplete!

```
let process config x = match config with
  gt x (match config with True → 1 | False → 0)
let myconfig = True
let main = process [missing ornament for myconfig] 42
```

Unsafely exchanging the values of boolean, selectively

```
type ornament inverse : bool ⇒ bool with
  | True ⇒ False
  | False ⇒ True
let process = lifting process : inverse → _ → _
let myconfig = lifting myconfig : bool
let main = lifting main : bool
```

- The inverse transformation is used selectively.
- The ornamentation typechecking / inference tracks the relations between the old and new versions of bool and ensures consistency.

Enforcing more invariants

```
type exp =
  | App of exp × exp
  | Con of int
  | Abs 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
      | None → None
      | Some (Con i) → None
      | Some (App (u, v)) → None
      | Some (Abs f) →
        (match eval v with
          Some x → eval (f x) | ..))
```

Enforcing more invariants

```
type exp =
  | App of exp × exp
  | Con of int
  | Abs 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
      | None → None
      | Some (Con i) → None
      | Some (App (u, v)) → None
      | Some (Abs f) →
        (match eval v with
          Some x → eval (f x) | ..))
```

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

Enforcing more invariants

```
type exp =
  | App of exp × exp
  | Con of int
  | Abs 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
      | None → None
      | Some (Con i) → None
      | Some (App (u, v)) → None
      | Some (Abs f) →
        (match eval v with
          Some x → eval (f x) | ..))
```

```
type exp' =
  | App' of exp' × exp'
  | Val of value'
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
      | None → None
      | Some (Con' i) → None
      | Some (Abs' f) →
        (match eval' v with
          Some x → eval' (f x) | ..))
```

Enforcing more invariants

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

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

```
type ornament 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

Code specialization: sets as unit maps

A set can be seen as a `unit map`

```
type α map =
  | Mnode of α map × key × α × α map
  | Mempty
```

Code specialization: sets as unit maps

A set can be seen as a `unit map`

$$\begin{aligned}&\textbf{type}\ \alpha\ \texttt{map} =\\ &\quad|\ \text{Mnode}\ \textbf{of}\ \alpha\ \texttt{map} \times \texttt{key} \times \alpha \times \alpha\ \texttt{map}\\ &\quad|\ \text{Mempty}\end{aligned}$$

but it can use a more compact representation:

$$\begin{aligned}&\textbf{type}\ \texttt{set} =\\ &\quad|\ \text{Snode}\ \textbf{of}\ \texttt{set} \times \texttt{key} \times \texttt{set}\\ &\quad|\ \text{Sempty}\end{aligned}$$

Code specialization: sets as unit maps

A set can be seen as a `unit map`

```
type α map =
  | Mnode of α map × key × α × α map
  | Mempty
```

but it can use a more compact representation:

```
type set =
  | Snode of set × key × set
  | Sempty
```

We may automate the translation:

```
type ornament mapset : unit map ⇒ set with
  | Mnode(l,k ,(), r) ⇒ Snode(l,k,r) when l r : mapset
  | Mempty → Sempty

 lifting * with mapset
```

NB: Will keep passing extra unit parameters in auxiliary functions

- These can also be removed by ornamentation of the arguments

Code generalization: from sets to maps

```
type set =
  | Snode of set × key × set
  | Sempty
```

```
type α map =
  | Mnode of α map × key × α × α map
  | Mempty
```

```
type ornament α setmap : set ⇒ α map with
  | Snode(l,k,r) ⇒ Mnode(l,k,_,r) when l r : α setmap
  | Mempty ⇒ Sempty
```

- The ornament relation α setmap is not a function:

$$\forall \boxed{v} : \alpha, \quad \boxed{Snode(l, k, r)} \Rightarrow \boxed{Mnode(l, k, \boxed{v}, r)}$$

- The code can only be partially lifted
- The missing parts must be user provided

This is the initial idea of Conor when introducing ornaments...

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 )))
```

proj. (length) function

Ornament. relation

The ornament relation

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

- The _ 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)
```

add & append

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

forget
information
(looks) easy

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

add & append

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

Lifting
(partial)
missing
information

