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:

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

let rec eval =
| 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:

```ocaml
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:

```ocaml
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
| ...
```

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

```ocaml
type binop' = Add' | Mul'
```
Motivation

In a statically-typed programming language with ADTs.

Imagine we wrote an evaluator for a simple language:

```ocaml
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:

```ocaml
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:

```latex
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:

```plaintext
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:

```ocaml
definition

let rec eval : expr \rightarrow int = function
| Const i \rightarrow i
| Add(u, v) \rightarrow eval u + eval v
| Mul(u, v) \rightarrow eval u \times eval v
| ...

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

# Use the types

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

```ocaml
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:

```plaintext
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:

```ocaml
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:

<table>
<thead>
<tr>
<th>let rec eval : expr → int = function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const i → i</td>
</tr>
<tr>
<td>Add(u, v) → eval u + eval v</td>
</tr>
<tr>
<td>Mul(u, v) → eval u × eval v</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>let rec eval' : expr' → int = function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const' i → i</td>
</tr>
<tr>
<td>Binop'(Add',u, v) → eval' u + eval' v</td>
</tr>
<tr>
<td>Binop'(Mul',u, v) → eval' u × eval' v</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

- 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:

\[
\textbf{type} \ \text{expr} = \\
\quad \text{Const} \ of \ \text{int} \\
\quad \text{Add} \ of \ \text{expr} \times \ \text{expr} \\
\quad \text{Mul} \ of \ \text{expr} \times \ \text{expr} \\
\quad \ldots
\]

\[
\textbf{type} \ \text{binop} = \\
\quad \text{Add} \\
\quad \text{Mul}
\]

\[
\textbf{type} \ \text{expr}' = \\
\quad \text{Const}' \ of \ \text{int} \\
\quad \text{Binop}' \ of \ \text{binop}' \times \ \text{expr}' \times \ \text{expr}' \\
\quad \ldots
\]
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:

```plaintext
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

```ocaml
let rec eval = function
  | Const i -> i
  | Add (u, v) -> eval u + eval v
  | Mul (u, v) -> eval u * eval v
  | ...
```
Use ornaments

```ocaml
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') ∈ oexpr  ⇒  eval u = eval' u'
```
Use ornaments

```ocaml
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') ∈ oexpr ⇒ eval u = eval' u'
```
Use ornaments

\[
\begin{align*}
\text{let rec } \text{eval} &= \text{function} \\
&| \text{Const } i \rightarrow i \\
&| \text{Add } (u, v) \rightarrow \text{eval } u + \text{eval } v \\
&| \text{Mul } (u, v) \rightarrow \text{eval } u \times \text{eval } v \\
&| \ldots
\end{align*}
\]

\[
\begin{align*}
\text{let rec } \text{eval}' &= \text{function} \\
&| \text{Const }' i \rightarrow i \\
&| \text{Binop}'(\text{Add}', u, v) \rightarrow \text{eval }' u + \text{eval }' v \\
&| \text{Binop}'(\text{Mul}', u, v) \rightarrow \text{eval }' u \times \text{eval }' v \\
&| \ldots
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{eval}' &= \text{lifting } \text{eval} : \text{oexpr} \rightarrow \text{int} \\
(u, u') \in \text{oexpr} &\Rightarrow \text{eval } u = \text{eval}' u'
\end{align*}
\]

- 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 $\alpha$ list = |
| Nil |
| Cons of $\alpha \times \alpha$ list |

| type $\alpha$ triple = |
| ( $\alpha \times \alpha \times \alpha$ ) |
Specialization

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

\[
\text{type } \alpha \text{ list } =
\begin{align*}
&| \text{Nil} \\
&| \text{Cons of } \alpha \times \alpha \text{ list}
\end{align*}
\]

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

\[
\text{type ornament } \alpha \text{ list3 } : \alpha \text{ list } \to \alpha \text{ pair with}
\begin{align*}
&| \text{Cons } (x0, \text{Cons}( x1, \text{Cons}( x2, \text{Nil } ))) \Rightarrow ( x0, x1, x2 ) \\
&| _\Rightarrow \sim
\end{align*}
\]
Specialization

