A Principled Approach to Ornamentation in ML

Thomas Williams, Didier Rémy

Inria - Gallium

June 15, 2018

Motivation

In a statically-typed programming language with ADTs.

Imagine we wrote an evaluator for a simple language:

```
type expr =
  | Const of int
  | Add of expr × expr
  | Mul of expr × expr
  | ...
```

```
let rec eval = function
  | Const i → i
  | Add ( u , v ) → eval u + eval v
  | Mul ( u , v ) → eval u × eval v
  | ...
```

Motivation

In a statically-typed programming language with ADTs.

Imagine we wrote an evaluator for a simple language:

```
type expr =
 | Const of int
 | Add of expr × expr
 | Mul of expr × expr
 | ...
```

```
let rec eval = function
 | Const i → i
 | Add ( u , v ) → eval u + eval v
 | Mul ( u , v ) → eval u × eval v
 | ...
```

We change the representation of expressions:

```
type binop' = Add' | Mul'
type expr' =
 | Const' of int
 | Binop' of binop' × expr' × expr'
 | ...
```

Motivation

In a statically-typed programming language with ADTs.

Imagine we wrote an evaluator for a simple language:

```
type expr =
 | Const of int
 | Add of expr × expr
 | Mul of expr × expr
 | ...
```

```
let rec eval = function
 | Const i → i
 | Add ( u , v ) → eval u + eval v
 | Mul ( u , v ) → eval u × eval v
 | ...
```

We change the representation of expressions:

```
type binop' = Add' | Mul'
type expr' =
 | Const' of int
 | Binop' of binop' × expr' × expr'
 | ...
```

What happens to the code we have already written ?

Use the types

Our first instinct is to compile the code and trust the typechecker:

```
let rec eval : expr → int = function
  | Const i → i
  | Add(u, v) → eval u + eval v
  | Mul(u, v) → eval u × eval v
  | ...
```

```
let rec eval' : expr' → int = function
  | Const i → i
  | Add(u, v) → eval' u + eval' v
  | Mul(u, v) → eval' u × eval' v
  | ...
```

Use the types

Our first instinct is to compile the code and trust the typechecker:

```
let rec eval : expr → int = function
  | Const i → i
  | Add(u, v) → eval u + eval v
  | Mul(u, v) → eval u × eval v
  | ...
```

```
let rec eval' : expr' → int = function
  | Const i → i
  | Add(u, v) → eval' u + eval' v
  | Mul(u, v) → eval' u × eval' v
  | ...
```

Use the types

Our first instinct is to compile the code and trust the typechecker:

```
let rec eval : expr → int = function
 | Const i → i
 | Add(u, v) → eval u + eval v
 | Mul(u, v) → eval u × eval v
 | ...
```

```
let rec eval' : expr' → int = function
 | Const' i → i
 | Add(u, v) → eval' u + eval' v
 | Mul(u, v) → eval' u × eval' v
 | ...
```

Use the types

Our first instinct is to compile the code and trust the typechecker:

```
let rec eval : expr → int = function
  | Const i → i
  | Add(u, v) → eval u + eval v
  | Mul(u, v) → eval u × eval v
  | ...
```

```
let rec eval' : expr' → int = function
  | Const' i → i
  | Add(u, v) → eval' u + eval' v
  | Mul(u, v) → eval' u × eval' v
  | ...
```

Use the types

Our first instinct is to compile the code and trust the typechecker:

```
let rec eval : expr → int = function
  | Const i → i
  | Add(u, v) → eval u + eval v
  | Mul(u, v) → eval u × eval v
  | ...
```

```
let rec eval' : expr' → int = function
  | Const' i → i
  | Binop'(Add',u, v) → eval' u + eval' v
  | Mul(u, v) → eval' u × eval' v
  | ...
```

Use the types

Our first instinct is to compile the code and trust the typechecker:

```
let rec eval : expr → int = function
 | Const i → i
 | Add(u, v) → eval u + eval v
 | Mul(u, v) → eval u × eval v
 | ...
```

```
let rec eval' : expr' → int = function
 | Const' i → i
 | Binop'(Add',u, v) → eval' u + eval' v
 | Mul(u, v) → eval' u × eval' v
 | ...
```