```
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
  with #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
  with #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
  with #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
  with #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)
```

How to proceed?

- in a principled manner—no arbitrary choices!
- so that the lifted program behaves similarly to the base one:

$$(\text{add}, \text{append}) \in \alpha\ \text{natlist} \to \alpha\ \text{natlist} \to \alpha\ \text{natlist}$$

implies:

$$\text{length (append n m)} = \text{add (length n) (length m)}$$

Code reuse by abstraction *a priori*

A design principle for modularity

Code reuse by abstraction *a priori*

A design principle for modularity

Polymorphic code
abstracts over the details
$\Lambda(\alpha, \beta)\ \ldots \lambda(x : \tau, y : \sigma)\ M$

F

A

Provide the details separately
as type and value arguments

$F\ A$

Code reuse by abstraction *a priori*

A design principle for modularity

Polymorphic code
abstracts over the details
$\Lambda(\alpha, \beta)\ \ldots \lambda(x : \tau, y : \sigma)\ M$

F

A $\qquad$ B

Provide the details separately
as type and value arguments

$F\ A$

Code reuse with a different
implementation of the details

$F\ B$

Code reuse by abstraction *a priori*

A design principle for modularity

Polymorphic code
abstracts over the details
$\Lambda(\alpha, \beta) \ \ldots \lambda(x : \tau, y : \sigma)\ M$

F

A $\qquad$ B

Provide the details separately
as type and value arguments

$F\ A$

Code reuse with a different
implementation of the details

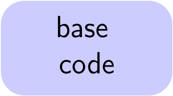
$F\ B$

Theorems for free

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

Lifting

No reasonable place for abstraction a priori



A

Lifting

Need to ornament some of the datatypes