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

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

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

\[
\text{type ornament } \alpha \text{ list3 : } \alpha \text{ list } \rightarrow \alpha \text{ pair with} \\
| \text{Cons (} x0, \text{Cons(} x1, \text{Cons(} x2, \text{Nil } ))) \Rightarrow ( x0, x1, x2 ) \\
| _ \Rightarrow \sim
\]

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

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

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

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

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

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

\[
\text{type } \text{nat} = \\
\mid \ Z \\
\mid \ S \text{ of } \text{nat} \\
\]

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

\[
\text{type ornament } \alpha \text{ natlist} : \text{nat } \Rightarrow \alpha \text{ list with} \\
\mid Z \Rightarrow \text{Nil} \\
\mid S \text{ tail } \Rightarrow \text{Cons } (_\_ , \text{tail} ) \text{ when tail : } \alpha \text{ natlist} \\
\]
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.
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

<table>
<thead>
<tr>
<th>let rec add m n = match m with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z → n</td>
</tr>
<tr>
<td>S m' → S (add m' n)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>let rec append m n = match m with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil → n</td>
</tr>
<tr>
<td>Cons(x, m') → Cons(x, append m' n)</td>
</tr>
</tbody>
</table>

```ocaml
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
\[ \Lambda(\alpha, \beta) \ldots \lambda(x : \tau, y : \sigma) M \]

Provide the details separately as type and value arguments

\[ \text{Provide the details separately as type and value arguments} \]

\[ F A \]
Code reuse by abstraction *a priori*

A design principle for modularity

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

Provide the details separately as type and value arguments

Code reuse with a different implementation of the details
Code reuse by abstraction \textit{a priori}

A design principle for modularity

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

Provide the details separately as type and value arguments
\[ \text{Provide the details separately as type and value arguments} \]

Code reuse with a different implementation of the details
\[ \text{Code reuse with a different implementation of the details} \]

\textbf{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

base code

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

Abstract over (depends only on) what is ornamented.

\[
\Lambda(\alpha, \beta) \lambda(x : \tau)(y : \sigma) M
\]

Inference (1)

Find a (most) generic version

Find its lifted version given an ornament specification

\( A \)

\( B \)
Lifting by abstraction *a posteriori*

Find a (most) generic version

$$\Lambda(\alpha, \beta) \lambda(x : \tau)(y : \sigma) \; M$$

Inference

$$A_{gen} = A_{gen} \; id_{args}$$

Find its lifted version given an ornament specification

$$B = A_{gen} \; orn_{args}$$
Lifting by abstraction *a posteriori*

Specialize according to the lifting specification

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

Find its lifted version given an ornament specification

\[ B = A_{gen} orn_{args} \]
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

```plaintext
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

```ocaml
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
```

```ocaml
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

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

Inference (1)

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

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

Id args

Inference

\[ A_{gen} \]

\[ A_{gen} \ orn_{args} \]

Reduction (3)

Simplification (4)
Lifting by abstraction *a posteriori*

Find a (most) generic version

\[ \Lambda(\alpha, \beta) \lambda(x : \tau)(y : \sigma) M \]

**Inference**

\[ A \sim A_{gen} \]

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

**Find its lifted version**

given an ornament specification

\[ B \sim A_{gen} orn_{args} \]
Lifting by abstraction \textit{a posteriori}

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

Find its lifted version given an ornament specification

\[
A \sim A_{gen} \sim B \sim A_{gen} \text{orn}_{args}
\]
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?
  \[ \text{append (Cons(x, _)) _ = Cons(x, _)} \]
In the paper

- The intermediate language $mML$ with dependent types
- Conditions that guarantee we can simplify $mML$ back to ML
- An encoding of ornaments in $mML$
- A logical relation on $mML$, 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?