Use the types

Our first instinct is to compile the code and trust the typechecker:

```
let rec eval : expr → int = function
 | Const i → i
 | Add(u, v) → eval u + eval v
 | Mul(u, v) → eval u × eval v
 | ...
```

```
let rec eval' : expr' → int = function
 | Const' i → i
 | Binop'(Add',u, v) → eval' u + eval' v
 | Binop'(Mul',u, v) → eval' u × eval' v
 | ...
```

Use the types

Our first instinct is to compile the code and trust the typechecker:

```
let rec eval : expr → int = function
  | Const i → i
  | Add(u, v) → eval u + eval v
  | Mul(u, v) → eval u × eval v
  | ...
```

```
let rec eval' : expr' → int = function
  | Const' i → i
  | Binop'(Add',u, v) → eval' u + eval' v
  | Binop'(Mul',u, v) → eval' u × eval' v
  | ...
```

- Manual process
  - Long
  - Error prone
- The typechecker misses some places where a change is necessary (exchange fields with the same type)

Let's do better

Linking types

- In our mental model, the old type and the new type are linked
- Let's keep track of this link
- A restricted class of transformation: ornaments, introduced by Conor McBride
- A *coherence* property for lifting functions

Related work

- Conor McBride, Pierre Dagand
- Hsiang-Shang Ko, Jeremy Gibbons
- Encoded in Agda
  - needs dependent types
  - and powerful encodings

What can we do in ML?

Ornaments in ML

- Define ornamentes as a primitive concept
- The correctness of the lifting is not internal anymore
- Restrict the transformation (stick to the syntax) to automate the lifting
- Prove the correctness of our transformation

We built a (prototype) tool for lifting

- implements this transformation
- on a restricted subset of ML

Use the types

Instead, define a relation:

```
type expr =
  | Const of int
  | Add of expr × expr
  | Mul of expr × expr
  | ...
```

```
type binop' = Add' | Mul'
type expr' =
  | Const' of int
  | Binop' of binop' × expr' × expr'
  | ...
```

Use ~~the types~~ **ornaments**

Instead, define a relation:

```
type expr =
  | Const of int
  | Add of expr × expr
  | Mul of expr × expr
  | ...
```

```
type binop' = Add' | Mul'
type expr' =
  | Const' of int
  | Binop' of binop' × expr' × expr'
  | ...
```

```
type ornament oexpr : expr ⇒ expr' with
  | Const i  ⇒ Const' i
  | Add(u, v) ⇒ Binop'(Add', u', v')  / when (u, u') ∈ oexpr
  | Mul(u, v) ⇒ Binop'(Mul', u', v')  \  and (v, v') ∈ oexpr
  | ...
```

Use ~~the types~~ **ornaments**

Instead, define a relation:

```
type expr =
  | Const of int
  | Add of expr × expr
  | Mul of expr × expr
  | ...
```

```
type binop' = Add' | Mul'
type expr' =
  | Const' of int
  | Binop' of binop' × expr' × expr'
  | ...
```

```
type ornament oexpr : expr ⇒ expr' with
  | Const i ⇒ Const' i
  | Add(u, v) ⇒ Binop'(Add', u, v) with u v : oexpr
  | Mul(u, v) ⇒ Binop'(Mul', u, v) with u v : oexpr
  | ...
```

Use ornaments

```
let rec eval = function
| Const i → i
| Add ( u , v ) → eval u + eval v
| Mul ( u , v ) → eval u × eval v
|  ...
```

Use ornaments

```
let rec eval = function
| Const i → i
| Add ( u , v ) → eval u + eval v
| Mul ( u , v ) → eval u × eval v
| ...
```

```
let eval' = lifting eval : oexpr → int
```