$$\underset{A}{\boxed{\text{base code}}} \xrightarrow{?} \underset{B}{\boxed{\text{Find its lifted version given an ornament specification}}}$$

Lifting by abstraction *a posteriori*

Abstract over (depends only on) what is ornamented.

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

A_{gen}

Inference
(1)

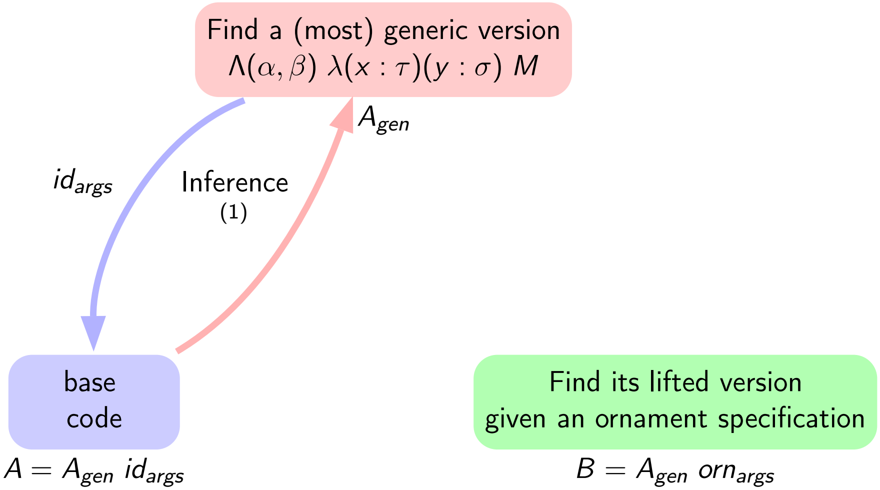
base
code

A

Find its lifted version
given an ornament specification

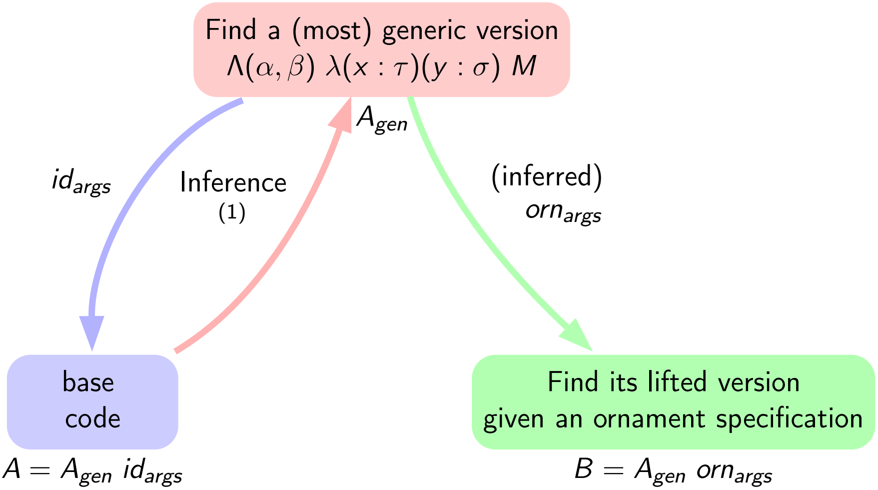
B

Lifting by abstraction *a posteriori*



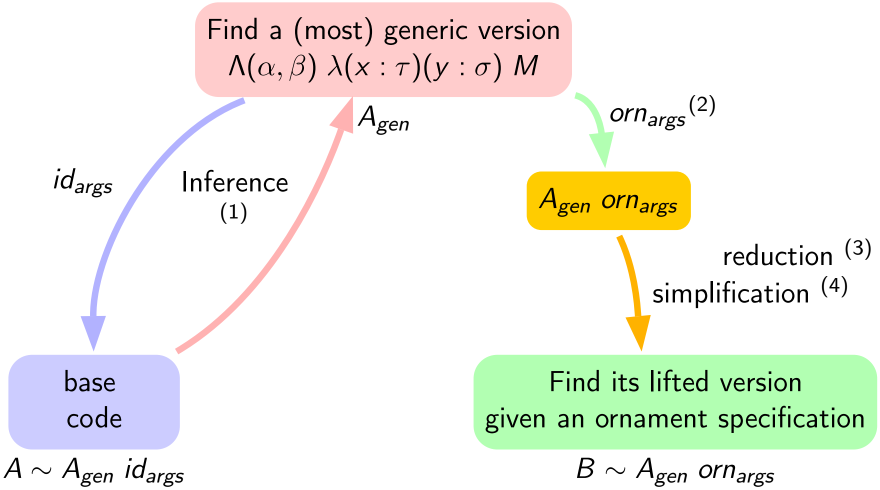
Lifting by abstraction *a posteriori*

Specialize according to the liftting specification

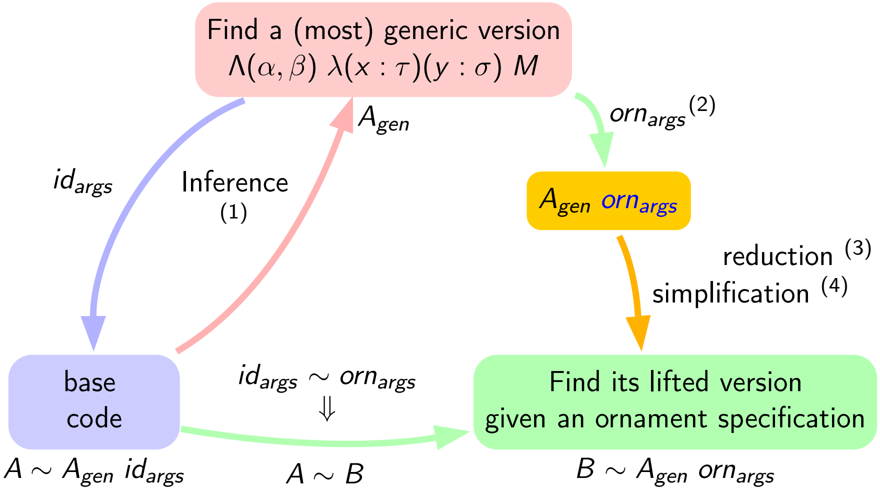


Lifting by abstraction *a posteriori*

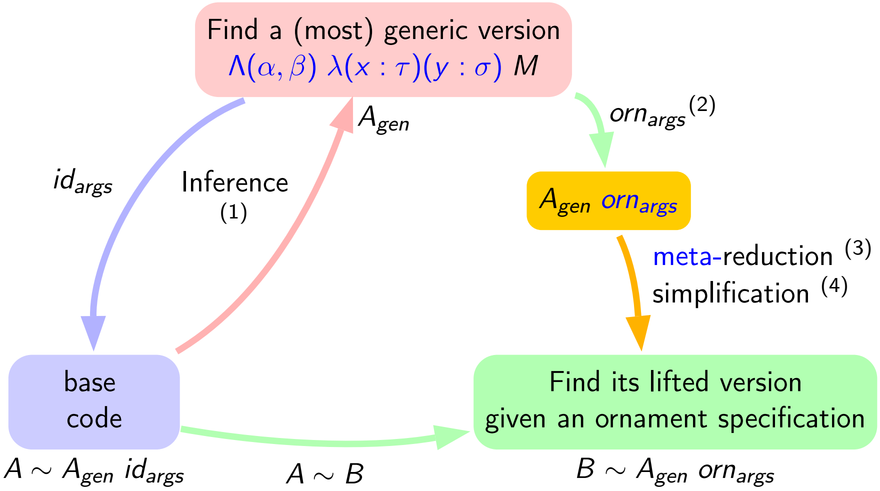
Simplify



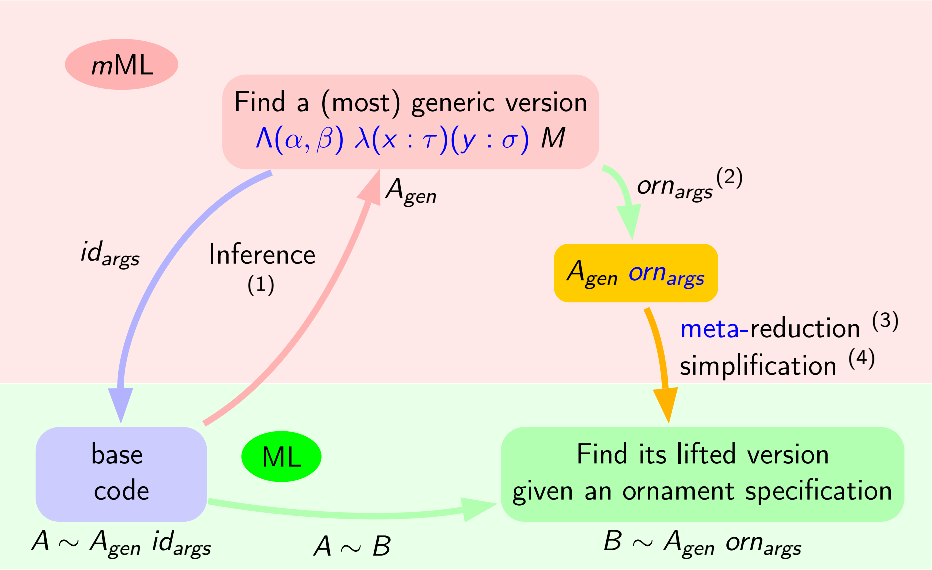
Lifting by abstraction *a posteriori*



Lifting by abstraction *a posteriori*



Lifting by abstraction *a posteriori*



Representing ornaments of nat

- 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.