Use ornaments

```
let rec eval = function
| Const i → i
| Add ( u , v ) → eval u + eval v
| Mul ( u , v ) → eval u × eval v
|  ...
```

```
let eval' = lifting eval : oexpr → int
```

$$(u, u') \in \mathsf{oexpr} \implies \mathsf{eval}\ u = \mathsf{eval}'\ u'$$

Use ornaments

```
let rec eval = function
| Const i → i
| Add ( u , v ) → eval u + eval v
| Mul ( u , v ) → eval u × eval v
| ...
```

```
let rec eval' = function
| Const' i → i
| Binop'(Add', u, v) → eval' u + eval' v
| Binop'(Mul', u, v) → eval' u × eval' v
| ...
```

```
let eval' = lifting eval : oexpr → int
```

$$(u, u') \in \mathsf{oexpr} \implies \mathsf{eval}\ u = \mathsf{eval}'\ u'$$

Use ornaments

```
let rec eval = function
| Const i → i
| Add ( u , v ) → eval u + eval v
| Mul ( u , v ) → eval u × eval v
| ...
```

```
let rec eval' = function
| Const' i → i
| Binop'(Add', u, v) → eval' u + eval' v
| Binop'(Mul', u, v) → eval' u × eval' v
| ...
```

```
let eval' = lifting eval : oexpr → int
```

$$(u, u') \in \text{oexpr} \implies \text{eval } u = \text{eval}' \ u'$$

- Clear specification of the function we want
- Also gives a specification for our tool
- In this case, since the relation is one-to-one, the result is unique

Specialization

From lists to homogeneous tuples:

(not in the version available online)

type α list $=$ $\mid$ Nil $\mid$ Cons **of** $\alpha \times \alpha$ list	**type** α triple $=$ ($\alpha \times \alpha \times \alpha$)

Specialization

From lists to homogeneous tuples:

(not in the version available online)

type α list =
  | Nil
  | Cons **of** $\alpha \times \alpha$ list

type α triple =
  ($\alpha \times \alpha \times \alpha$)

type ornament α list3 : α list $\to \alpha$ pair **with**
  | Cons (x0, Cons(x1, Cons(x2, Nil))) $\Rightarrow$ (x0, x1, x2)
  | _ $\Rightarrow$ $\sim$

Specialization

From lists to homogeneous tuples:
(not in the version available online)

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

```
type α triple =
  ( α × α × α )
```

```
type ornament α list3 : α list → α pair with
 | Cons (x0, Cons( x1, Cons( x2, Nil ))) ⇒ ( x0, x1, x2 )
 | _ ⇒ ∼
```

- More than simply reorganizing: *restricts* the possible values.

Specialization

```
let rec map f = function
  | Nil -> Nil
  | Cons (x, xs) -> Cons (f x, xs)
```

Specialization

```
let rec map f = function
  | Nil → Nil
  | Cons (x, xs) → Cons (f x, xs)
```

let map_triple = **lifting** map : $(\alpha \to \beta) \to \alpha$ list3 $\to \beta$ list3

Specialization

```
let rec map f = function
  | Nil → Nil
  | Cons (x, xs) → Cons (f x, xs)
```

```
let rec map_triple f (x0,x1,x2) =
  let (y1,y2) = map_pair f (x1, x2) in
  (f x0, y1, y2)
and map_pair f (x1,x2) =
  (f x1, map_one f x2)
and map_one f x2 = f x2
```

```
let map_triple = lifting map : (α → β) → α list3 → β list3
```

- The map function has been unfolded

Specialization

```
let rec map f = function
  | Nil → Nil
  | Cons (x, xs) → Cons (f x, xs)
```

```
let map_triple f (x0, x1, x2) =
 (f x0, f x1, f x2)
```

let map_triple = **lifting** map : $(\alpha \to \beta) \to \alpha$ list3 $\to \beta$ list3

- The `map` function has been unfolded
- We could automatically remove the noise

Specialization

```
let rec map f = function
  | Nil → Nil
  | Cons (x, xs) → Cons (f x, xs)
```

```
let map_triple f (x0, x1, x2) =
 (f x0, f x1, f x2)
```

```
let map_triple = lifting map : (α → β) → α list3 → β list3
```

- The map function has been unfolded
- We could automatically remove the noise
- Exhibits invariant that was already present in the code
- Allows better representation

Adding data

type nat $=$
$\mid$ Z
$\mid$ S **of** nat

type α list $=$
$\mid$ Nil
$\mid$ Cons **of** $\alpha \times \alpha$ list

Adding data

```
type nat =
  | Z
  | S of nat
```

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

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

Adding data

```
type nat =
 | Z
 | S of nat
```

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

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

Additional data: the relation is not one-to-one anymore.