```
type α natS = Z' | S' of α
```

- The ornamented datatype will piggy bag on this skeleton:

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

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

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

Representing ornaments of nat

- 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.

```
type α natS = Z' | S' of α
```

- The ornamented datatype will piggy bag on this skeleton:

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

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

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

- For convenience, we pack them in a datatype

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

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 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 orn₁  orn₂  patch =
  let rec add m n =
    match  orn₁.proj m with
      | Z' → n
      | S' m' →  orn₂.inj (S' (add m' n)) (patch m n)
  in add
```

and back to append

```
let add_gen orn₁ orn₂ patch =
  let rec add m n =
    match orn₁.proj m with
      | Z' → n
      | S' m' → orn₂.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)
```

or back to add

```
let add_gen orn1 orn2 patch =
  let rec add m n =
    match orn1.proj m with
      | Z' → n
      | S' m' → orn2.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 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)}

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

Type Inference

Needed for 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

```
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
```

Type Inference

Needed for 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 ...

```
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
```

Type Inference

Needed for 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

```
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
```

Type Inference

Needed for 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 (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
```

Staging

We need meta-reduction to

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

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

```
let add_gen = fun orn₁ orn₂ patch →
  let rec add m n =
    match orn₁.proj    m with
      | Z' → n
      | S' m' → orn₂.inj   S' (add m' n)   (patch m n)
  in add

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

Staging

We need meta-reduction to

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

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

```
let add_gen = fun orn₁ orn₂ patch ⇒#
  let rec add m n =
    match orn₁.proj # m with
      | Z' → n
      | S' m' → orn₂.inj # S' (add m' n) # (patch m n)
  in add

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

Staging

We need meta-reduction to

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

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

```
let add_gen = fun orn₁ orn₂ patch ⇛
  let rec add m n =
    match orn₁.proj # m with
      | Z' → n
      | S' m' → orn₂.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 of the lifted code

```
let add_gen orn₁ orn₂ patch ⇏
  let rec add m n =
    match orn₁.proj # m with
      | Z' → n
      | S' m' → orn₂.inj # S' (add m' n) # (patch m n)
  in add

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

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

Meta-reduction

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

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

Elimination of the encoding

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

      (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(_, xs) →

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

and reducing again

Eliminating the encoding

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

                  n



    | Cons(_, xs) →



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

Back to ML

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

Back to ML

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

Back to ML

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

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

Back to ML

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

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

Beyond ornaments

Theoretical limits of ornaments

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, *e.g.* CPS transform, defunctionalization, *etc.* deforestation
- ✘ add a new unrelated constructor to a datatype (datatype extension)

Practical limits of ornaments

Lifting is syntactic

- ✘ ornamentation points are derived from the syntax.
- ✘ unfolding of recursion

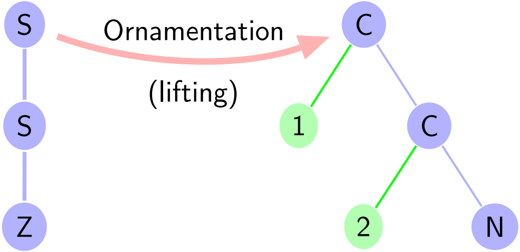
A useful scenario for unfolding of recursion

- Use (homogeneous) fix-length (long enough) lists instead of tuples to benefit from library functions (*e.g.* maps and folds).
- Lift the code back into tuples for efficiency.

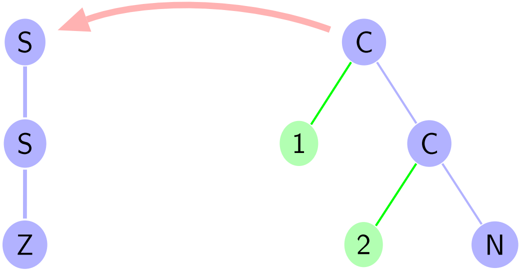
Solutions

- perform unfolding as a preprocessing
- extend the notion of syntactic lifting?