Adding data: patches

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

Adding data: patches

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

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

Adding data: patches

```
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(_,m') → Cons(#2, append m' n)
```

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

Adding data: patches

```
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)
```

```
let append = lifting add : α natlist → α natlist → α natlist
 | #2 <- (match m with Cons(x,_) -> x)
```

A user-provided *patch* describing the additional information.

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

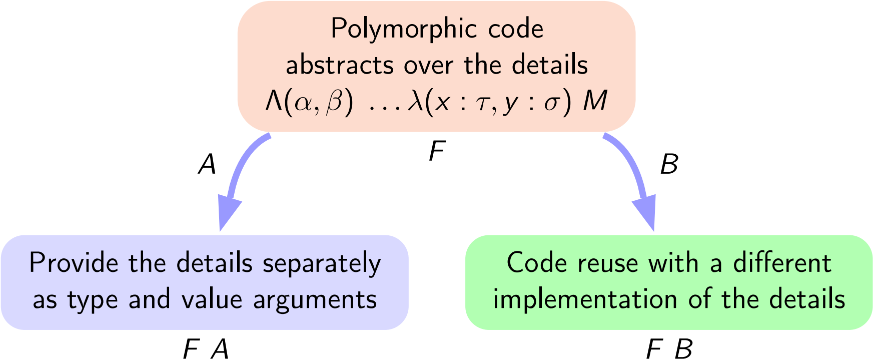
A

Provide the details separately
as type and value arguments

$F\ A$

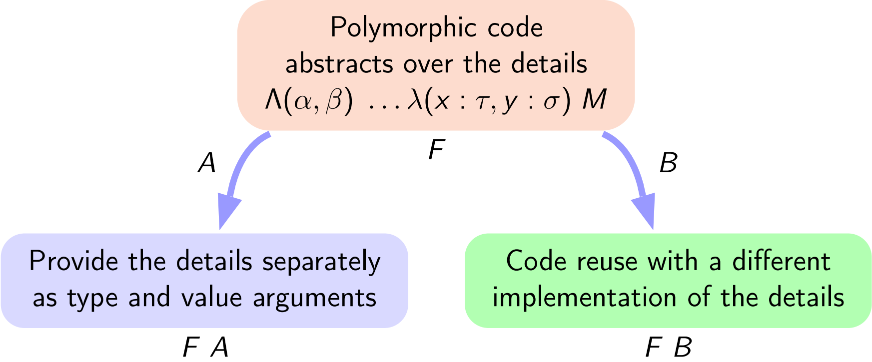
Code reuse by abstraction *a priori*

A design principle for modularity



Code reuse by abstraction *a priori*

A design principle for modularity

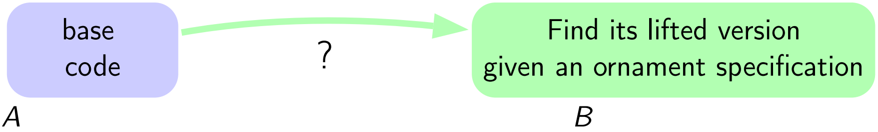


Theorems for free

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

Lifting

Need to ornament some of the datatypes



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)

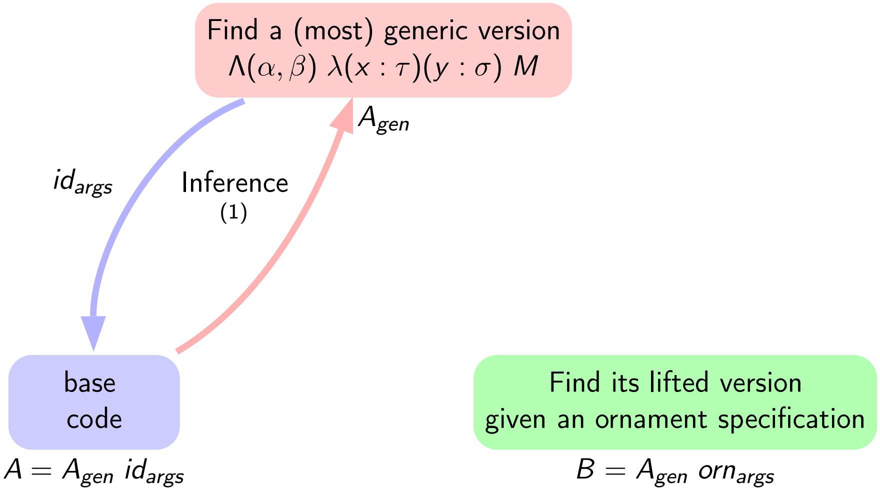
base
code

A

Find its lifted version
given an ornament specification

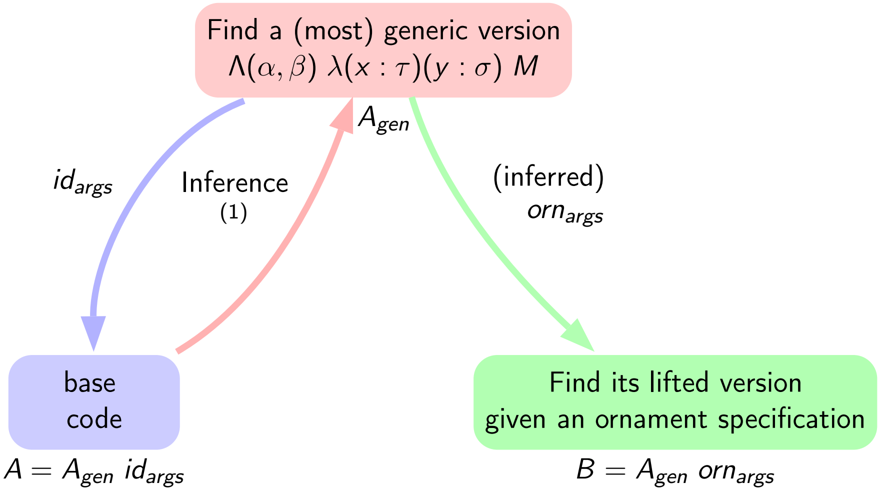
B

Lifting by abstraction *a posteriori*



Lifting by abstraction *a posteriori*

Specialize according to the liftting specification



Example

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

Example

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

```
let prj = function
  | Z → Z' | S x → S' x
let inj x p = match x with
  | Z' → Z | S' x → S x
let patch _ _ = ()
let add = add_gen prj inj patch
```

Example

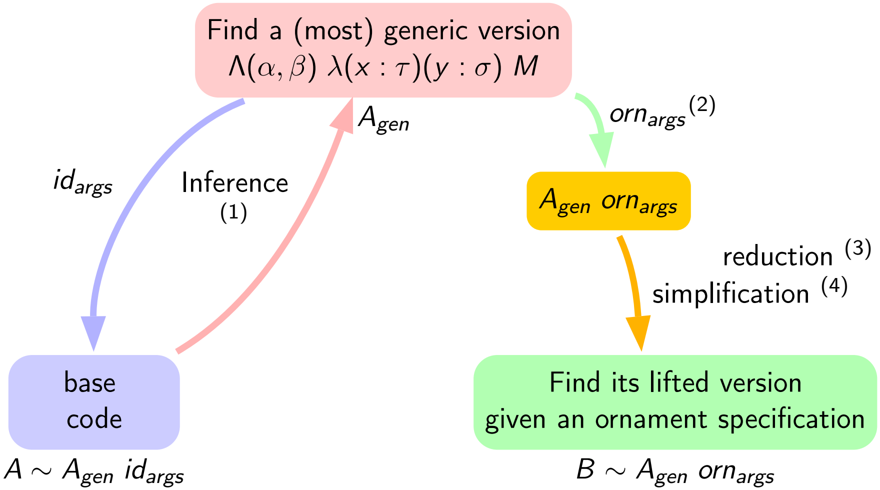
```
let add_gen prj inj patch =
  let rec add m n =
    match prj m with
      | Z' → n
      | S' m' → inj (add' m' n) (patch m n)
```

```
let prj = function
  | Z → Z' | S x → S' x
let inj x p = match x with
  | Z' → Z | S' x → S x
let patch _ _ = ()
let add = add_gen prj inj patch
```