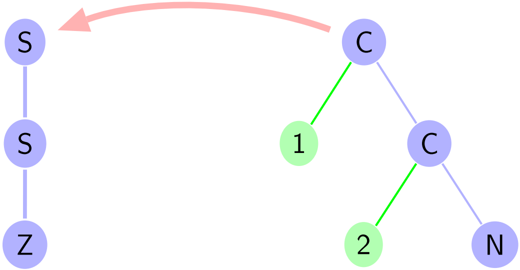
De-ornamentation



De-ornamentation



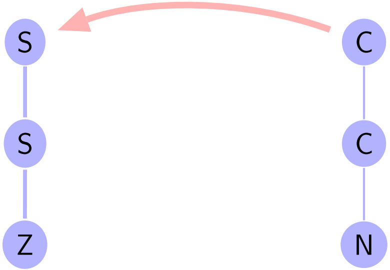
De-ornamentation



Why useful?

- undo the ornamentation...
- offer a simplified view: locations, type annotations on ASTs, *etc.*
- remove information in datatypes that became obsolete/erroneous
- change information by combination of with re-ornamentation

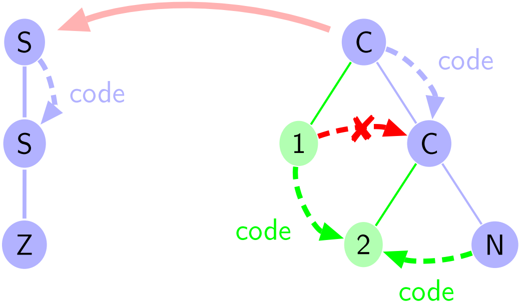
De-ornamentation



Trival case

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

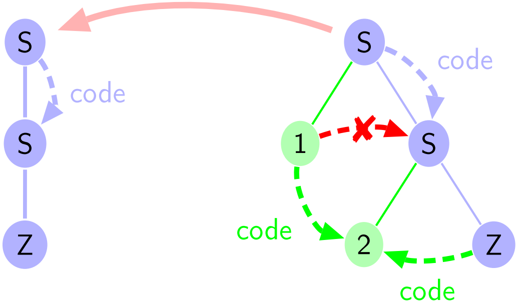
De-ornamentation



Normal case

- The source is an ornamentation or the target.
  Need to throw away the green code (should be dead code on the left)

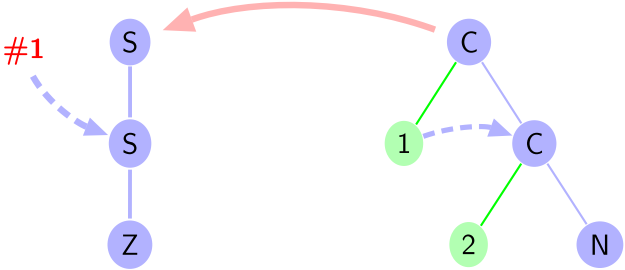
De-ornamentation



Normal case

- The source is an ornamentation or the target.
  Need to throw away the green code (should be dead code on the left)
- Related work: *Type theory in color* by Bernardy and Moulin (ICFP 2013) A type system to check (non) dependencies.
  The blue parts need to coincide exactly.

De-ornamentation



General case

- The blue may be depend on the green.
  Need code patches in the target
  to replace missed bindings and pattern matchings

Combining transformations

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

Combining transformations

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

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 pattern matching.
  - Inverse transformation that restores deep pattern matching.
  - Factor identical branches.
- Introduce / inline let bindings.

Combining transformations

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

General tooling already needed for pre/post processing

Code inference

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

Combining transformations

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

General tooling already needed for pre/post processing

Code inference

Ornamentation like transformations

- Ornamenting in several steps: complex but isomorphic transformations, followed by simpler, non-reversible ornamentations.
- Deornamentation could precede (or follow) ornamentation.
- Extensible datatypes ?
  See *Trees that grows* by Shayan Najd & Simon Peyton Jones:
  - Their solution is by abstraction a priori.
  - Abstraction *a posteriori* alternative?

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

Other useful semantic preserving transformations?

- CPS transformation, Defunctionalization, Deforestation, *etc.*
- Many compiler optimisations 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.
- 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:
    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? (boilerplate code?)

Ornaments are just one little tool

fits well within ML and could be further explored in many directions (see more at http://gallium.inria.fr/~remy/ornaments/)

Let's automate the boring parts of programming!