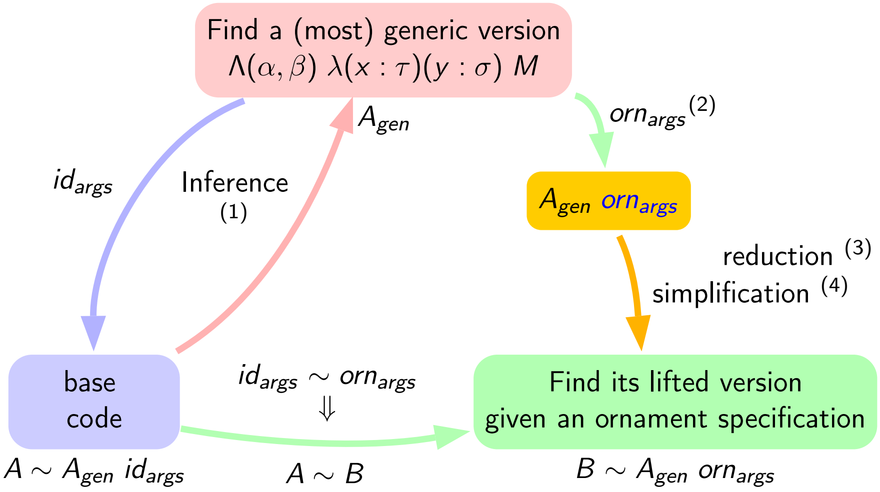
```
let prj = function
  | Nil → Z' | Cons(_,x) → S' x
let inj x p = match x with
  | Z' → Nil | S' x → Cons(p,x)
let patch (Cons(x,_)) _ = x
let append = add_gen prj inj patch
```

Lifting by abstraction *a posteriori*

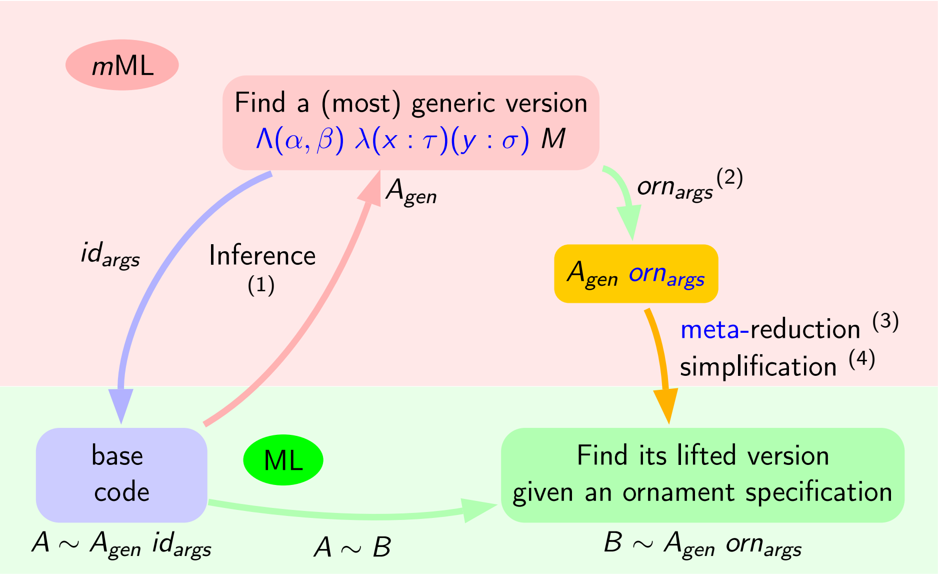
Simplify



Lifting by abstraction *a posteriori*



Lifting by abstraction *a posteriori*



Implementation

Prototype

- On a small subset of OCaml
- Precisley follows the process outlined here
- Available online: `http://gallium.inria.fr/~remy/ornaments`
- ... with many more examples.

Patches

- By property-based code inference?

  append (Cons(x, _)) _ = Cons(x, _)

In the paper

- The intermediate language *m*ML with dependent types
- Conditions that guarantee we can simplify *m*ML back to ML
- An encoding of ornaments in *m*ML
- A logical relation on *m*ML, and an interpretation of ornaments
- A formal description of the lifting
- A proof that lifted terms are indeed related at the correct type

Future work

- New implementation, with support for most of OCaml
- Support for GADTs
- How to write robust patches?
- Formal results in the presence of effects

Conclusion

- A principled way of transforming programs along ornaments
- Through abstraction and specialization
- Could this be generalized to other transformations?